J of Parallel and Distributed Computing, 2007, vol. 67(3), pp. 346–359

Distributed Computation of the $k$nn Graph for Large High-Dimensional Point Sets

Erion Plaku [a], Lydia E. Kavraki [a,*]

[a] *Rice University, Department of Computer Science, Houston, Texas 77005, USA*

Abstract

High-dimensional problems arising from robot motion planning, biology, data mining, and geographic information systems often require the computation of k nearest neighbor ($k$nn) graphs. The $k$nn graph of a data set is obtained by connecting each point to its k closest points. As the research in the above-mentioned fields progressively addresses problems of unprecedented complexity, the demand for computing $k$nn graphs based on arbitrary distance metrics and large high-dimensional data sets increases, exceeding resources available to a single machine. In this work we efficiently distribute the computation of $k$nn graphs for clusters of processors with message passing. Extensions to our distributed framework include the computation of graphs based on other proximity queries, such as approximate $k$nn or range queries. Our experiments show nearly linear speedup with over one hundred processors and indicate that similar speedup can be obtained with several hundred processors.

Key words: Nearest neighbors; Approximate nearest neighbors; $k$nn graphs; Range queries; Metric spaces; Robotics; Distributed and Parallel Algorithms

1 Introduction

The computation of proximity graphs for large high-dimensional data sets in arbitrary metric spaces is often necessary for solving problems arising from robot motion planning [14,31,35,40,45, 46, 56], biology [3, 17, 33, 39, 42, 50, 52], pattern recognition [21], data mining [18, 51], multimedia systems [13], geographic information systems [34, 43, 49], and other research fields. Proximity graphs are typically based on nearest neighbor relations. The nearest neighbor or, in general, the k nearest neighbor ($k$nn) graph of a data set is obtained by connecting each point in the data set to its k closest points from the data set, where a distance metric [11] defines closeness.

As an example, research in robot motion planning has in recent years focused on the development of sampling-based motion planners motivated by the success of the Probabilistic RoadMap (`PRM`) method [31] for solving problems involving multiple and highly complex robots. Sampling-based motion planning algorithms [14,31,41,45,46] rely on an efficient sampling of the solution space and construction of the $k$nn graph for the sampled points. The $k$nn graph captures the connectivity of the solution space and is used to find paths that allow robots to move from one point in the environment to another while satisfying certain constraints such as avoiding collision with obstacles.

* Corresponding author.
Email address: `kavraki@cs.rice.edu` (Lydia E. Kavraki).

Article published in J. Parallel Distrib. Comput. (2007), vol. 67(3), pp. 346–359

The use of $k$nn graphs combined with probabilistic methods also has promise in the study of protein folding [3, 17, 52]. Other applications of $k$nn searches and graphs in biology include classifying tumors based on gene expression profiles [42], finding similar protein sequences from a large database [33, 39], and docking of molecules for computer-assisted drug design [50].

In pattern recognition and data mining, nearest neighbors are commonly used to classify objects based upon observable features [13, 18, 21, 51]. During the classification process, certain features are extracted from the unknown object and the unknown object is classified based on the features extracted from its k nearest neighbors.

In geographic information systems [34, 43, 49], a commonly encountered query is the computation of $k$nn objects for points in space. Examples of queries include finding "the nearest gas stations from my location" or "the five nearest stars to the north star."

As research in robot motion planning, biology, data mining, geographic information systems, and other scientific fields progressively addresses problems of unprecedented complexity, the demand for computing $k$nn graphs based on arbitrary distance metrics and large high-dimensional data sets increases, exceeding resources available to single machines [48]. In this paper, we address the problem of computing the $k$nn graph utilizing multiple processors communicating via message passing in a cluster system with no-shared memory and where the amount of memory available to each processor is limited. Our model of computation is motivated by many scientific applications where the computation of the $k$nn graph is needed for massive data sets whose size is dozens or even hundreds of gigabytes. Since such applications are also computationally intensive and the $k$nn graph constitutes only one stage of the overall computation, it is important to fit as much of the data as possible in the main memory of each available processor in order to reduce disk operations. The challenge lies in developing efficient distributed algorithms for the computation of the $k$nn graph, since nearest-neighbors computations depend on all the points in the data set and not just on the points stored in the main memory of a processor.

The contribution of our work is to develop a distributed decentralized framework that efficiently computes the $k$nn graph for extensively large data sets. Our distributed framework utilizes efficient communication and computation schemes to compute partial $k$nn results, collect information from the partial results to eliminate certain communication and computation costs, and gather partial results to compute $k$nn queries for each point in the data set. The partial $k$nn results are computed by constructing and querying sequential $k$nn data structures [8, 23, 26–28, 55] stored in the main memory of each processor. Our distributed framework works with any sequential $k$nn data structure and is also capable of taking advantage of specific implementations of sequential $k$nn data structures to improve the overall efficiency of the distribution.

Our distributed framework supports the efficient computation by hundreds of processors of very large $k$nn graphs consisting of millions of points with hundreds of dimensions and arbitrary distance metrics. Our experiments show nearly linear speedup on one hundred and forty processors and indicate that similar speedup can be obtained on several hundred processors.

Our distributed framework is general and can be extended in many ways. One possible extension is the computation of graphs based on approximate $k$nn queries [4, 16, 32, 36, 38, 47, 54]. In an approximate $k$nn query, instead of computing exactly the k closest points to a query point, it suffices to compute k points that are within a $(1 + \epsilon)$ hypersphere from the k-th closest point to the query point. Approximate $k$nn queries provide a trade-off between efficiency of computation and quality of neighbors computed for each point. Another possible extension is the computation of graphs based on range queries [5, 9, 19]. In a range query, we are interested in computing all the points in the data set that are within some predefined distance from a query point. Range

queries are another type of proximity queries with a wide applicability.

The rest of the paper is organized as follows. In section 2 we review related work. In section 3 we formally define the problem of computing the $k$nn graph, describe the distributed model of computation, and then present a distributed algorithm for computing $k$nn graphs. We also discuss how to extend our distributed framework to compute graphs based on approximate $k$nn and range queries. In section 4 we describe the experimental setup, the data sets for testing the efficiency of the distribution, and the results obtained. We conclude in section 5 with a discussion on the distributed algorithm.

2 Related Work

The computation of the $k$nn graph of a data set S is based on the computation of the k nearest neighbors for each point $s \in S$. The computation of the k nearest neighbors for a point $s \in S$ is referred to as a $k$nn query and s is referred to as a query point. In this section we review work related to the sequential and distributed computation of $k$nn queries and $k$nn graphs.

Motivated by challenging problems in many research fields, a large amount of work has been devoted to the development of efficient algorithms for the computation of $k$nn queries [4, 8, 15, 23, 26–28, 36, 55]. The computation of $k$nn queries usually proceeds in two stages. During a preprocessing stage, a data structure T_S is constructed to support the computation of $k$nn queries from a given a data set S. During the query stage, searches are performed on T_S to compute the k nearest neighbors of a query point s, denoted $T_S(s, k)$. The same data structure T_S can be used for the computation of the k nearest neighbors from the data set S to any query point; T_S is modified only when points are added to or removed from S. In $k$nn literature, references to the $k$nn data structure encompass both the structure of the data, T_S, and the algorithms to query the data structure to obtain the k nearest neighbors, $T_S(s, k)$.

Researchers have developed many efficient data structures for the computation of $k$nn queries based on Euclidean distance metrics [4, 15, 23, 26, 36]. However, many challenging applications stemming from robot motion planning, biology, data mining, geographic information systems, and other fields, require the computation of $k$nn queries based on arbitrary distance metrics. Although a challenging problem, progress has been made towards the development of efficient data structures supporting $k$nn queries based on arbitrary distance metrics [8, 28, 55]. During the preprocessing stage, usually a metric tree [53] is constructed hierarchically as the data structure T_S. One or more points are selected from the data set and associated with the root node. Then the distances from the points associated with the root node to the remaining points in the data set are computed and used to partition the remaining points in the data set into several smaller sets. Each set in the partition is associated with a branch extending from the root node. The extension process continues until the cardinality of the set in the partition is smaller than some predefined constant. For example, in the construction of vantage-point trees [55], the median sphere centered at the point associated with the root is used to partition the data set into points inside and outside the sphere, while in the construction of `GNAT` [8], the data set is clustered using an approximate k-centers algorithm and each center is associated with a branch extending from the root. Figure 1 illustrates the construction of a metric tree for `GNAT`. During the query stage, precomputed distances, lower and upper bounds on the distances between points associated with the nodes of the tree, triangle inequality, and other properties and heuristics are used to prune certain branches of the tree in order to improve the efficiency of computing $k$nn queries.

In addition to the development of more efficient sequential $k$nn algorithms, distributed algorithms that take full advantage of all the available resources provide a viable alternative for the

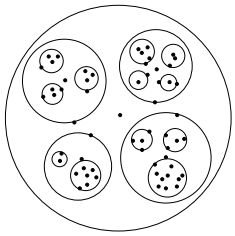


Fig. 1. The hierarchical structure of the metric tree in `GNAT` [8]. Illustrated are the points of the data set and the spheres, represented as circles, associated with each node of the tree.

computation of $k$nn queries. Research on distributed $k$nn algorithms mostly focuses on the development of algorithms for computing $k$nn queries based on Euclidean distance metrics. The work in [43] addresses parallel processing of $k$nn queries in declustered spatial data in 2-dimensional Euclidean spaces for multiple processors communicating through a network. The work is based on the parallelization of nearest neighbor search using the R-tree data structure [26] and could be generalized to d-dimensional Euclidean spaces. The work in [29] shows how to compute $k$nn queries for 3-dimensional Euclidean point sets for a shared-memory grid environment utilizing general grid middle-ware for data intensive problems. In [49], $k$nn queries are computed efficiently in an integration middleware that provides federated access to numerous loosely coupled, autonomous data sources connected through the internet.

There is also research on distributed algorithms for computing $k$nn graphs, but the focus again is on $k$nn graphs based on Euclidean distance metrics. In [20], an algorithm is presented for the computation of the $k$nn graph for a 2-dimensional Euclidean point set for coarse grained multicomputers that runs in time $\Theta(n \log n/p + t(n,p))$, where n is the number of points in the data set, p is the number of processors, and $t(n,p)$ is the time for a global-sort operation. The work in [12] develops an optimal algorithm for the computation of the $k$nn graph of a point set in d-dimensional Euclidean space based on the well-separated pair decomposition of a point set that requires $O(\log n)$ time with $O(n)$ processors on a standard CREW PRAM model.

Recent progress in the development of parallel and distributed databases and data mining has spawned a large amount of research in the development of efficient distributed algorithms for the computation of $k$nn queries [44,57]. In [1], the PN-tree is developed as an efficient data structure for parallel and distributed multidimensional indexing and can be used for the computation of $k$nn queries for certain distance metrics. The computation of $k$nn queries using the PN-tree data structure depends on clustering of points into nodes and projections of nodes over each dimension, which is not generally possible for arbitrary distance metrics. In [7], a parallel algorithm is developed for computing multiple $k$nn queries based on arbitrary distance metrics from a database. Experimental results on Euclidean data sets and up to 16 servers show close to linear speedup. We note that the focus of parallel and distributed databases and data mining is on applications that are usually I/O intensive and require the computation of one or several $k$nn queries. The focus of the work in this paper is different. This paper targets applications that are computationally intensive, require the computation of the $k$nn graph, and the computation of the $k$nn graph constitutes only one stage of the entire computation. Our motivation comes from our research in robot motion planning and biology [14,22,30,37,40,45,46], where the $k$nn graph is computed to assist in the computation of paths for robots or exploration of high-dimensional state spaces to analyze important properties of biological processes.

3 Distributed Computation of the $k$nn Graph

Our distributed algorithm for the computation of the $k$nn graph utilizes sequential $k$nn data structures, such as those surveyed in section 2, for the partial computation of $k$nn queries and efficient communication and computation schemes for computing the k nearest neighbors of each point in the data set. The description of the distributed algorithm is general and applicable to any $k$nn data structure. We initially present the distributed algorithm by considering the $k$nn data structure as a black-box and later discuss how to take advantage of the specific implementations of the $k$nn data structures to improve the efficiency of the distributed algorithm for the computation of the $k$nn graph. At the end of this section, we also discuss possible extensions to the distributed framework including computation of graphs based on approximate $k$nn and range queries.

3.1 Problem Definition

Let the pair $(\mathcal{S}, \rho)$ define a metric space [11], where $\mathcal{S}$ is a set of points and $\rho : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}^{\geq 0}$ is a distance metric that for any $x, y, z \in \mathcal{S}$ satisfies the following properties: (i) $\rho(x, y) \geq 0$ and $\rho(x, y) = 0$ iff $x = y$ (positivity); (ii) $\rho(x, y) = \rho(y, x)$ (symmetry); and (iii) $\rho(x, y) + \rho(y, z) \geq \rho(x, z)$ (triangle inequality). Let $S = \{s_1, s_2, \ldots, s_n\} \subset \mathcal{S}$ be a collection of n points defining the data set. Let $k \in \mathbb{N}$ be an integer representing the number of nearest neighbors. The k nearest neighbors ($k$nn), denoted by $N_S(s_i, k)$, of a point $s_i \in S$ are defined as the k closest points to s_i from $S - \{s_i\}$ according to the metric ρ. The $k$nn graph of S, $G(S, k) = (V, E)$, is an undirected graph where $V = \{v_1, \ldots, v_n\}$, with v_i corresponding to s_i, and $E = \{(v_i, v_j) \in V^2 : s_i \in N_S(s_j, k) \vee s_j \in N_S(s_i, k)\}$.

3.2 Model of Computation

Let $P = \{P_1, P_2, \ldots, P_p\}$ be the set of all the available processors for the computation of $G(S, k)$. Let $S_1, S_2, \ldots, S_p$ be a partition of S, i.e., $S = S_1 \cup S_2 \cup \cdots \cup S_p$ and $S_i \cap S_j = \emptyset$ for all $i, j \in \{1, 2, \ldots, p\}$ and $i \neq j$. Let T_{S_i} be a data structure that computes $k$nn queries for the set S_i, i.e., querying T_{S_i} with $s \in S$ and $k \in \mathbb{N}$, denoted $T_{S_i}(s, k)$, produces $N_{S_i}(s, k)$, where $N_{S_i}(s, k)$ denotes the k closest points to s from $S_i - \{s\}$.

In our model of computation, processors communicate via message passing and there is no-shared memory available. We assume restrictions on the memory available to each processor $P_i \in P$, similar to [20]. Processor P_i is capable of storing the set S_i, the data structure T_{S_i}, a small number of query points $C_i \subset S - S_i$, $k$nn query results for $|S_i| + |C_i|$ points, and a small amount of bookkeeping information. In addition, processor P_i is equipped with a small communication buffer to exchange messages, query points, and results with other processors.

Our model of computation is particularly well-suited for many scientific applications in different research fields such as robot motion planning, biological applications, etc., which often generate hundreds of gigabytes of data and require the computation of the $k$nn graph $G(S, k)$ for such extensively large data sets S. In addition to storage requirements, such applications are also computationally intensive and the computation of the $k$nn graph $G(S, k)$ constitutes only one stage of the overall computation. Therefore, effective distribution schemes, as the one we propose in this work, should use most of the main memory of each processor to store as much of the data set S as possible.

3.3 Local Nearest Neighbor Queries

Each processor $P_i \in P$ constructs a $k$nn data structure T_{S_i} for the computation of $k$nn queries $N_{S_i}(s, k)$ for any point $s \in S$. The objective of the $k$nn data structure T_{S_i} is the efficient compu-

tation of $k$nn queries $N_{S_i}(s,k)$ based on arbitrary distance metrics for any point $s \in S$. Several examples of efficient $k$nn data structures can be found in [4, 8, 15, 23, 28, 36, 55]. The distributed algorithm considers the $k$nn data structure T_{S_i} as a black-box it can query to obtain $N_{S_i}(s,k)$ for any point $s \in S$. In this way, the distributed algorithm is capable of computing the $k$nn graph utilizing any $k$nn data structure T_{S_i}. In addition, the distributed algorithm is also capable of taking advantage of specific implementations of the $k$nn data structures, as we discuss in section 3.5.7. It is important to note that processor P_i cannot query the $k$nn data structure T_{S_i} to obtain $N_S(s,k)$, since only the portion S_i of the data set S is available to processor P_i. In our distributed algorithm we show how to compute $N_S(s,k)$ for all $s \in S$ efficiently.

3.4 Data and Control Flow

Before relating the details of the distributed algorithm, we discuss data and control flow dependencies. The computation of the $k$nn query $N_S(s_i,k)$ for a point $s_i \in S_i$ requires the combination of results obtained from querying the $k$nn data structures T_{S_1}, ..., T_{S_p} associated with the processors P_1, ..., P_p, respectively, since S is partitioned into S_1, ..., S_p. Under our model of computation, each processor P_i is capable of storing only one $k$nn data structure, i.e., T_{S_i}, due to the limited amount of memory available to processor P_i. Furthermore, each processor P_i is equipped with only a small communication buffer which makes the communication of the data structure T_{S_i} from P_i to other processors prohibitively computationally expensive. Therefore, the only viable alternative for computing the $k$nn query $N_S(s_i,k)$ is for processor P_i to send the point s_i to other processors which in turn query their associated $k$nn data structures and send back the results to P_i.

An efficient distribution of the computation of the $k$nn graph $G(S,k)$ can be achieved by maximizing useful computation. For each processor $P_i \in P$, the useful computation consists of (i) the computation of $k$nn for points owned by processor P_i and (ii) the computation of $k$nn for points owned by other processors. In order to maximize the useful computation by reducing idle times, it is important that each processor P_i handles requests from other processors as quickly as possible [25, 58].

The computation of $k$nn queries for points owned by processor P_i requires no communication, while the computation of $k$nn queries for points owned by other processors requires communication. Each processor P_i has a limited cache C_i, thus only a small number of points from other processors can be stored in C_i. Following general guidelines set forth in [25, 58] for efficient distribution algorithms, in order to accommodate as many points from other processors as possible, it is important that processor P_i empties its cache C_i quickly. Hence, before computing any $k$nn queries for points it owns, processor P_i first computes any $k$nn queries pending in the cache C_i. Furthermore, in order to minimize the idle or waiting time of other processors, processor P_i also handles any communication requests made by other processors before computing any $k$nn queries for the points it owns. In this way, processor P_i gives higher priority to requests from other processors and computes $k$nn queries for points it owns only when there are no pending requests from other processors. The overall effect of this schema is shorter idle and waiting times for each processor which translates into better utilization of resources for useful computations.

We have designed a decentralized architecture for our distributed implementation of the computation of the $k$nn graph $G(S,k)$. We utilize only asynchronous communication between different processors in order to minimize idle and waiting times. Each processor P_i is responsible for the computations of $k$nn queries utilizing the data structure T_{S_i}, posting of requests to other processors for the computation of $k$nn queries for points it owns, and ensuring a timely response to requests made by other processors.

DKNNG Algorithm: Distributed Computation of the $k$nn Graph

Input: $S_i \subset S = \{s_1, s_2, \ldots, s_n\}$, points
k, number of nearest neighbors

Output: $N_S(s_i, k)$ for each $s_i \in S_i$

Computation by processor P_i. All communications are done asynchronously.

```
 1: initialize cache C_i ← ∅
 2: construct knn data structure T_{S_i}
 3: while computing G(S,k) is not over do
 4:   if cache C_i is not empty then
 5:     select and remove one point s from C_i
 6:     compute query T_{S_i}(s,k)
 7:     send results to owner of s
 8:   if cache C_i is not full then
 9:     post request to fill C_i
10:   if received "cache is not full" then
11:     P' ← processors posting the request
12:     select points from S_i to send to P'
13:     send the selected points to P'
14:     request results from P'
15:   if received points then
16:     if cache C_i is full then
17:       create room in C_i
18:     add received points to C_i
19:   if received results then
20:     update results
21:   if no pending requests then
22:     select one point s from S_i
23:     compute query T_{S_i}(s,k)
24:     update results
25: end while
```

Algorithm 1. High-level description of the distributed computation of the $k$nn graph. The pseudocode is applicable to each processor $P_i \in P$. The computation of the $k$nn graph proceeds in stages and at the end of the computation each processor P_i stores the $k$nn results $N_S(s_i, k)$ for each point $s_i \in S_i$.

3.5 Distributed Computation

The distributed $k$nn algorithm DKNNG proceeds through several stages, as illustrated in Algorithm 1. After initializing the cache C_i to be empty and constructing the $k$nn data structure T_{S_i} associated with the points in S_i (lines 1–2), each processor P_i enters a loop until the computation of the $k$nn graph $G(S, k)$ is complete (lines 3–24). Inside the loop, each processor P_i tests if there are pending computations in the cache C_i (lines 4–7), if the cache C_i needs to be filled (lines 8–9), if there are pending requests from other processors (lines 10–20), or if it should compute $k$nn queries from the points S_i it owns (lines 21–24). As the computation proceeds, certain issues arise including how processor P_i processes queries pending in the cache, removes and adds queries to the cache, communicates with other processors to fill its cache and the cache of other processors with query points. Processor P_i considers the different stages in an order that attempts to minimize the idle and waiting times of other processors by handling pending requests from other processors [25, 58] as quickly as possible before it computes $k$nn queries for the points it owns. We now describe each stage of DKNNG in more detail.

3.5.1 Cache is Not Empty

Processor P_i selects one point s from the cache C_i and computes its nearest neighbors utilizing the $k$nn data structure T_{S_i}. The selection of the point s can be done in a variety of ways. One possibility is to apply the first-in first-out principle and select from the cache C_i the point s that has been in the cache C_i for the longest time. Processing the points in the order of addition to the cache has the benefit of guaranteeing that each point will eventually be processed and thus avoids the starvation problem. On the other hand, since other processors will possibly send to processor P_i several points at a time, processing of points in the cache C_i from processor P'' will start only after all the points in the cache C_i from processor P' have been processed (assuming

that points from processor P' were added to the cache C_i before the points from processor P''.)

In order to ensure a timely response to the requests from other processors, we introduce randomization to the selection process. A weight w_s is associated with each point s in the cache C_i. The weight w_s is directly proportional to the time the point s has been in the cache C_i and inversely proportional to the number of points in the cache C_i owned by the processor that owns the point s. The constant of proportionality can be defined to also include a measure of the state of computation for each processor and the importance of the computation of the nearest neighbors for the point s to the processor that owns the point s. A probability p_s is computed as $p_s = w_s / \sum_{s' \in C_i} w_{s'}$ and a point s is selected with probability p_s from the cache C_i. The weight w_s of a point s increases the longer s waits in the cache C_i, since w_s is directly proportional to the waiting time of s. Therefore, as s waits in the cache, the probability p_s increases and thus the likelihood that s will be the next point selected from C_i increases as well. In order to guarantee that there is no starvation and to ensure that points in the cache do not wait for long periods of time, we use a first-in first-out strategy to select points that have been in the cache for longer than a predefined maximum amount of time.

3.5.2 Cache is Not Full

The objective of processor P_i is to maximize the useful computation [25, 58]. Consequently, it is important that there are always some points in the cache C_i, so that processor P_i can spend useful computation time computing $k$nn queries on behalf of other processors. In this way, processor P_i postpones the computation of $k$nn queries for its points as much as possible – these computations require no communication and can be done anytime.

Processor P_i requests from other processors to send to it several points in order to fill its cache C_i. In order to avoid communicating the same request over and over again, processor P_i sends the request only to those processors that have responded to a previous similar request. Processor P_i maintains a flag f_j for each processor $P_j \in P - \{P_i\}$ that indicates if processor P_j has responded to a "cache is not full" request from processor P_i. Processor P_i sends the request to processor P_j only if the flag f_j is set to `true`. After posting the request to processor P_j, processor P_i sets the flag f_j to `false`. The flag f_j is set again to `true` when processor P_i receives a response from processor P_j. Initially, processor P_i assumes that every other processor P_j has responded to its request to fill the cache C_i.

When processor P_j receives a request from P_i to fill the cache C_i, processor P_j selects one or more points uniformly at random from S_j and sends the selected points to processor P_i. The selected points are never again considered by processor P_j for filling the cache C_i of processor P_i. In order to ensure that the same point is never sent to a processor more than once, processor P_j associates with each point $s_j \in S_j$ and each processor $P_i \in P - \{P_j\}$ a single bit b_{s_j,P_i} that indicates if point s_j has been sent to processor P_i. A point s_j is considered for selection iff b_{s_j,P_i} is set to `false`. The bit b_{s_j,P_i} is set to `true` when the point s_j is sent to processor P_i.

The bookkeeping information requires the additional storage of $(|P| - 1) + (|P| - 1)|S_j|$ bits by each processor $P_j \in P$. This additional amount of bookkeeping is very small compared to the size of the data S_j. As an example, if $|P| = 100$, S_j is `2GB` large, each point $s_j \in S_j$ has 200 dimensions, and each dimension is represented by a 64-bit `double`, the amount of bookkeeping is 15.84`MB` or equivalently 0.77% of the size of S_j.

3.5.3 Received Points

Processor P_i receives points from other processors only as a response to the request of P_i to fill its cache C_i. If there is room in the cache C_i, the received points are added to the cache C_i.

Otherwise, processor P_i selects one point from the cache C_i, using the selection process discussed in section 3.5.1, and computes the $k$nn query associated with it. This process continues until all the received points are added to the cache C_i.

3.5.4 Received Results

Processor P_i associates with each point $s_i \in S_i$ the combined results of the $k$nn queries computed by P_i and other processors. Let $N_S(s_i, k)$ be the current $k$nn results associated with the point s_i. Initially, $N_S(s_i, k)$ is empty. Let $N_{S_j}(s_i, k)$ be the $k$nn results computed by processor P_j utilizing the $k$nn data structure T_{S_j}. The set $N_S(s_i, k)$ is updated by considering all the points $s \in N_{S_j}(s_i, k)$ and adding to $N_S(s_i, k)$ the point s iff $|N_S(s_i, k)| < k$ or $\rho(s, s_i) < \min_{s' \in N_S(s_i,k)} \rho(s', s_i)$. In other words, processor P_i merges the $k$nn results computed by processor P_j with the current results. In the merging process, only the k closest neighbors are associated with the result $N_S(s_i, k)$.

3.5.5 No Pending Requests

If there are no pending requests, processor P_i has computed all the $k$nn queries associated with the points in the cache C_i and has requested from other processors to fill its cache C_i. Furthermore, processor P_i has handled all the requests from other processors and, thus, it is free to compute $k$nn queries from the points S_i it owns. Therefore, processor P_i selects one of the points $s_i \in S_i$, which has not been selected before, and computes the $k$nn query $N_{S_i}(s_i, k)$ utilizing the data structure T_{S_i}. The computed results $N_{S_i}(s_i, k)$ are merged with the current results $N_S(s_i, k)$, as described in section 3.5.4.

Processor P_i uses the computations of the $k$nn queries for points it owns to avoid possible idle or waiting times when it has handled all the requests from other processors. Doing such computations only when it is necessary increases the effective utilization of processors [25, 58] and consequently results in an efficient distribution of the computation of the $k$nn graph $G(S, k)$.

3.5.6 Termination Criteria

The computation of the $k$nn graph $G(S, k)$ is complete when $N_S(s, k)$ has been computed for each $s \in S$. Since S is partitioned into $S_1, \ldots, S_p$, it suffices if for each S_i and each $s_i \in S_i$, the computation of $N_S(s_i, k)$ is complete. Observe that the computation of $N_S(s_i, k)$ is complete iff processor P_i, which owns the point s_i, has combined the $k$nn query results $N_{S_1}(s_i, k), \ldots, N_{S_p}(s_i, k)$, i.e., each processor P_j has computed the $k$nn queries for the point s_i utilizing the $k$nn data structure T_{S_j} and processor P_i has gathered and combined all the computed results into $N_S(s_i, k)$.

When the computation of $N_S(s_i, k)$ is complete for all $s_i \in S_i$, processor P_i notifies all the other processors that the computation of $k$nn queries for S_i is complete. Processor P_i meets the termination criteria when the computation of $k$nn queries for S_i is complete and when processor P_i receives from all the other processors $P_j \in P - P_i$ the notification that the computation of $k$nn queries for S_j is complete.

Processor P_i detects the completion of the computation of $N_S(s_i, k)$ for all $s_i \in S_i$ by maintaining a counter. The counter is initially set to 0 and incremented each time processor P_i computes a query for a point it owns or when processor P_i receives results from another processor. When the counter reaches the value $|P||S_i|$, the computation of $N_S(s_i, k)$ is complete for each point $s_i \in S_i$, since each processor has computed $k$nn queries using each point $s_i \in S_i$.

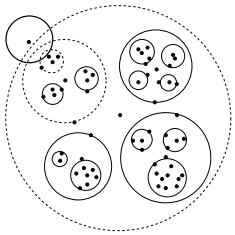


Fig. 2. Querying the GNAT [8] data structure. The empty circle drawn with a thick black perimeter represents the hypersphere $B(s_i)$ centered at s_i. During the computation of the query $T_{S_i}(s_i, k)$, there is no need to consider points inside the circles with the solid lines, since these circles do not intersect $B(s_i)$.

3.5.7 Exploiting Properties of the knn Data Structure

Each processor P_i utilizes the $k$nn data structure T_{S_i} for the computation of $k$nn queries. As presented in Algorithm 1 (line 6), DKNNG considers each $k$nn data structure T_{S_i} as a black-box. However, DKNNG not only works with any $k$nn data structure, but it can easily take advantage of special properties of the $k$nn data structure T_{S_i} to improve the overall performance.

The general idea consists of utilizing information gathered during the computation of $k$nn queries by processor P_i or other processors to prune future computations of $k$nn queries. Past computations could provide information that can be used to eliminate from consideration points in a data set that are too far from a query point to be its nearest neighbors. Such an idea has been successfully used by many efficient nearest neighbors algorithms [4,8,15,23,28,36,55]. The overall effect is that the computation of $k$nn queries by other processors for some of the points owned by processor P_i may become unnecessary and thus the efficiency of DKNNG is improved since certain communication and computation costs are eliminated.

Pruning local searches: For the clarity of exposition, we illustrate how to take advantage of the $k$nn data structures by considering GNAT [8] and kd-trees [23] as the data structure T_{S_i}. The description, however, is applicable to many $k$nn data structures [4, 15, 28, 36, 55].

GNAT is a hierarchical data structure based on metric trees [53] that supports the efficient computation of nearest neighbor searches. At the root level, the set S_i is partitioned into smaller sets using an approximate k-centers algorithm and each center is associated with a branch extending from the root. The construction process continues recursively until the cardinality of the set in the partition is smaller than some predefined constant. An illustration is provided in Figure 1.

GNAT is a suitable choice since it is efficient for large data sets and supports the computation of $k$nn queries for arbitrary metric spaces, which is important in many research areas such as robot motion planning, biological applications, etc. An important property of GNAT is that the efficiency for the computation of $k$nn for a point $s_i \in S$ can be improved by limiting the search only to points that are inside a small hypersphere $B(s_i)$ centered at s_i, as illustrated in Figure 2. We compute the radius $r_{B(s_i)}$ of $B(s_i)$ as the largest distance from s_i to a point in $N_S(s_i, k)$. When $N_S(s_i, k)$ is empty, $r_{B(s_i)}$ is set to ∞. As described in section 3.5.5, $N_S(s_i, k)$ contains the partial results of the $k$nn query for the point s_i. The computation of the $k$nn query $N_{S_j}(s_i, k)$ by processor P_j can significantly be improved by sending to it, in addition to s_i, the radius $r_{B(s_i)}$, since a point $s_j \in S_j$ cannot be one of the k closest neighbors to s_i if it is outside $B(s_i)$, since there are already at least k points inside $B(s_i)$.

Other $k$nn data structures, such as kd-trees [23] could be exploited in a similar way. A kd-tree constructs a hierarchical representation of the data set based on axis-aligned hyperplane

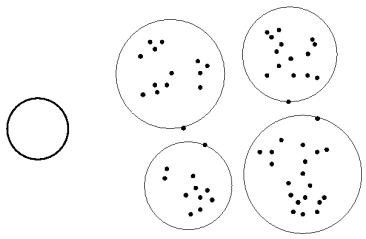


Fig. 3. Additional pruning of $k$nn searches. The hypersphere $B(s_i)$ is represented by the empty circle with the thick black perimeter, while the centers $B(c_j)$ are represented by the circles with the points inside. Processor P_i can avoid sending the point s_i to processor P_j, since $B(s_i)$ does not intersect any of the hyperspheres $B(c_j)$. Any point $s_j \in S_j$ is at least a distance $r_{B(s_i)}$ away from s_i and thus cannot be in $N_S(s_i, k)$, since $N_S(s_i, k)$ already contains k points that are inside $B(s_i)$.

decompositions. At each level, an axis and a hyperplane through that axis is chosen and the data set is split into two sets depending on the side of the hyperplane each data point is located. The hyperplane is typically chosen to be the median of the points with respect to the coordinates of the chosen axis. The process is repeated recursively until the cardinality of the remaining data set is smaller than some predefined constant. The efficiency of the $k$nn search can be improved by pruning subtrees whose associated hyperplane does not intersect with $B(s_i)$, since points in these subtrees cannot be closer to s_i than points already in $B(s_i)$. Therefore, the radius $r_{B(s_i)}$ can be used by processor P_j to prune certain branches during the computation of $N_{S_j}(s_i, k)$.

Reducing communication and computation costs: The idea of pruning the $k$nn search by limiting it to points inside a small hypersphere around the query point can be further utilized in other parts of `DKNNG` to improve its efficiency. During initialization, each processor P_i selects several points $C_i \subset S_i$ as center points and associates each point $s_i \in S_i$ with the closest center $c_i \in C_i$. The computation of the centers can be done in a variety of ways including algorithms similar to k-means [6] or approximate k-centers [8]. The radius of the hypersphere $B(c_i)$ is computed as the largest distance from $c_i \in C_i$ to a point associated with c_i. Processor P_i then sends c_i and $r_{B(c_i)}$ for each $c_i \in C_i$ to all other processors. Consider now the computation of $N_{S_j}(s_i, k)$ by processor P_j for some point $s_i \in S_i$. A point $s_j \in N_{S_j}(s_i, k)$ will be merged with $N_S(s_i, k)$ iff $\rho(s_j, s_i) < r_{B(s_i)}$. Since $s_j \in S_j$ is contained inside $B(c_j)$ for some $c_j \in C_j$, then we know s_j cannot be merged with $N_S(s_i, k)$ if $B(s_i) \cap B(c_j) = \emptyset$. Hence, processor P_i can avoid sending to processor P_j any point $s_i \in S_i$ such that $B(s_i) \cap B(c_j) = \emptyset$ for all $c_j \in C_j$, since none of the points in $N_{S_j}(s_i, k)$ will be merged with $N_S(s_i, k)$. Such pruning further reduces the communication and computation costs of `DKNNG`. The idea is illustrated in Figure 3.

3.5.8 Improving the Distribution by Preprocessing the Data Set

The efficiency of `DKNNG` can be further improved by partitioning the set S into sets $S_1, \ldots, S_p$, such that the partitioning improves the pruning of the $k$nn search as discussed in section 3.5.7. One possibility is to partition S into p clusters of roughly the same size and assign each cluster to one processor. The clusters can be again computed using algorithms similar to k-means [6] or approximate k-centers [8]. The partition of the set S into clusters increases the likelihood that $B(c_i) \cap B(c_j) = \emptyset$ for $c_i \in C_i$ and $c_j \in C_j$, since points in S_i belong to a different cluster than points in S_j. Consequently, the likelihood that, for $s_i \in S_i$, $B(s_i) \cap B(c_j) = \emptyset$ for all $c_j \in C_j$, also increases. In such cases, the computation of the $k$nn query $N_{S_j}(s_i, k)$ becomes unnecessary and thus reduces overall computational and communication costs.

3.6 Extensions to DKNNG

Although DKNNG is presented for the computation of the $k$nn graph, the proposed framework can be extended and generalized in several ways. We now discuss how to apply and extend our distributed framework to compute graphs based on approximate $k$nn queries and range queries.

3.6.1 Computation of Graphs Based on Approximate knn Queries

The computation of graphs based on approximate $k$nn queries is a viable alternative to the computation of graphs based on $k$nn queries for many applications in robot motion planning, biology, and other fields, especially for large high-dimensional data sets. In an approximate $k$nn query, instead of computing exactly the k closest points to a query point, it suffices to compute k points that are within a $(1+\epsilon)$ hypersphere from the k-th closest point to the query point. As the number of points and the dimension of each point in the data set increases, the computation of $k$nn queries becomes computationally expensive and can be made more efficient by considering approximate $k$nn queries. Approximate $k$nn queries provide a trade-off between efficiency of computation and quality of neighbors computed for each point.

The DKNNG algorithm of Algorithm 1 easily supports the computation of graphs based on approximate $k$nn queries. Since each data structure T_{S_i} is considered as a black-box, it suffices to use data structures T_{S_i} that support the computation of approximate $k$nn queries instead of $k$nn queries. Furthermore, efficient approximate nearest-neighbors algorithm exploit past computations to prune future searches in a similar fashion as exact nearest-neighbors algorithms [4,16,36]. Therefore, all the exploitations of the $k$nn data structures to improve the efficiency of DKNNG, as discussed in section 3.5.7, are applicable to approximate $k$nn data structures as well.

3.6.2 Computation of Graphs Based on Range Queries

Another extension is the computation of range queries instead of $k$nn queries. In a range query, we are interested in computing all the points $s \in S$ that are within some predefined distance ϵ from a query point $s_i \in S$, i.e., $R_S(s_i, \epsilon) = \{s \in S : \rho(s, s_i) \leq \epsilon\}$. Thus, we would like to compute $R_S(s_i, \epsilon)$ for all the points $s_i \in S$. This is easily achieved by using data structures that support range queries instead of data structures that support $k$nn queries. As in the case of the $k$nn search, a large amount of work has been devoted to the development of efficient algorithms for the computation of range queries (see [19] for extensive references). One potential problem with range queries that could affect the performance of DKNNG is that the communication of the range results $R_{S_j}(s_i, \epsilon)$ from processor P_j to processor P_i could be less efficient than the communication of $N_{S_j}(s_i, k)$, since it is possible that $|R_{S_j}(s_i, \epsilon)| > k$. On the other hand, the update of range results is more efficient than the update of $k$nn results, since processor P_i updates range results by simply appending $R_{S_j}(s_i, k)$ to $R_S(s_i, k)$. Furthermore, in the cases where the application we are interested in do not require processor P_i to store the results of $R_S(s_i, \epsilon)$, i. e., the results $R_S(s_i, \epsilon)$ can be stored in any of the available processors, then the communication of the range results $R_{S_j}(s_i, k)$ from processor P_j to processor P_i is unnecessary. Unlike in the case of the computation of $k$nn queries where results computed from other processors can be used to reduce the radius of the hypersphere centered at the query point, in the case of range queries, the radius of the hypersphere centered at the query point remains fixed to ϵ throughout the distributed computation. Therefore, processor P_j can avoid communicating the range results $R_{S_j}(s_i, \epsilon)$ to processor P_i and keep the results stored in its memory. In cases where storing $R_{S_j}(s_i, \epsilon)$ exceeds the memory capacity available to processor P_j, then processor P_j writes the range results $R_{S_j}(s_i, \epsilon)$ to a file.

4 Experiments and Results

Our experiments were designed to evaluate the performance of DKNNG compared to the sequential implementation.

4.1 Hardware and Software Setup

The implementation of DKNNG was carried out in ANSI C/C++ using the MPICH implementation of MPI standard for communication. Code development and initial experiments were carried out in Rice Terascale Cluster, PBC Cluster, and Rice Cray XD1 Cluster ADA. The experiments reported in this paper were run on the Rice Terascale Cluster, a 1 TeraFLOP Linux cluster based on Intel® Itanium®2 processors. Each node has two 64-bit processors running at 900MHz with 32KB/256KB/1.5MB of L1/L2/L3 cache, and 2GB of RAM per processor. The nodes are connected by a Gigabit Ethernet network. For the DKNNG experiments, we used two processors per node.

4.2 Data Sets

In this section, we describe the data sets we used to test the performance of DKNNG, how these data sets were generated, and the distance metrics we used in the computation of the $k$nn graph.

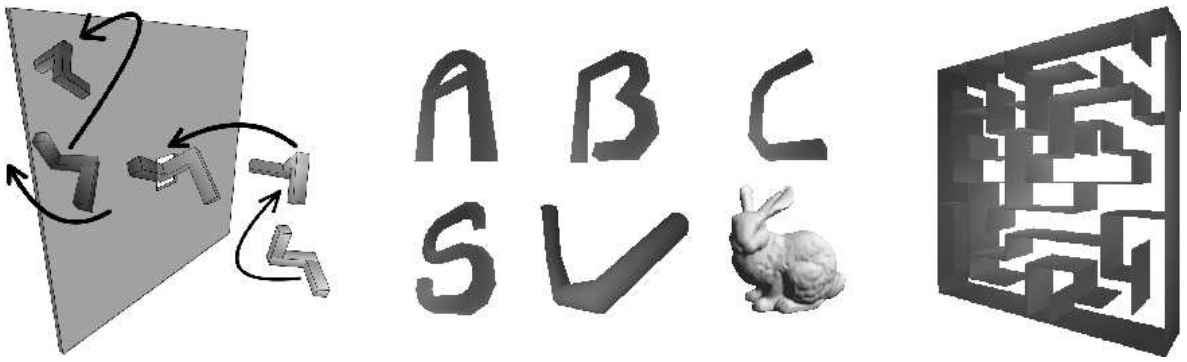
4.2.1 Number of Points and Dimensions

We tested the performance of DKNNG on data sets of varying number of points and dimension. We used data sets with 100000, 250000, and 500000 points and 90, 105, 174, 203, 258, and 301 dimensions. The 18 data sets obtained by combining all possible variations in the number of points and dimension varied in capacity from 83MB to 1.3GB. In addition, we also used data sets with 1000000, 2000000, and 3000000 points and 504 and 1001 dimensions. The additional 6 data sets obtained by combining all possible variations in the number of points and dimensions varied in capacity from 3.75GB to 22.37GB.

4.2.2 Generation of Data Sets

Each data set was obtained from motion planning benchmarks, since motion planning is our main area of research. The objective of motion planning is to compute collision-free paths consisting of rotations and translations for robots comprised of a collection of polyhedra moving amongst several polyhedral obstacles. An example of a path obtained by a motion planning algorithm is shown in Figure 4(a). In our benchmarks, the robots consisted of objects described by a large number of polygons, such as 3-dimensional renderings of the letters of the alphabet, bent cylinders, and the bunny, as illustrated in Figure 4(b), and the obstacles consisted of the walls of a 3-dimensional maze, as illustrated in Figure 4(c). Floor and ceiling are removed from Figure 4(c) to show the maze.

Efficient motion planning algorithms, such as [24, 31, 41, 45, 46], construct the $k$nn graph using points representing configurations of the robots. The configuration of a robot is a 7-dimensional point parameterized by a 3-dimensional position and an orientation represented by a 4-dimensional quaternion [2]. (We note that the robot has only 6 degrees of freedom, three for translation and three for rotation. The fourth parameter in the parameterization of the rotation is redundant, since unit quaternions suffice for the representation of rotations.) The configuration of ℓ robots is a 7ℓ-dimensional point obtained by concatenating the configurations of each robot. The distance between two configurations of a single robot is defined as the geodesic distance in $SE(3)$ [10]. For multiple robots, the distance between two configurations is defined by summing the $SE(3)$ distances between the corresponding configuration projections for each robot [14]. The computation of the geodesic distance is an expensive operation as it requires the evaluation of several



(a) (b) (c)

Fig. 4. A motion planning benchmark. The objective of the motion planning is to compute collision-free paths consisting of rotations and translations for robots comprised of a collection of polyhedra moving amongst several polyhedral obstacles in 3-dimensional environments. Rendering of objects is not on the same scale. (a) A path where the robot goes from one side of a wall to the other side of the wall by wiggling its way through a small hole. Illustrated are several consecutive poses of the robot as it follows the path. The arrows indicate the direction of motion from one pose to the other. (b) Several different robots varying from 3-dimensional renderings of the letters of the alphabet to a collection of bent cylinders and complex geometric models such as the bunny. (c) A 3-dimensional maze representing the environment where the robots are allowed to move.

points	100000, 250000, 500000						1000000, 2000000, 3000000	
dimension	90	105	174	203	258	301	504	1001
point type	`emb`	`cfg`	`emb`	`cfg`	`emb`	`cfg`	`cfg`	`cfg`
distance	`euc`	`geo`	`euc`	`geo`	`euc`	`geo`	`geo`	`geo`

(a) (b)

Table 1
Data sets used for the computation of the $k$nn graph. The geodesic (`geo`) and Euclidean (`euc`) distances are used for the computation of $k$nn graphs on configuration (`cfg`) and embedding (`emb`) points, respectively. (a) For each dimension, data sets were generated with 100000, 250000, and 500000 points each. (b) For each dimension, data sets were generated with 1000000, 200000, and 3000000 points each.

sin, cos, and square root functions. Alternatively, in order to improve on the efficiency of the computation, the $k$nn graph can be based on the Euclidean distance defined for embeddings of configurations [14]. The $k$nn graph based on embedding points provides a trade-off between efficiency of computation and quality of neighbors computed for each point. An embedding of a single robot configuration is a 6-dimensional point obtained by rotating and translating two 3-dimensional points selected from the geometric description of the robot as specified by the configuration and concatenating the transformed points.

Data sets generated and used by `PRM`: The data sets generated by the motion planning algorithm `PRM` [31] are described in Table 1(a). `PRM` constructs a $k$nn graph by sampling configurations of the robot uniformly at random and connecting each configuration to its k closest neighbors by a simple path. The algorithm guarantees that the sampled configurations and the connections

between configurations avoid collisions with the obstacles. In order for PRM to find paths for multiple robots in environments with many obstacles or narrow passages, millions of configurations are typically sampled. As described in Table 1(a), we used data sets consisting either of configuration points or embedding points. All data sets are generated by using PRM [31] to solve motion planning problems in the maze environment of Figure 4. The number of robots was set to 15, 29, and 43 to obtain configuration points of 105, 203, and 301 dimensions and embedding points of 90, 174, and 258 dimensions, respectively. The robots consisted of 3-dimensional renderings of the letters of the alphabet, where the i-th robot is the ($i \mod 26$)-th letter of the alphabet.

Large data sets: We generated large data sets consisting of 1000000, 2000000, and 3000000 points of 504 and 1001 dimensions by sampling configurations uniformly at random, as described in Table 1(b). We did not use these large data sets for planning because (i) the scope of this paper is the distributed computation of the $k$nn graph and not planning and (ii) time limitations, since planning involves collision checking of configurations and paths and for the data sets of Table 1(b) would require several weeks of computation.

4.3 Efficiency of DKNNG

To measure the efficiency of DKNNG, we ran the distributed code on various data sets, as described in section 4.2, using different numbers of processors.

4.3.1 Description of Experiments

Number of nearest neighbors: We tested the performance of the sequential and DKNNG algorithms for various values of $k \in \{15, 45, 150\}$. We found very little variation in the running times for the sequential algorithm and DKNNG, thus in all our experiments we only report results obtained for $k = 15$.

Measuring the computation time of the sequential algorithm: The computation time required by the sequential algorithm is estimated by randomly selecting 500 points from the data set, computing the associated $k$nn queries and calculating the average time required to compute one $k$nn query. The total time is then obtained by multiplying the average time to compute one $k$nn query by the number of points in the data set and adding to it the time it takes to construct the $k$nn data structure. Our experiments with data sets of 100000 and 250000 points and 90, 105, and 174 dimensions showed little variation (less than 60 seconds) between estimated sequential time and actual sequential time.

Each data set can be stored in a single machine: The sizes of the data sets described in Table 1(a) vary from 83MB to 1.3GB. These sizes were chosen to make a fair comparison between DKNNG and the sequential algorithm by ensuring that each data set fits in the main memory of a single machine (see section 4.1 for a description of the hardware we used.) The efficiency of DKNNG would be even higher if each data set does not fit in the main memory of a single machine, as it is often the case in emerging applications in robot motion planning [40, 45, 46] and biology [3, 33, 39, 42, 50, 52], since the performance of the sequential implementation would deteriorate due to frequent disk access.

For each of the 18 data sets of Table 1(a), we ran DKNNG on 64, 80, and 100 processors. Table 2 contains a summary of the results. For each experiment, we report the computation time required by the sequential version and the efficiency obtained with p processors. The efficiency is calculated as $t_1/(t_p \cdot p)$, where t_1 is the time required by a single processor to compute the $k$nn graph and t_p is the time required by DKNNG to compute the $k$nn graph when run on p processors.

Storing each data set requires several machines: We also tested the performance of DKNNG on the

	[emb, euc, d = 90, k = 15]			[cfg, geo, d = 105, k = 15]		
	100000	250000	500000	100000	250000	500000
1	108.52m	1076.35m	5396.78m	1875.72m	12016.02m	47301.52m
64	0.922	0.970	0.986	0.992	0.996	0.998
80	0.905	0.958	0.982	0.990	0.995	0.998
100	0.889	0.958	0.976	0.989	0.994	0.997

	[emb, euc, d = 174, k = 15]			[cfg, geo, d = 203, k = 15]		
	100000	250000	500000	100000	250000	500000
1	197.14m	1838.97m	8128.69m	3666.77m	22602.67m	92797.77m
64	0.913	0.969	0.986	0.992	0.996	0.998
80	0.876	0.958	0.981	0.990	0.996	0.998
100	0.851	0.944	0.974	0.988	0.995	0.998

	[emb, euc, d = 258, k = 15]			[cfg, geo, d = 301, k = 15]		
	100000	250000	500000	100000	250000	500000
1	264.37m	2014.33m	8610.76m	5416.69m	34381.19m	141771.51m
64	0.889	0.958	0.979	0.992	0.997	0.998
80	0.853	0.946	0.973	0.990	0.996	0.998
100	0.813	0.931	0.966	0.987	0.996	0.998

Table 2
Efficiency of DKNNG on the data sets of Table 1(a). The heading of each table indicates the type of the points, the distance metric, the dimension of each point, and the number k of nearest neighbors (corresponding to a column in Table 1(a)). For each table, experiments were run with 100000, 250000, and 500000 points and on 1, 64, 80, and 100 processors. The row of processor 1 indicates the running time in minutes of the sequential algorithm. For experiments with 64, 80, and 100 processors, we indicate the efficiency of DKNNG computed as $t_1/(t_p \cdot p)$, where t_p is the running time of DKNNG on p processors.

data set of Table 1(b). The purpose of these data sets, which vary in size from 3.75GB to 22.37GB, is to test the efficiency of DKNNG for large data sets, where several machines are required just to store the data. We ran DKNNG using 20, 100, 120, and 140 processors. We based the calculations of the efficiency on the running time for 20 processors, since each processor is capable of storing 1/20-th of the data set in its main memory. This again ensures a fair computation of the efficiency as the number of processors is increased, since the performance of DKNNG on 20 processors does not suffer from frequent disk access. The efficiency is calculated as $20t_{20}/(t_p \cdot p)$, where t_{20} is the running time of DKNNG on 20 processors, and t_p is the running time of DKNNG on p processors, for $p = 100, 120, 140$. The results are presented in Table 3.

	[cfg, geo, d = 504, k = 15]			[cfg, geo, d = 1001, k = 15]		
	1000000	2000000	3000000	1000000	2000000	3000000
eff[20, 100]	1.000	0.999	1.000	0.999	0.999	0.999
eff[20, 120]	1.000	0.999	1.000	0.999	0.999	0.999
eff[20, 140]	1.000	0.999	1.000	0.998	0.997	0.999

Table 3
Efficiency of DKNNG on the large data sets of Table 1(b). The heading of each table indicates the type of the points, the distance metric, the dimension of each point, and the number k of nearest neighbors (corresponding to a column in Table 1(b)). For each table, experiments were run with 1000000, 2000000, and 3000000 points and on 20, 100, 120, and 140 processors. We report the efficiency of DKNNG on 100, 120, and 140 processors in rows eff[20, 100], eff[20, 120], and eff[20, 140], respectively. The efficiency is calculated based on the running time of DKNNG on 20 processors, i.e., $20t_{20}/(t_p \cdot p)$, where t_{20} is the running time of DKNNG with 20 processors, and t_p is the running time of DKNNG on p processors, for $p = 100, 120, 140$.

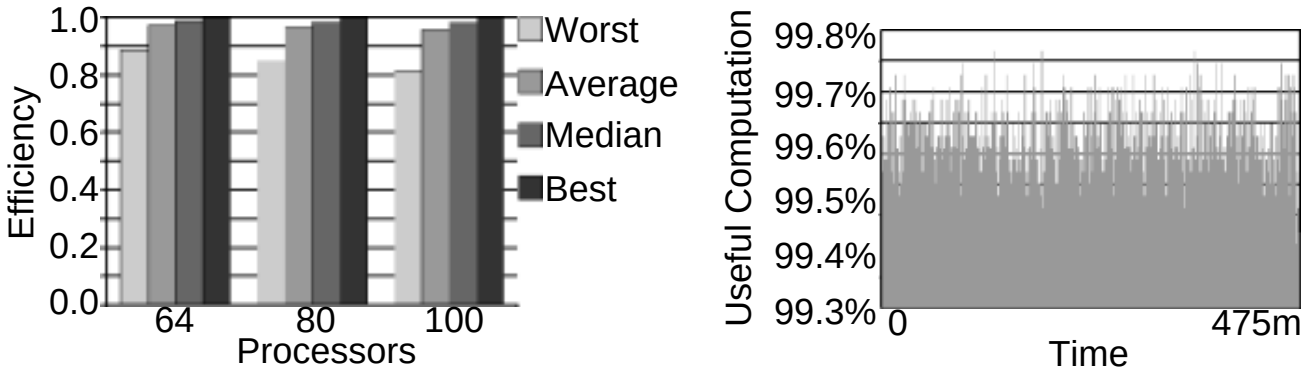
4.3.2 Results

The overall efficiency of DKNNG is reasonably high on all our benchmarks. From the results in Table 2, the efficiency on 64 processors ranges from 0.889 to 0.998 with an average of 0.974 and median of 0.989. When the number of processors is increased to 80, the efficiency ranges from 0.853 to 0.998 with an average of 0.966 and median of 0.986. The efficiency of DKNNG still remains high even on 100 processors, where the efficiency ranges from 0.813 to 0.998 with an average of 0.958 and median of 0.982.

For each dimension, we observe that the efficiency of DKNNG increases as the number of points in the data set increases. As an example, for $d = 90$ and $p = 100$, the efficiency of DKNNG is 0.889 for 100000 points, 0.958 for 250000 points, and 0.976 for 500000 points. One important factor that contributes to the increase in the efficiency is that each processor P_i has now more points S_i and thus more opportunities to fill in possible idle or waiting times by computing $k$nn queries for points $s_i \in S_i$ it owns.

Our experiments also indicate a high efficiency of DKNNG even when different metrics are used. Even though the computation of the Euclidean metric is much faster than the computation of the multiple geodesic metric in $SE(3)$, the efficiency of DKNNG remains high in both cases.

For a fixed number of points, we observe that the performance of DKNNG increases as the cost of computing the distance metric increases. For a fixed number of points and a fixed distance metric, we observe that the performance of DKNNG decreases as the dimension increases. The increase in the cost of computing the distance metric increases the useful computation time since more time is spent computing queries. The increase in the dimension increases communication costs since more data is communicated over the network when sending queries from one processor to another. Hence, depending on whether the increase in useful computations dominates over the increase in communication costs, the performance of DKNNG either increases or decreases, respectively. As an example, when using the Euclidean metric, the efficiency of DKNNG for $n = 250000$ and $p = 100$ is 0.958 for $d = 90$, 0.944 for $d = 174$, and 0.931 for $d = 258$. When using a computationally more expensive metric, such as the geodesic metric, the efficiency of DKNNG for $n = 250000$ and $p = 100$ is 0.994 for $d = 105$, 0.995 for $d = 203$, and 0.996 for $d = 301$.



(a) Efficiency (b) Useful computation

Fig. 5. Performance of the DKNNG algorithm. (a) Comparison of the ideal efficiency with the worst, average, median, and the best efficiency obtained for all the data sets of Table 1(a). (b) Characterization of the useful computation for $p = 100$ for the data set with $d = 105$ and $n = 500000$ by showing what percentage of the time is spent computing $k$nn queries.

Figure 5(a) compares the ideal efficiency to the worst, average, median, and the best efficiency obtained for all the data sets of Table 1(a) when DKNNG is run on 64, 80, and 100 processors. Figure 5(a) indicates a nearly ideal efficiency for the DKNNG algorithm.

Figure 5(b) presents logged data for the data set with $d = 105$ and $n = 500000$, when DKNNG is run on 100 processors, showing the percentage of time spent computing $k$nn queries. The plot in Figure 5(b) is characteristic of the behavior of DKNNG on the other data sets as well. Figure 5(b) indicates that most of the time, ranging from 99.35% to 99.77%, is spent doing useful computation, i.e., computation of $k$nn queries on behalf of other processors and computation of $k$nn queries for points owned by the processor, which results in a highly efficient distributed algorithm.

The results in Table 3 indicate that DKNNG achieves high efficiency for large data sets. As discussed earlier, increasing the number of points and the cost of computing the distance metric increases the useful computation time since more time is spent computing $k$nn queries, resulting in an almost ideal efficiency for DKNNG.

5 Discussion

Our work is motivated by increasingly high-dimensional problems arising from motion planning [14, 31, 35, 40, 45, 46], biological applications [3, 33, 39, 42, 50, 52], pattern recognition [21], data mining [18, 51], multimedia systems [13], geographic information systems [34, 43], and many other research fields, which often require efficient computations of $k$nn graphs based on arbitrary distance metrics. This paper presents an algorithm for efficiently distributing the computation of such graphs using a decentralized approach.

Our distributed framework is general and can be extended in many ways. Possible extensions include the computation of graphs based on other types of proximity queries, such as approximate $k$nn or range queries, as discussed in section 3.6. In addition, our distributed framework allows for the exploitation of certain properties of the data structures used for the computation of $k$nn queries, approximate $k$nn queries, or range queries, to improve the overall efficiency of the DKNNG algorithm.

The efficiency of DKNNG derives in part from a careful prioritization and handling of requests be-

tween processors and a systematic exploitation of properties of $k$nn data structures. Throughout the distributed computation, each processor uses computations that do not require communications to fill in idle times and gives a higher priority to computations requests by other processors in order to provide a timely response. Information collected during the computation of $k$nn queries by one processor is shared with other processors which use it to prune the computations of their $k$nn queries. The information sharing in some cases makes the computation of certain $k$nn queries completely unnecessary and thus reduces communication and computation costs. Our experimental results suggest that our distributed framework supports the efficient computation by hundreds of processors of very large $k$nn graphs consisting of millions of points with hundreds of dimensions and arbitrary distance metrics.

We intend to integrate `DKNNG` with our motion planning algorithms [45, 46] and use it for the solution of robot motion planning problems of unprecedented complexity. While recent motion planners can successfully solve robot motion planning problems with tens of dimensions, our objective is to solve robot motion planning problems consisting of thousands of dimensions. Other challenging applications stemming from biology, pattern recognition, data mining, fraud detection, geographic information systems that rely on the computation of $k$nn graphs could benefit similarly from the use of our distributed `DKNNG` framework.

Acknowledgements

The authors would like to thank Cristian Coarfa for helpful discussions and comments. Work on this paper has been supported in part by NSF 0205671, NSF 0308237, ATP 003604-0010-2003, NIH GM078988, and a Sloan Fellowship to LK. Experiments reported in this paper have been obtained on equipment supported by EIA 0216467, NSF CNS 0454333, and NSF CNS 0421109 in partnership with Rice University, AMD, and Cray.

References

[1] M. H. Ali, A. A. Saad, M. A. Ismail, The PN-tree: A parallel and distributed multidimensional index, Distributed and Parallel Databases 17 (2) (2005) 111–133.

[2] S. L. Altmann, Rotations, Quaternions, and Double Groups, Clarendon Press, Oxford, England, 1986.

[3] M. S. Apaydin, D. L. Brutlag, C. Guestrin, D. Hsu, J.-C. Latombe, Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion, Journal of Computational Biology 10 (3–4) (2003) 257–281.

[4] S. Arya, D. M. Mount, S. Nathan, An optimal algorithm for approximate nearest neighbor searching in fixed dimensions, Journal of the ACM 45 (6) (1998) 891–923.

[5] M. J. Atallah, S. Prabhakar, (Almost) Optimal parallel block access for range queries, Information Sciences 157 (1-2) (2003) 21–31.

[6] C. M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Oxford, England, 1995.

[7] B. Braunmüller, M. Ester, H.-P. Kriegel, J. Sander, Multiple similarity queries: a basic DBMS operation for mining in metric databases, IEEE Transactions on Knowledge and Data Engineering 13 (1) (2001) 79–95.

[8] S. Brin, Near neighbor search in large metric spaces, in: International Conference on Very Large Data Bases, San Francisco, California, 1995, pp. 574–584.

[9] F. Buccafurri, G. Lax, Fast range query estimation by n-level tree histograms, IEEE Transactions on Knowledge and Data Engineering 51 (2) (2004) 257–275.

[10] F. Bullo, R. M. Murray, Proportional derivative (PD) control on the Euclidean group, in: European Control Conference, Vol. 2, Rome, Italy, 1995, pp. 1091–1097.

[11] D. Burago, Y. Burago, S. Ivanov, I. D. Burago, A Course in Metric Geometry, American Mathematical Society, 2001.

[12] P. B. Callahan, Optimal parallel all-nearest-neighbors using the well-separated pair decomposition, in: IEEE Symposium on Foundations of Computer Science, 1993, pp. 332–340.

[13] M. Cheng, S. Cheung, T. Tse, Towards the application of classification techniques to test and identify faults in multimedia systems, in: IEEE International Conference on Quality Software, Los Alamitos, California, 2004, pp. 32–40.

[14] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, S. Thrun, Principles of Robot Motion: Theory, Algorithms, and Implementations, MIT Press, Cambridge, MA, 2005.

[15] B. Cui, B. C. Ooi, J. Su, K.-L. Tan, Contorting high-dimensional data for efficient main memory knn processing, in: ACM SIGMOD International Conference on Management of Data, San Diego, California, 2003, pp. 479–490.

[16] B. Cui, H. T. Shen, J. Shen, K.-L. Tan, Exploring bit-difference for approximate knn search in high-dimensional databases, in: Australasian Database Conference, Newcastle, Australia, 2005, pp. 165–174.

[17] P. Das, M. Moll, H. Stamati, L. E. Kavraki, C. Clementi, Low-dimensional free energy landscapes of protein folding reactions by nonlinear dimensionality reduction, Proceedings of the National Academy of Science USA 103 (26) (2006) 9885–9890.

[18] B. V. Dasarathy, Nearest-neighbor approaches, in: W. Klosgen, J. M. Zytkow, J. Zyt (Eds.), Handbook of Data Mining and Knowledge Discovery, Oxford University Press, New York, New York, 2002, pp. 88–298.

[19] M. de Berg, M. van Kreveld, M. H. Overmars, Computational Geometry: Algorithms and Applications, Springer, Berlin, 1997.

[20] F. Dehne, A. Rau-Chaplin, Scalable parallel computational geometry for coarse grained multicomputers, International Journal of Computational Geometry & Applications 6 (3) (1996) 379–400.

[21] R. O. Duda, P. E. Hart, D. G. Stork, Pattern Classification, John Wiley & Sons, New York, New York, 2000.

[22] P. W. Finn, L. E. Kavraki, Computational approaches to drug design, Algorithmica 25 (1999) 347–371.

[23] J. H. Freidman, J. L. Bentley, R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, ACM Transactions on Mathematical Software 3 (3) (1977) 209–226.

[24] R. Geraerts, M. Overmars, A comparitive study of probabilistic roadmap planners, in: J.-D. Boissonnat, J. Burdick, K. Goldberg, S. Hutchinson (Eds.), Algorithmic Foundations of Robotics V, Springer-Verlag, 2002, pp. 43–58.

[25] A. Grama, G. Karypis, V. Kumar, A. Gupta, An Introduction to Parallel Computing: Design and Analysis of Algorithms, Addison Wesley, Boston, MA, 2003.

[26] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, 1984, pp. 47–54.

[27] P. Indyk, Nearest neighbors in high-dimensional spaces, in: J. E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, CRC Press, 2004, pp. 877–892.

[28] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, R. Zhang, iDistance: An adaptive B+-tree based indexing method for nearest neighbor search, ACM Transactions on Database Systems 30 (2) (2005) 364–397.

[29] R. Jin, G. Yang, G. Agrawal, Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance, IEEE Transactions on Knowledge and Data Engineering 17 (1) (2005) 71–89.

[30] L. E. Kavraki, Geometry and the discovery of new ligands, in: J.-P. Laumond, M. H. Overmars (Eds.), Algorithms for Robotic Motion and Manipulation, A. K. Peters, 1997, pp. 435–448.

[31] L. E. Kavraki, P. Švestka, J.-C. Latombe, M. H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Transactions on Robotics and Automation 12 (4) (1996) 566–580.

[32] F. Korn, B.-U. Pagel, C. Faloutsos, On the 'dimensionality curse' and the 'self-similarity blessing', IEEE Transactions on Knowledge and Data Engineering 13 (1) (2001) 96–111.

[33] H.-P. Kriegel, M. Pfeifle, S. Schönauer, Similarity search in biological and engineering databases, IEEE Data Engineering Bulletin 27 (4) (2004) 37–44.

[34] W.-S. Ku, R. Zimmermann, H. Wang, C.-N. Wan, Adaptive nearest neighbor queries in travel time networks, in: ACM International Workshop on Geographic Information Systems, Bremen, Germany, 2005, pp. 210–219.

[35] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, H. Inoue, Motion planning for humanoid robots, in: D. Paolo, R. Chatila (Eds.), International Symposium of Robotics Research, Vol. 15 of Springer Tracts in Advanced Robotics, Springer Verlag, Siena, Italy, 2005, pp. 365–374.

[36] E. Kushilevitz, R. Ostrovsky, Y. Rabani, Efficient search for approximate nearest neighbor in high dimensional spaces, SIAM Journal of Computing 30 (2) (2000) 457–474.

[37] S. M. LaValle, P. W. Finn, L. E. Kavraki, J.-C. Latombe, Efficient database screening for rational drug design using pharmacophore-constrained conformational search, in: International Conference on Computational Molecular Biology, Lyon, France, 1999, pp. 250–260.

[38] T. Liu, A. W. Moore, A. Gray, K. Yang, An investigation of practical approximate nearest neighbor algorithms, in: L. K. Saul, Y. Weiss, L. Bottou (Eds.), Advances in Neural Information Processing Systems, MIT Press, Cambridge, MA, 2005, pp. 825–832.

[39] S. McGinnis, T. L. Madden, BLAST: at the core of a powerful and diverse set of sequence analysis tools, Nucleic Acids Research 32 (2004) W20–W25.

[40] M. Moll, L. E. Kavraki, Path planning for variable resolution minimal-energy curves of constant length, in: IEEE International Conference on Robotics and Automation, Barcelona, Spain, 2005, pp. 2143–2147.

[41] M. Morales, S. Rodriguez, N. Amato, Improving the connectivity of PRM roadmaps, in: IEEE International Conference on Robotics and Automation, Taipei, Taiwan, 2003, pp. 4427–4432.

[42] M. Paik, Y. Yang, Combining nearest neighbor classifiers versus cross-validation selection, Statistical Applications in Genetics and Molecular Biology 3 (1) (2004) article 12.

[43] A. Papadopoulos, Y. Manolopoulos, Parallel processing of nearest neighbor queries in declustered spatial data, in: ACM International Workshop on Advances in Geographic Information Systems, Rockville, Maryland, 1996, pp. 35–43.

[44] A. N. Papadopoulos, Y. Manolopoulos, Nearest Neighbor Search: A Database Perspective, Series in Computer Science, Springer Verlag, Berlin, Germany, 2005.

[45] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, L. E. Kavraki, Sampling-based roadmap of trees for parallel motion planning, IEEE Transactions on Robotics 21 (4) (2005) 597–608.

[46] E. Plaku, L. E. Kavraki, Distributed sampling-based roadmap of trees for large-scale motion planning, in: IEEE International Conference on Robotics and Automation, Barcelona, Spain, 2005, pp. 3879–3884.

[47] E. Plaku, L. E. Kavraki, Quantitative analysis of nearest-neighbors search in high-dimensional sampling-based motion planning, in: Workshop Algo Found Robot, New York, NY, 2006, in press.

[48] A. Qasem, K. Kennedy, J. Mellor-Crummey, Automatic tuning of whole applications using direct search and a performance-based transformation system, The Journal of Supercomputing 36 (2) 183–196.

[49] T. Schwarz, M. Iofcea, M. Grossmann, N. Hönle, D. Nicklas, B. Mitschang, On efficiently processing nearest neighbor queries in a loosely coupled set of data sources, in: ACM International Workshop on Geographic Information Systems, Washington, DC, 2004, pp. 184–193.

[50] B. K. Shoichet, D. L. Bodian, I. D. Kuntz, Molecular docking using shape descriptors, Journal of Computational Chemistry 13 (3) (1992) 380–397.

[51] P.-N. Tan, M. Steinbach, V. Kumar, Introduction to Data Mining, Addison-Wesley, Boston, Massachusetts, 2005.

[52] S. Thomas, G. Song, N. M. Amato, Protein folding by motion planning, Physical Biology 2 (2005) S148–S155.

[53] J. K. Uhlmann, Satisfying general proximity/similarity queries with metric trees, Information Processing Letters 40 (4) (1991) 175–179.

[54] R. Weber, K. Böhm, Trading quality for time with nearest-neighbor search, in: C. Zaniolo, P. Lockemann, M. Scholl, T. Grust (Eds.), Inter. Conf. on Advances in Database Technology, Vol. 1777, 2000, pp. 21–35.

[55] P. N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: ACM-SIAM Symposium on Discrete Algorithms, Austin, Texas, 1993, pp. 311–321.

[56] M. Yim, K. Roufas, D. Duff, Y. Zhang, C. Eldershaw, S. Homans, Modular reconfigurable robots in space applications, Autonomous Robots 14 (2–3) (2003) 225–237.

[57] M. J. Zaki, C.-T. Ho (Eds.), Large-Scale Parallel Data Mining, Vol. 1759 of Lecture Notes in Computer Science, Springer Verlag, New York, 2000.

[58] A. Y. Zomaya (Ed.), Parallel and Distributed Computing Handbook, McGraw-Hill Professional, New York, NY, 1995.