MPR — A Partitioning-Replication Framework for Multi-Processing kNN Search on Road Networks

Siqiang Luo
University of Hong Kong
sqluo@cs.hku.hk

Ben Kao
University of Hong Kong
kao@cs.hku.hk

Xiaowei Wu
University of Vienna
wxw0711@gmail.com

Reynold Cheng
University of Hong Kong
ckcheng@cs.hku.hk

***Abstract*—We study the problem of executing road-network k-nearest-neighbor (kNN) search on multi-core machines. State-of-the-art kNN algorithms on road networks often involve elaborate index structures and complex computational logic. Moreover, most kNN algorithms are inherently sequential. These make the traditional approach of parallel programming very costly, laborious, and ineffective when they are applied to kNN algorithms. We propose the MPR (Multi-layer Partitioning-Replication) mechanism that orchestrates CPU cores and schedules kNN query and index update processes to run on the cores. The MPR mechanism performs workload analysis to determine the best arrangement of the cores with the objective of optimizing quality-of-service (QoS) measures, such as system throughput and query response time. We demonstrate the effectiveness of MPR by applying it to a number of state-of-the-art kNN indexing methods running on a multi-core machine. Our experiments show that multi-processing using our MPR approach requires minimal programming effort. It also leads to significant improvements in query response time and system throughput compared with other baseline parallelization methods.**

***Index Terms*—kNN search; road network; adaptive approach;**

I. Introduction

With the advances of GPS systems and mobile devices, location-based services have found widespread applications. Many of these applications have turned into billion-dollar businesses, such as Uber (taxi-hailing service, valued at $68 billion in 2017), Mobike (bike-sharing service, valued at $2.7 billion in 2018), and *Pokémon GO* (location-based game, $1.8 billion revenue in 2 years since launch). For these systems, the k-nearest-neighbor (kNN) search on road networks is an indispensable function. For example, Uber finds cars in its fleet that are the closest to a pick-up request; A *Pokémon GO* game server searches the closest *Pokémon*s for a player; Mobike locates the nearest servicing bikes for a rider.

In these applications, a road network is typically modeled as a graph such that a node models a road junction and an edge models a road segment connecting two junctions. On the network is a set $\mathcal{M}$ of objects (e.g., taxis, *Pokémon*s). Given a query location q, a kNN search for q locates the k objects in $\mathcal{M}$ that are the closest to q in terms of network distance.

The kNN problem on road networks has been extensively studied. In particular, researchers have invented sophisticated index structures and algorithms for very efficient kNN query processing. Some of these algorithms, e.g., V-tree, provide sub-millisecond query processing time. Despite the advances, there are significant limitations of existing works when they are deployed in large-scale systems. In this paper we put forward three *adaptability requirements* that kNN algorithms should satisfy. We propose the MPR framework, which provides a simple transformation of a given kNN solution into one that satisfies the three requirements.

Adaptability Requirements. A large-scale location-based-service system typically has to serve large volumes of *queries* and *udpates*. For instance, Didi taxi-hailing service receives thousands of requests per second during peak hours [10]. As another example, in the US, there were around 20 million daily active *Pokémon GO* players in 2016. Players can activate a "nearby tracking" function in the game, which helps them locate the nearest *Pokémon*s. Besides queries, object updates contribute heavy loads to the systems. For example, each Didi vehicle reports its location to the system every 3 to 5 seconds, and there are about 4 million active drivers in China providing Didi services. To handle the heavy workloads under a variety of applications and environments, a kNN solution should satisfy three adaptability requirements, namely, *system adaptability*, *workload adaptability*, and *performance adaptability*.

System adaptability refers to how easy it is to implement a kNN solution (data structures and algorithms) on systems with different modern hardware configurations. As a location-based service expands and scales, its system needs to be upgraded to more powerful machines (e.g., one with many CPU cores). Existing kNN solutions, however, are mostly sequential algorithms, which are designed and evaluated under a single-threading environment. A good kNN solution should be able to exploit the full potential of high-performance hardware with minimum engineering efforts.

Workload adaptability refers to how well a kNN solution can be configured to handle a mixture of query and update workloads. Different applications (e.g., taxi-hailing vs. location-based game) under different environments (e.g., taxi-hailing service in New York vs. that in Beijing) have very different query/update loads. State-of-the-art kNN solutions are mostly designed with the objective of minimizing query processing time. In particular, elaborate index structures are used to speed up queries, often at the expense of slower updates. As we will show later, this may not lead to the best system performance, especially when updates constitute a non-trivial portion of the system load. A good kNN solution

should be tunable for the system to strike the best balance between query and update processing when it is applied to a wide spectrum of applications and environments with very different query/update loads.

Performance adaptability refers to how well a kNN solution can be configured to meet a certain performance requirement. As we have mentioned, the chief design objective of most traditional kNN algorithms is to optimize query time. As pointed out in [10], query time, which is the amount of time taken to process an isolated query, is a *micro measure* of system performance. In practice, *macro measures*, which are holistic measures of a system, are of greater significance. For example, from the perspective of a user, *query response time*, which is the amount of time between the arrival of a query to the reporting of a result, is the primary measure; from the perspective of the business that operates a location-based service, *throughput*, which is the number of queries served per unit time, is a very important measure. A good kNN solution should be tunable so that a given performance measure, such as query response time or throughput, can be optimized.

Meeting all three adaptability requirements poses a big challenge in the design of a kNN solution. For example, most existing kNN algorithms are sequential. These algorithms generally employ complex data structures and logic, which makes it very difficult to rewrite the codes and execute them as multithreaded processes on many-core machines. Although there are tools, such as YUCCA and Intel Parallelization Compiler (IPC), which automatically convert a single-threaded program to a multi-threaded one, our investigation shows that these tools are ineffective when they are applied to convert kNN programs. To illustrate, we applied IPC to convert implementations of three kNN algorithms, namely, *Dijkstra*, V-tree, and TOAIN, and executed them on an 8-core machine. The multithreaded version was less than 2% faster than the single-threaded version. The reason of the poor performance gain was that these kNN algorithms are based on graph exploration, which is intrinsically sequential and iterative — the next node to explore is chosen based on the nodes that have already been explored. Indeed, our experiments of applying IPC to the kNN programs showed that only the part on data initializations (which accounted for 2% of the total computation) were successfully parallelized; the main logic of the programs remained sequential.

Meeting the workload adaptability and the performance adaptability requirements necessitates a highly tunable kNN solution. As reported in [10], most existing kNN solutions fall short in providing a wide gamut of query/update processing times. For example, experiments reported in [10] show that under a high update load, the more sophisticated V-tree method gives a much lower throughput compared with the basic *Dijkstra* algorithm. This is because a high update load induces much update processing for V-tree, which employs a complex index structure. On the contrary, by not maintaining an elaborate index, *Dijkstra* weathers the storm of updates gracefully with very efficient update processing. In [10], the TOAIN algorithm is proposed, which employs a very flexible index structure called SCOB to answer kNN queries. The SCOB index is highly tunable to provide either very efficient query processing or very efficient update processing.

Our MPR Approach. In this paper we propose the MPR (Multi-layer Partitioning-Replication) mechanism that converts a single-threaded kNN solution into one that meets the three adaptability requirements. Here, we highlight two design principles of MPR.

(1) We focus on multicore technology and multiprocessing as a means to improve a system's computation power. As the rate of CPU clock speed improvement diminishes in recent years, multicore technology has become a major approach to improve CPU performance. For example, the Intel Xeon Phi processor family offer up to 72 cores per CPU. Lower-end models with 4-16 cores are also commonly used. The key idea of MPR is to orchestrate the multiple cores to serve a given mixture of query/update loads so that certain macro measures are optimized.

(2) We integrate existing single-threaded kNN solutions to our MPR framework through an extremely lightweight wrapper that interfaces a solution to our system. In particular, we do not design new kNN index structures or algorithms, and there are no "invasive procedures" that significantly modify the original code of a kNN solution. This is in sharp contrast to parallelizing a kNN program by code rewriting, which requires much engineering efforts.

Contributions. We summarize our contributions as follows.
- We put forward three adaptability requirements in the design of kNN solutions for large-scale location-based services. We point out the inadequacy of existing state-of-the-art solutions in meeting these requirements.
- We propose MPR, a mechanism that provides straightforward porting of a single-threaded kNN solution to a multicore server. MPR intelligently schedules the cores to serve a given mixture of query and update loads to optimize for a given macro measure.
- We provide mathematical models that estimate query response time and system throughput. Our models express the macro measures as functions of a few key systems elements, which include (1) a given single-threaded kNN algorithm (expressed in terms of its query/update processing time), (2) the number of CPU cores used, and (3) query/update loads. Based on our mathematical models, MPR analytically determines the optimal strategy of utilizing the cores to achieve the best system performance in terms of a given macro measure.
- We conduct experiments to evaluate MPR's effectiveness. We compare MPR experimentally against other baseline methods for answering kNN queries on a multicore platform. Our results show that MPR outperforms those baseline methods. It is highly effective and is very robust in its ability to adapt to a wide spectrum of environments with different number of CPU cores, different query/update loads, and different system performance objectives.

The rest of the paper is organized as follows. Section II mentions some related works. Section III describes the idea of data replication and partitioning, which are the basic building

blocks of MPR. Section IV describes the MPR mechanism and provides the mathematical analysis based on which MPR optimizes system performance. Section V presents experimental results. Finally, Section VI concludes the paper.

II. Related Works

The problem of answering kNN queries on road networks have been extensively studied [6], [14], [15], [8], [17], [16], [9], [10], [3]. In this section we briefly mention a few representative solutions. These include *Dijkstra* [6], IER [14], DisBrw [15], ROAD [8], G-tree [17], V-tree [16], G-Grid [9], and TOAIN [10].

Dijkstra's algorithm [6] (*Dijkstra*) is a well-known algorithm for determining the shortest paths from a source node to other nodes in a graph. Starting from a source node s, *Dijkstra* visits other nodes in the order of their distances from s. To answer a kNN query from a query point q, we run *Dijkstra* from q and explore the graph just enough to locate the k closest objects to q. *Dijkstra* does not use an elaborate index and therefore has very low object update costs.

ROAD [8] is based on *Dijkstra*. It speeds up kNN query processing for cases where objects are sparsely located in a network. Specifically, ROAD partitions a graph into many subgraphs (called Rnets). These Rnets are merged to form larger Rnets in a hierarchical fashion. An indicator is associated with each Rnet signaling whether the Rnet contains any objects. During a *Dijkstra* expansion, if an Rnet with no objects is to be explored, the search inside the Rnet is skipped. Compared with *Dijkstra*, ROAD gives a faster query time at the expense of an update cost; when an object is updated, the indicators of some Rnets have to be updated accordingly.

G-tree [17] and V-tree [16] build a similar subgraph hierarchy like ROAD, but instead of an indicator, each subgraph is associated with other information. Specifically, G-tree uses *Occurrence-Lists (OLs)* to record the objects that are located in the sub-graphs, while V-tree identifies *border nodes* that are at the *boundaries* of subgraphs. By maintaining the lists of nearest objects to the border nodes, V-tree is able to answer kNN queries very efficiently. It is shown that V-tree generally outperforms G-tree and ROAD [16].

G-Grid [9] is a GPU-based method for accelerating object updates. G-Grid makes use of the *butterfly shuffle* operation, which allows low-cost swapping of data between GPU threads. Since butterfly shuffling is not supported by multicore CPUs, we do not further consider G-Grid in this paper.

TOAIN [10] is a recently proposed kNN algorithm. It employs a SCOB index, which is highly configurable to optimize for either query or update processing time. Given a query/update load, TOAIN performs workload analysis to configure the SCOB index such that throughput is optimized. Our MPR framework shares similar philosophy with TOAIN in optimizing for macro measures through workload analysis. However, while TOAIN optimizes *throughput* by *configuring the* SCOB *index*, MPR can optimize either *throughput* or *query response time* by *scheduling the CPU cores*. In fact, as we will show later, MPR can easily employ TOAIN as the single-threaded kNN solution under its framework — TOAIN's configuring of the SCOB index and MPR's scheduling of the CPU cores (to execute TOAIN's queries and updates processes) can work hand-in-hand to achieve the best system performance.

There are other existing kNN solutions, such as IER [14] and DisBrw [15]. Since they are generally outperformed by V-tree, we do not consider them in this paper. Instead, we consider *Dijkstra*, V-tree, and TOAIN as the single-threaded kNN solutions, and evaluate the MPR framework on its efficiency in executing these solutions on a multicore platform. We chose *Dijkstra* because it is the most basic solution without using an elaborate index. Its update cost is thus low. We chose V-tree because it is generally the most query-efficient solution. Finally, we chose TOAIN because it is the most adaptable solution to varying query/update loads.

There are also a number of works on parallel graph processing [5], [13], [7], [12], [11]. These studies put forward general computational models (e.g., MapReduce, Vertex-centric computing) for graph algorithms. However, these models provide poor supports for application-specific optimizations. In particular, it is very difficult to implement highly-optimized sequential algorithms, such as V-tree and TOAIN, on these parallel graph processing frameworks. In contrast, our approach (MPR) provides a simple and general method for executing single-threaded programs on a multicore platform.

III. Replication and Partitioning

Load partition and *data replication* are two classic approaches to parallel computing [4]. In this section we outline the idea of multiprocessing a single-threaded kNN solution on a multi-core machine through data replication and partitioning. We consider a system whose workload is presented as a single stream of kNN queries and object updates with stochastic arrivals. We focus on scheduling the CPU cores to serve queries and updates to achieve multiprocessing. We are interested in scenarios in which query/update loads are high. In such scenarios, a query/update has to be queued when all cores are occupied upon its arrival. We assume a FCFS queuing discipline. Moreover, we require that the execution of queries and updates be equivalent to a serial execution in the tasks' arrival order.

Let us first consider two straightforward schemes, called *Full-Replication* (F-Rep) and *Full-Partitioning* (F-Part), of utilizing the multiple cores. We use $C_1, \ldots, C_\eta$ to denote the η cores in the system that we use to serve kNN queries and updates. We call these cores *worker cores*, or *w-cores* for short. Each worker core C_i exclusively accesses a data structure (e.g., an index) that maintains a subset, $\mathcal{M}_i$, of the object set $\mathcal{M}$. Also, each core is associated with a FCFS queue (called a *worker's task queue*, or *w-queue* for short) in which queries/updates yet to be served by the core are lined up. When a worker core is free, it retrieves the task at the head of its w-queue and executes the task based on a single-threaded kNN solution (e.g., V-tree). Figure 1(a) illustrates the schemes by showing the contents of the w-queues in a system

with 2 worker cores. We also show on the left of the figure the content of the task queue if the tasks were to be executed in a single-core machine. (For convenient, we call this queue the *g-queue*, for *global queue*.)

Under the full replication (F-Rep) scheme, the object set $\mathcal{M}$ is fully replicated at each w-core, i.e., $\mathcal{M}_i = \mathcal{M},\ \forall 1 \leq i \leq \eta$. If we employ a certain single-threaded kNN solution, any data structures used by the kNN solution are replicated at each w-core. Since each w-core maintains a full copy of the data, a query can be processed by any core. Updates, however, have to be installed in all the copies. Hence, an update has to be replicated and be processed by all the w-cores. We use a *scheduler core* (or s-core for short) to manage the stream of query/update tasks that arrive at the system. In particular, the s-core dispatches queries to the w-cores in a round robin fashion for load balancing. It also replicates updates and put them into the w-queues of all the w-cores. In Figure 1(a), under F-Rep, we see that the eight queries (Q1-Q8) are evenly distributed to the two worker cores. We observe that the w-queues are shorter than the g-queue leading to shorter queuing delay for the queries compared with a single-core system. For example, if the service time of each query/update is 1 time unit[1], then Q8's queuing delay is 9 units for a single-core system (7 queries + 2 updates in front of it in the g-queue), while that is 5 units for a 2-worker-core system under F-Rep (3 queries + 2 updates in front of Q8 in Core 2's w-queue). This improves query response time.

Another basic approach to utilizing the multiple cores is full partitioning (F-Part). Under this scheme, the object set $\mathcal{M}$ is partitioned into η partitions, each being managed by a core. That is, $\bigcup_{i=1}^{\eta} \mathcal{M}_i = \mathcal{M}$ and $\mathcal{M}_i \cap \mathcal{M}_j = \emptyset,\ \forall i \neq j$. The partitioning can be done in a number of ways. For example, objects in $\mathcal{M}$ can be distributed to the cores in a round robin fashion. This balances the update loads across the cores if objects generate updates at a similar rate (e.g., in a taxi-hailing application, vehicles report their locations at the same periodicity). If objects are updated at different rates, we can distribute the "updates" instead of the "objects" over the w-cores to balance the update loads. Specifically, kNN solutions generally handle an object's location change by a delete operation (on its index) followed by an insert operation. To record a location change of an object $o \in \mathcal{M}_i$, we execute a delete update by Core C_i followed by an insert update (to some partition $\mathcal{M}_j$). To balance the load, insert updates are distributed to the w-cores in a round robin fashion. For applications where object deletes and inserts do not come in pair (e.g., in *Pokémon GO*, a *Pokémon* appears (insert) at or disappears (delete) from a location, but may not move from one location to another), we apply a similar load balancing strategy: while a delete has to be executed by the w-core C_i whose partition $\mathcal{M}_i$ contains the to-be-deleted object, object inserts are distributed "round-robin-ly" over the w-cores. In the following discussion, we assume that a suitable update load balancing scheme is put in place.

Under F-Part, there is no data replication, hence, each update (insert/delete) is handled by one w-core. Query processing, however, has to involve all the cores. In particular, the query results obtained from the w-cores have to be aggregated. Specifically, given a query point q, each w-core returns (at most) k nearest objects to q for a total of (at most) $k \cdot \eta$ objects. We use an *aggregator core* (or a-core) to collect these objects, compare their distances from q, and return the k closest ones to q as the final query result. In Figure 1(a), we show the w-queues under F-Part. In the illustration, the updates are distributed over the w-cores while queries are replicated over them. Query response time is improved (although mildly) compared with the single-core system. For example, the queuing delay for Q8 is 8 units under F-Part, which is one unit shorter than that of the single-core system.

We remark that both F-Rep and F-Part can be easily implemented for multiprocessing a single-threaded kNN solution on a multicore machine. Essentially, one only needs to program the simple logic of a scheduler core and (for F-Part) an aggregator core. There is no sophisticated parallel indexing or parallel query/update processing. It also works with most existing kNN solutions and with (basically) any number of cores. These schemes thus satisfy our *system adaptability* requirement. However, F-Rep and F-Part are too "rigid" in the way they utilize the w-cores. In essence, F-Rep replicates updates and distributes queries, while F-Part replicates queries and distributes updates. Hence, F-Rep is fully query-friendly while F-Part is fully update-friendly. This inflexibility makes them not good solutions in meeting the *workload adaptability* and the *performance adaptability* requirements. For example, in Figure 1(a), we see that F-Rep gives shorter queuing delay (and thus better query response time) than F-Part. This is because the workload is query-heavy. In Figure 1(b), we illustrate a reversal of fortune of the two schemes with an update-heavy workload.

To meet the workload and performance adaptability requirements, we need a more flexible scheme that can determine the best arrangement (i.e., replication vs. partitioning) of the w-cores. In particular, this arrangement should be (1) workload cognizant and (2) optimizing for a given target macro measure. In the next section, we introduce MPR, which generalizes F-Rep and F-Part for meeting the above design objectives.

Finally, we remark that most kNN solutions decouple road network index and object index. Under our schemes, we only partition objects among the cores. Hence, each core maintains an index of its own object partition; The index structure of the road network, on the other hand, is shared by all cores.

IV. MPR

The basic idea of MPR is to organize the CPU cores into z layers, each being a core formation that is a hybrid of F-Rep and F-Part. Whether a layer formation leans more towards F-Rep or more towards F-Part is controlled by a pair of parameters x, y. We call the triple (x, y, z) an MPR

[1]In practice, query time is usually much larger than update time with existing kNN solution. We make this simplification just for illustration purpose.

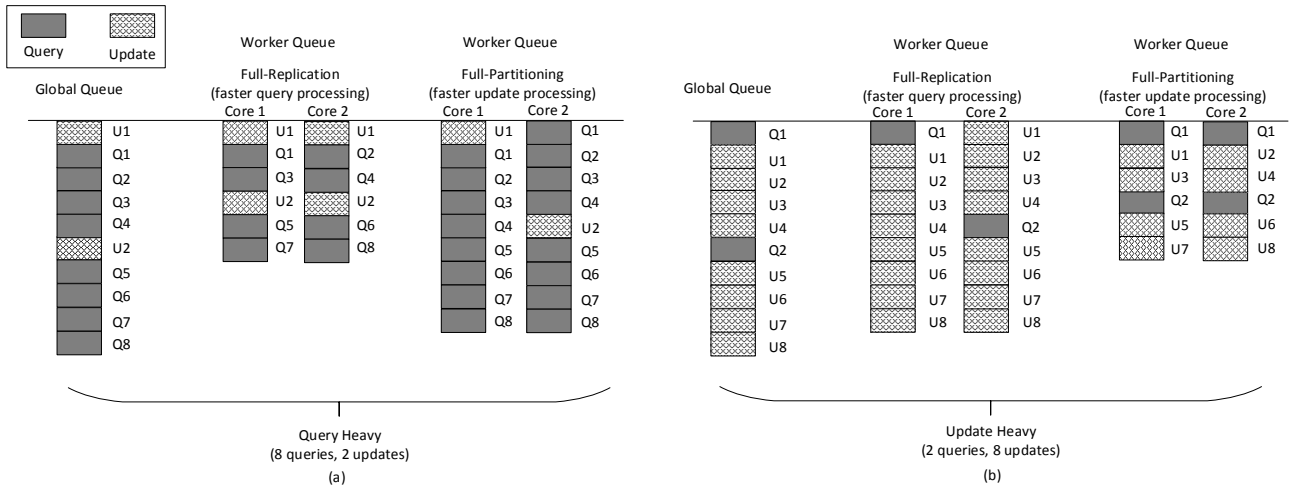

Fig. 1. Comparing queue contents under F-Rep and F-Part for (a) query-heavy scenario and (b) update-heavy scenario.

configuration. Given (1) a single-threaded kNN solution, (2) a query/update workload, and (3) a macro measure, MPR self-configures (i.e., automatically determines a configuration triple) by analytically solving an optimization problem. For presentation purposes, we first present a single layer formation, called a *core matrix* (Section IV-A). Then, we present an analytical study and show how a single layer MPR configuration is determined for optimizing either query response time or throughput (Section IV-B). Next, we extend it to a multiple layer MPR scheme (Section IV-C).

A. Core Matrix

A core matrix consists of $x \cdot y$ worker cores, one scheduler core, and one aggregator core, for a total of $x \cdot y + 2$ CPU cores. Figure 2 shows an example with $x = 4$ and $y = 3$. Figuratively, the $x \cdot y$ w-cores are arranged in an x-by-y matrix with x columns and y rows. We use $Col(i)$ and $Row(j)$ to represent the set of w-cores in the i-th column and those in the j-th row, respectively. The object set $\mathcal{M}$ is replicated y times. Each row of w-cores manage one data copy. Within each row, the data copy is partitioned among the x w-cores of the row. We use $C_{i,j}$ to denote the core in $Col(i)$ and $Row(j)$; we use $\mathcal{M}_{i,j}$ to denote the object set maintained by Core $C_{i,j}$. In other words, $\bigcup_{C_{i,j} \in Row(j)} \mathcal{M}_{i,j} = \mathcal{M}\ \forall 1 \leq j \leq y$ and $\mathcal{M}_{i_1,j} \cap \mathcal{M}_{i_2,j} = \emptyset\ \forall i_1 \neq i_2$. The core matrix arrangement can be seen as a hybrid of F-Rep and F-Part — It is not complete replication because no core maintains the complete data set (unless $x = 1$); It is not complete partitioning either because (as we will explain later) cores located in the same column maintain the same set of objects. The configuration parameters x, y control the degree of partitioning and replication, respectively. We next describe in greater detail how the various cores handle queries and updates.

[Scheduler Core] The s-core receives a single stream of queries and updates. When a query q arrives, the s-core selects a row, say $Row(j)$, in a round robin fashion to process the query. The s-core then appends the query q into the w-queues of all the w-cores in $Row(j)$. The s-core then appends a record (q,j) into the *task queue* of the aggregator core, informing the a-core that an outstanding query q is to be served by the w-cores in $Row(j)$.

When an update arrives, the s-core considers two cases. If the update is an insert of an object o, the s-core selects a column, say $Col(i)$, in a round robin fashion. It appends the insert update into the w-queues of all the w-cores in $Col(i)$ and registers a record (o,i) in a hash table. The object o is to be inserted into the data partitions held by all the w-cores in $Col(i)$. If the update is a delete of an object o, the s-core looks up a record (o,i) from the hash table. It then appends a delete update to the w-queues of all the w-cores in $Col(i)$.

[Worker Core] A w-core executes the tasks given to it in its w-queue by running a single-threaded kNN solution. We assume that a kNN solution $\mathcal{A}$ provides three interfaces, namely, $\mathcal{A}.Q(l,k)$ (query the k closest objects from location l), $\mathcal{A}.I(o,l)$ (insert object o at location l), and $\mathcal{A}.D(o)$ (delete object o). To process a query/insert/delete task, a w-core executes the corresponding module of $\mathcal{A}$. After processing a query q and obtaining a result r, the w-core appends a record (q,r) into a *result queue* of the a-core for further aggregation.

[Aggregator Core] When the a-core receives a query record (q,j) from the s-core, it maintains a partial answer of q. Upon receiving a result record (q,r) from a w-core, the a-core updates the partial answer of q. When all w-cores in $Row(j)$ have submitted their results, the a-core returns the answer as the final answer.

Algorithms 1-3 show the pseudo codes executed by the three types of cores. We remark that our scheme ensures correct query processing. Firstly, a query q is processed by all the w-cores in a row of the core matrix. Since the w-cores of a row collectively partition the complete object set $\mathcal{M}$, the kNN results aggregated by the a-core that are based on the partial results reported by the row's w-cores are derived from the complete object set. Secondly, queries and updates are processed in an order that is equivalent to a serial execution

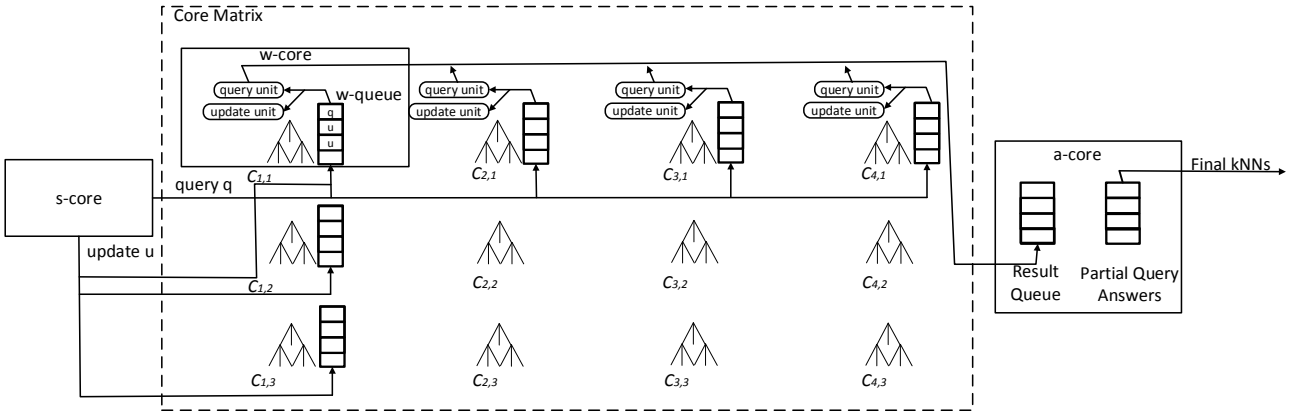

Fig. 2. Core matrix.

of the tasks in their arrival order. To see this, let us consider a query q and an update u of an object o such that u arrives at the system before q. Suppose the s-core chooses $Col(i)$ to process the update and $Row(j)$ to process the query. In this case, the s-core will submit u to the w-queue of Core $C_{i,j}$ before it submits q to the same w-queue. Effectively, u and q are "serialized" in the order of u followed by q at Core $C_{i,j}$. This guarantees that any query q is effectively computed after all updates that arrive before q are processed.

Algorithm 1: `Scheduler-Core(`C, $\mathcal{A}$, T`)`

input : C: the core matrix; $\mathcal{A}$: the give single threading algorithm; T: the arrived task.

$j \leftarrow 0$; $i \leftarrow 0$
if T *is a query* **then**
  for $i = 1$ *to* x **do**
    append the query task T into the w-queue of the w-core $C_{i,j}$
  $j \leftarrow (j+1)\%y$
if T *is an insert* **then**
  $o \leftarrow T.insert\_object$
  for $j = 1$ *to* y **do**
    append the insert task T into the w-queue of the w-core $C_{i,j}$
  $hash[o] \leftarrow i$
  $i \leftarrow (i+1)\%x$
if T *is a delete* **then**
  $o \leftarrow T.delete\_object$
  $i' \leftarrow hash[o]$
  for $j = 1$ *to* y **do**
    append the delete task T into the w-queue of the w-core $C_{i',j}$

Algorithm 2: `Worker-Core`

while *w-queue is not empty* **do**
  $T \leftarrow$ the top task from w-queue
  if T *is a query* **then**
    $r \leftarrow \mathcal{A}.Q(T.query\_location, k)$
    send $(T.query\_id, r)$ to a-core
  if T *is an insert* **then**
    $\mathcal{A}.I(T.insert\_object, T.insert\_location)$
  if T *is a delete* **then**
    $\mathcal{A}.D(T.delete\_object)$

With the core matrix arrangement, queries are distributed across the y rows while updates are distributed across the x columns. This allows our scheme to adapt to different query/update loads by adjusting the configuration parameters x, y. For an environment with a heavy query load, we should increase y so that more queries can be executed in parallel. For an update-heavy environment, we should increase x instead.

Algorithm 3: `Aggregator-Core(`q, r`)`

input : q: query id; r: partial answer for q.

Update the partial answer for query q using r
if *all the partial answers for* q *have been processed by the a-core* **then**
  send out the partial query answer for q as its final query answer
  remove the entry for q from the partial query answers

B. Optimization

Our objective is to determine the best configuration x, y given a workload, a single-threaded kNN solution, and a target macro measure such as query response time and throughput. Our approach is to formulate a mathematical model to express a macro measure in terms of a workload's characteristics and a kNN solution's characteristics. We first illustrate our approach with a query response time analysis, which is followed by a throughput analysis.

1) Query response time: Given a single-threaded kNN solution $\mathcal{A}$ (e.g., V-tree), let t_q and V_q be the average and the variance of the executing time of an isolated query at a worker core. Also, let t_u and V_u be those of an isolated update. The set of values (t_q, V_q, t_u, V_u) characterize solution $\mathcal{A}$. We assume that these values can be obtained via a simple empirical study for a given application (e.g., by executing isolated queries and updates on a single core with a given set of objects $\mathcal{M}$ and collecting execution times statistics). We characterize a workload by (λ_q, λ_u), where λ_q and λ_u are the arrival rates (in number of second) of queries and updates, respectively. We assume query (update) arrivals form a Poisson process.

The response time of a query q consists of three components:

(1) t_s: the amount of time taken by the scheduler core to process q. This includes selecting a row, say, $Row(j)$, for processing the query and appending q to the w-queues of all x w-cores in $Row(j)$.

(2) t_w: the amount of time q spends at a worker core. This includes the queuing delay spent in the w-queue and the processing time taken by a w-core on q[2].
(3) t_a: the amount of time taken by the aggregator core to aggregate the partial kNN results computed by the x w-cores in $Row(j)$.

We note that t_s is dominated by the time taken to write into the x w-queues of the worker cores in a row and t_a (results merging) is proportional to x. We thus model $t_s + t_a = \tau \cdot x$, where τ is a constant that can be determined empirically. The average query response time, R_q, is given by:

$$R_q = \overline{t_w} + \tau \cdot x. \tag{1}$$

The expected value of t_w, denoted by $\overline{t_w}$, can be determined using the following lemma.

Lemma 1. *If a w-core is not overloaded (i.e., the task arrival rate at the core does not exceed its servicing rate), then*

$$\overline{t_w} = \frac{\lambda_q t_q^2(1+\gamma_q)/\left\lfloor \frac{\eta}{x} \right\rfloor + \lambda_u t_u^2(1+\gamma_u)/x}{2(1-\lambda_q t_q/\left\lfloor \frac{\eta}{x} \right\rfloor - \lambda_u t_u/x)} + t_q, \tag{2}$$

where $\gamma_q = V_q/t_q^2$ and $\gamma_u = V_u/t_u^2$ are the squared coefficients of variation of t_q and t_u, respectively; η = number of available cores in the system that can be deployed as worker cores.

Proof. For a single FCFS queuing system that serves a mixture of queries and updates such that query and update arrivals form two independent Poisson processes with arrival rates λ_q^* and λ_u^*, respectively, it is shown in [10] that the expected query response time, T_q, for a single queue system is given by:

$$T_q = \frac{\lambda_q^*(V_q + t_q^2) + \lambda_u^*(V_u + t_u^2)}{2(1-\lambda_q^* t_q - \lambda_u^* t_u)} + t_q. \tag{3}$$

If we consider a worker core and its w-queue as such a single queue system, and put Equation 3 in the context of this system, we have the following mapping of the variables:

$$T_q \rightarrow \overline{t_w};\ \ \lambda_q^* \rightarrow \lambda_q/y;\ \ \lambda_u^* \rightarrow \lambda_u/x. \tag{4}$$

This is because the average amount of time a query spends at a w-core ($\overline{t_w}$) is the average response time (T_q in Equation 3) a query experiences at the worker core's queuing system. Also, since our scheme distributes queries across the y rows and updates across the x columns, the query/update arrival rates at each w-cores are λ_q/y and λ_u/x, respectively. Furthermore, the number of worker cores deployed (= $x \cdot y$) should not exceed the number of available worker cores η. Hence, $y = \lfloor \eta/x \rfloor$. Finally, by definition, $V_q = \gamma_q t_q^2$ and $V_u = \gamma_u t_u^2$. Applying all these substitutions and those in (4) to Equation 3, we obtain Equation 2. □

[2]Technically speaking, q is processed by all x w-cores in $Row(j)$. The delay experienced by q at each of the x w-cores could be different. However, our scheme schedules the same number of queries to all the w-cores in a row and the updates are distributed to the w-cores in the row in a round robin fashion. Each w-core in the same row is thus given almost the same loads. From our experiments, the differences in the queuing delays experienced at different w-cores of the same row is negligible. We thus model t_w as the time spent on one w-core to simplify the mathematical analysis.

Combining Equations 1 and 2, we have

$$\begin{aligned} R_q &= \frac{\lambda_q t_q^2(1+\gamma_q)/\left\lfloor \frac{\eta}{x} \right\rfloor + \lambda_u t_u^2(1+\gamma_u)/x}{2(1-\lambda_q t_q/\left\lfloor \frac{\eta}{x} \right\rfloor - \lambda_u t_u/x)} + t_q + \tau \cdot x \\ &= F(x). \end{aligned} \tag{5}$$

Given a kNN solution's characteristics (t_q, V_q, t_u, V_u), a system's characteristics (η, τ), and a workload's characteristics (λ_q, λ_u), the average query response time R_q can be expressed as a function $F(x)$ based on Equation 5. To optimize the system for query response time, we determine the value of x that minimizes $F(x)$. This can be easily done by evaluating $F(x)$ for $1 \le x \le \eta$. By selecting the optimal value of x, our core matrix scheme can be adjusted to adapt to different system, workload, and kNN solution environments to achieve the best query response time.

2) Throughput: We can perform a similar analysis on throughput. We define throughput as the number of queries that can be processed by the system per unit time, which can be expressed as the query arrival rate, λ_q. From the perspective of a location-based service provider, a performance measure of interest is the *maximum throughput* that the system can sustain. We use $\widehat{\lambda_q}$ to represent this maximum throughput. Two practical concerns limit $\widehat{\lambda_q}$. First, as queries are admitted to the system at higher rates, queues build up. For example, with a simple M/M/1 queue, the expected amount of time that a task spends in a queuing system (queuing delay + servicing time) is $1/(\mu - \lambda)$ where λ and μ are the task arrival rate and servicing rate, respectively. As the task arrival rate increases (and approaching μ), the amount of time a task spends in the system approaches infinity. To guard against this runaway scenario, we impose a query response time bound, denoted by R_q^* as a *quality-of-service* requirement. Specifically, $\widehat{\lambda_q}$ cannot be so large that the resulting query response time exceeds R_q^*. Second, the total processing time of all queries and updates submitted to a worker core should not exceed the total capacity of the core. These two constraints can be mathematically expressed by the following inequalities:

$$\frac{\widehat{\lambda_q} t_q^2(1+\gamma_q)/\left\lfloor \frac{\eta}{x} \right\rfloor + \lambda_u t_u^2(1+\gamma_u)/x}{2(1-\widehat{\lambda_q} t_q/\left\lfloor \frac{\eta}{x} \right\rfloor - \lambda_u t_u/x)} + t_q + \tau \cdot x \le R_q^* \tag{6a}$$

$$\frac{\widehat{\lambda_q}}{\lfloor \eta/x \rfloor} t_q + \frac{\lambda_u}{x} t_u < 1 \tag{6b}$$

Solving these two inequalities gives

$$\begin{aligned} \widehat{\lambda_q} &\le \frac{\left\lfloor \frac{\eta}{x} \right\rfloor}{t_q} \min\Big\{ \frac{2(x-\lambda_u t_u)(R_q^* - t_q - \tau x) - \lambda_u t_u^2(1+\gamma_u)}{x t_q (1+\gamma_q) \lfloor \eta/x \rfloor^2 + 2x(R_q^* - t_q - \tau x)}, \frac{x - \lambda_u t_u}{x} \Big\} \\ &= G(x). \end{aligned} \tag{7}$$

From Equation 7, we see that the maximum throughput $\widehat{\lambda_q}$ is bounded by the function $G(x)$. To optimize throughput, we find the value of x that maximizes $G(x)$.

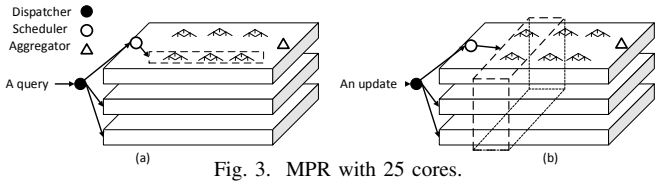

Fig. 3. MPR with 25 cores.

C. Multiple Layers

Under the core matrix organization, the s-core manages every query and update. In particular, it writes to x w-queues (in a matrix row) on receive of a query and to y w-queues (in a matrix column) on receive of an update. If it takes τ' seconds to perform such a write, the scheduler will be overloaded if $(\lambda_q \cdot x + \lambda_u \cdot y)\tau' > 1$. For moderate workloads (λ_q, λ_u) and moderate number of worker cores (x, y), the core matrix scheme usually suffices. However, as we scale our solution to larger systems and very heavy workloads, a single schedule core could become a bottleneck.

Our MPR scheme addresses this issue by structuring the cores in z core matrices. Figure 3 illustrates how the MPR scheme handles (a) a query and (b) an update. In the figure, each layer represents a core matrix, which stores a complete replicate of the data objects and operates as we discussed in the previous sections (see Figure 2). One core in the system is designated as the *dispatcher*, called the d-core. It handles every query and update and dispatches them to the s-cores of the layers. Specifically, when a query q arrives, the d-core distributes q to a layer in a round robin fashion. Within that layer, q is processed by x w-cores in a matrix row. This is illustrated by a stripe outlined in Figure 3(a). When an update u arrives, the d-core dispatches u to every layer. The s-cores of the layers schedule u to be processed by a column of worker cores in their respective layers. This is illustrated by a slice outlined in Figure 3(b).

Note that with z layers, the dispatcher communicates with only z schedulers. Also, the number of w-cores in each layer is η/z, where η is the total number of w-cores in the whole system. Compared with the single-layered case, there are fewer w-cores (a fraction of $1/z$) whose w-queues an s-core has to write into under the multi-layered structure. This solves the scheduler bottleneck problem.

We remark that with MPR, queries are distributed among z layers while updates are replicated and executed at all layers. This, at first sight, might seem to favor query processing than update processing. Interestingly, our optimization (via Equations 5 and 7) is able to avoid this favoritism. This is because the optimization procedure configures the number of rows and columns in a core matrix (i.e., it determines an x and hence a corresponding y) by taking a query/update workload (λ_q, λ_u) as input. With z layers, the query load of a layer (a core matrix) is reduced to λ_q/z while the update load of a layer remains λ_u. With a smaller query load, the configuration of a core matrix will adjust to have more columns (larger y) and fewer rows (smaller x). To obtain the best MPR configuration (x, y, z), we enumerate z and determine the optimal configuration of the core matrix for each value of z analytically (using Equations 5 and 7). The configuration (x, y, z) that optimizes a target macro measure will be chosen.

Before we end this section, let us shed more light on the optimality of our scheme, particularly on the core matrix structure. In Section IV-A, we describe how η worker cores are arranged into x columns and y rows, and how queries/updates are processed by the w-cores. Note that with this "rectangular structure", each row consists of the same number (x) of workers and that each row is given the same query load (λ_q/y). An interesting question is whether a non-rectangular arrangement of the w-cores could give even better performance (e.g., in query response time) than the optimal rectangular core matrix structure. If we consider the w-cores in a matrix row as a *group* of workers, then the *grouping* based on the core matrix is very *regular* — every group has the same number of workers. A more *generic* grouping of the η w-cores would allow groups (rows) with different numbers of workers. Moreover, the query loads assigned to the groups could also be different. We call this irregular arrangement a *generic grouping scheme*. We can show that the optimal configuration of our rectangular core matrix structure is optimal in query response time under any generic grouping schemes. The proof of this theorem is given in [1]. We remark that this result is only of theoretical interest. Nonetheless, it provides a theoretical guarantee on the performance of our system. Moreover, practically, a rectangular core matrix structure vastly simplifies the scheduling logic of the s-core, which makes our MPR scheme very easy to implement.

V. Experiment

We present experiment results for evaluating MPR's performance. We compare MPR against other baseline multi-processing schemes with a focus on their adaptability to the system/workload characteristics. In Section V-A, we describe the experiment setup, which include datasets, a workload generator, multiprocessing schemes, kNN solutions, and also performance measurements. In Section V-B we use an illustrative case to demonstrate how MPR adapts to a given set of system/workload characteristics much better than other baseline schemes. Finally, we compare MPR against the baselines over a wide range of application settings in Section V-C.

A. Setup

Data. We conduct our experiments on data of five real road networks, namely, Beijing (BJ), North West America (NW), New York City (NY), USA East (USA(E)), and USA West (USA(W)). Table I shows the characteristics of these networks. For BJ, we obtained trajectories of around $3,000$ taxis from UCAR [2], which is a popular taxi hailing service in Beijing. These trajectories are presented as a single stream of object (taxi) location updates. There are 8.74 million updates in the stream. For NW, we obtained a real dataset of points of interests (POIs) in NW, such as restaurants, hospitals, and schools. These POIs help us model location-based games or bike-sharing services in which the occurrences of objects are usually clustered at POIs. USA(E) and USA(W) are two large datasets for testing the scalability.

TABLE I
ROAD NETWORKS.

Symbol	Network	#Edges	#Nodes	Additional data
BJ	Beijing	2,690,296	1,285,215	3,000 taxi trajectories
NW	US North West	2,840,208	1,207,945	13,132 POIs
NY	New York City	733,846	264,346	
USA(E)	USA East	8,778,114	3,598,623	
USA(W)	USA West	15,248,146	6,262,104	

For each road network, we generate updates under two modes: taxi hailing mode (TH) and random update mode (RU). We use X-Y (e.g., BJ-TH) to denote a scenario of using road network X with update mode Y. For each scenario, we generate (1) a set $\mathcal{M}$ of objects and their initial locations, and (2) a stream of queries and object updates. Given a size $m = |\mathcal{M}|$, we randomly select m nodes in the network at each of which an object is created and placed. Queries are generated as a Poisson process at an arrival rate of λ_q. For RU, updates are generated as another Poisson process with arrival rate λ_u. Each update is either an insert or a delete with equal probability. For an insert, a new object o is created and a node is randomly pick at which o is placed; For a delete, an object is randomly picked and removed. For TH, we model an object's movement from a node u to a node v as a delete at node u followed by an insert at a neighboring node v. Object movements are generated as a Poisson process at an arrival rate of $\lambda_u/2$. Since a movement generates two updates (insert + delete), the update arrive rate is λ_u.

Update generation for scenarios BJ-TH and NW-RU are exceptions to the above rules because we have real data to generate the updates. Specifically, for BJ-TH, updates are given by the real UCAR trajectory data. For NW-RU, an *insert* update will only place an object at one of the POIs.

Multiprocessing schemes and kNN solutions. We consider four multiprocessing schemes, namely, F-Rep, F-Part MPR, and a special case of MPR in which we employ only one layer (i.e., $z = 1$). We denote the last scheme 1MPR (for 1-layer-MPR). We also consider three single-threaded kNN solutions, namely, *Dijkstra*, V-tree, and TOAIN. These solutions are described in Section II. *Dijkstra* uses minimal indexing and is thus more update-friendly; V-tree, on the other hand, uses a more elaborate index structure and is more query-friendly. TOAIN is a very flexible algorithm. It provides a family of indexes (some are more query-friendly and others are more update-friendly). Given a query/update load, TOAIN automatically selects an index that optimizes throughput. In our experiment study, we report the performance of TOAIN with the best index structure it picks.

We evaluated all combinations of multiprocessing schemes and kNN solutions. In the following, we use $M(S)$ (e.g., MPR(TOAIN)) to denote the case of applying a multiprocessing scheme M with a single-threaded kNN solution S. All algorithms are implemented in C++ using the native multithreading library and codes are compiled with the O3 flag. Experiments are conducted on a server with two 10-core Intel Xeon E5-2600 v3 (Haswell) processors and 96 GB of physical memory. We use 19 cores in our experiments.

Measurements. We measure the expected query response time (R_q) and the maximum throughput ($\widehat{\lambda_q}$) as two target performance metrics. We measure R_q by running the system for 200 seconds with a query/update stream generated as described previously. We compute the average response time of all the queries that are completed by the end of the 200 seconds execution run, and report the average as R_q. For the case in which a core is overloaded, (e.g., if $\lambda'_q t_q + \lambda'_u t_u > 1$ where λ'_q and λ'_u are the query and update arrival rates at a w-core, respectively), we report "Overload". To measure $\widehat{\lambda_q}$, we repeat the above 200-second run while gradually increasing the value of λ_q. We determine the largest λ_q that does not cause a core to be overloaded or R_q to exceed a response time bound R^*_q. The resulting λ_q is reported as $\widehat{\lambda_q}$.

B. Case Study

We evaluated the 4 multiprocessing schemes under a large number of scenarios including different road networks, kNN solutions, query/update loads, and parameter settings. In this section we give a detailed analysis on one illustrative scenario comparing the four schemes. In particular, we show how MPR searches for the best configuration to achieve the best system performance. Since we are interested in high-load situations (in which multiprocessing and thus a good scheme is needed), the scenario chosen in this section has relatively high query/update arrival rates. In Section V-C we summarize other illustrative cases of the hundreds of scenarios we have studied.

We consider BJ-RU (Beijing road network, random update mode). We set m = 10,000 objects, k = 10, λ_q =15,000, λ_u = 50,000, and use TOAIN as the single-threaded kNN solution. Table II shows the query response time for TOAIN running with one CPU core (top row) and those of the four schemes running on 19 available cores (next 4 rows). For the schemes, the configurations (x, y, z) are listed in the middle 3 columns; the number of d-core, s-cores, a-cores, and the total number of cores used are listed in the right 4 columns of the table. Note that when the number of partitions (x) = 1, our schemes do not apply any aggregator core (so # a-core = 0) because with a single partition, the result given by a w-core on a query is complete; no aggregation is needed. Similarly, when the number of layer (z) = 1, no dispatcher core is used.

From Table II, we see that running TOAIN on a single core leads to overload. This is due to the very heavy workloads. For example, to serve 15,000 queries per second, query processing time (t_q) has to be below 1/15,000 = 66μs. In our experiment, using TOAIN, we register a t_q of about 170 μs. Hence, a single core cannot handle the query load, let alone updates. Interestingly, using more cores (19) does not help if we apply the two baseline schemes F-Rep and F-Part; the system is still overloaded under either scheme. This is because F-Rep distributes queries across 18 cores, which helps handle the high query load. However, updates are replicated across them. This causes two issues: (1) Each worker core is overloaded by the heavy update load and (2) The scheduler has to write into all 18 w-queues for each update. With the very high update load, the scheduler is overloaded. The opposite can be said for F-Part, which is overloaded by the heavy query load.

TABLE II
QUERY RESPONSE TIME R_q (CASE STUDY)

Scheme	R_q (μs)	#partitions (x)	#replicas (y)	#layers (z)	#dispatcher	#schedulers	#aggregators	#cores used
TOAIN	Overload	–	–	–	–	–	–	1
F-Rep(TOAIN)	Overload	1	18	1	0	1	0	19
F-Part(TOAIN)	Overload	17	1	1	0	1	1	19
1MPR(TOAIN)	973	3	5	1	0	1	1	17
MPR(TOAIN)	**385**	1	3	4	1	4	0	17

TABLE III
MAXIMUM THROUGHPUT $\widehat{\lambda_q}$ (CASE STUDY)

Scheme	$\widehat{\lambda_q}$ (queries/s)	#partitions (x)	#replicas (y)	#layers (z)	#dispatcher	#schedulers	#aggregators	#cores used
TOAIN	8,791	–	–	–	–	–	–	1
F-Rep(TOAIN)	0	1	18	1	0	1	0	19
F-Part(TOAIN)	157	17	1	1	0	1	1	19
1MPR(TOAIN)	35,131	2	8	1	0	1	1	18
MPR(TOAIN)	37,640	1	8	2	1	2	0	19

The 1MPR scheme configures itself into an $x = 3$ by $y = 5$ core matrix based on the optimization that uses Equation 5. This arrangement strikes a balance between distributing queries and updates across the cores, which results in an R_q of slightly less than 1ms. MPR further reduces R_q by about 2.5 times to 385 μs. This is achieved by employing 4 layers (z = 4). With 4 scheduler cores, MPR avoids the scheduler bottleneck and gives a better performance.

An MPR configuration (x, y, z) uses xyz (w-cores) + 1 (d-core) + z (s-cores) + z (a-cores) cores. The exceptions are when $x = 1$, no a-cores are used and when $z = 1$, no d-core is used. With 19 available cores, there are 31 possible MPR configurations, among which MPR looks for the best one using Equation 5. We measure the query response times under all 31 configurations. The results are plotted in Figure 4. The x and z values of each configuration are shown (y is determined for given x and z). Points for configurations that share the same z are connected by a line for clarity. Out of these 31 configurations, the system is overloaded in 17 of them. These overloaded cases are illustrated in Figure 4 by points that hit the "ceiling" of the y-axis[3]. We can consider F-Rep, F-Part and 1MPR as special cases of MPR. In particular, 1MPR explores all the configurations with z = 1. The configurations adopted by each of the 4 schemes are labeled in Figure 4. From the plot, we see that choosing the right configuration is very critical as the response times given by the configurations differ widely. Also, MPR's strategy of employing multiple layers pays off as we see more non-overloaded configurations when the system uses 2 or more layers ($z \geq 2$). Finally, MPR is successful in locating the best configuration (in terms of response time) based on the analytical formula (Equation 5). Table III compares the four schemes in throughput performance. 1MPR and MPR configure themselves with throughput optimization using Equation 7. In this experiment, we set the response time bound R_q^* to 100ms. We start with a small λ_q and gradually increase it until either the system is overloaded or the response time bound is exceeded (i.e., $R_q > R_q^*$). The largest λ_q registered is reported as the maximum throughput ($\widehat{\lambda_q}$). From the table, we see that F-Rep on 19 cores gives 0 throughput. This is because F-Rep replicates updates and executes them on all the w-cores. Under the high update rate, the scheduler core is overloaded in communicating with all 18 w-cores for each update. F-Part distributes updates over 17 w-cores. It thus manages the update load well. However, F-Part does not distribute queries. Hence, it cannot cope with a high query load. This results in a very low throughput (157). 1MPR and MPR are doing much better giving throughputs of 35,131 and 37,640, respectively. An interesting observation is that 1MPR and MPR reconfigure themselves when the target macro measure changes from query response time to throughput; Comparing Tables II and III, we see that 1MPR changes its configuration from (3, 5, 1) to (2, 8, 1), while MPR changes its from (1, 3, 4) to (1, 8, 2). This shows that both 1MPR and MPR provide *performance adaptability*.

[3]Response time is undefined for overloaded cases. The (ceiling) points in the plot are shown for illustration purpose only.

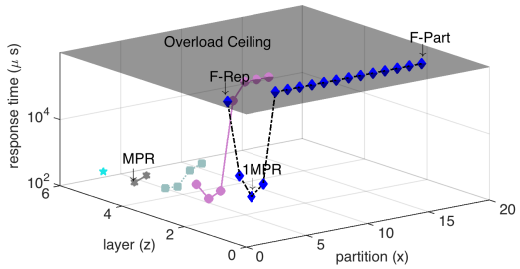

Fig. 4. MPR configurations (case study).

C. Adaptability

We have evaluated the schemes on 150 scenarios that cover a wide spectrum of kNN solutions, road networks, update models, number of CPU cores, and workloads. Out of these 150 cases, MPR gives the best query response time or throughput (depending on the objective macro measure) in 145 cases. This shows that MPR is highly adaptable to different system/workload/performance characteristics and requirements. In this section we further illustrate MPR's adaptability with a few selected sets of performance results.

Our first set of results show how the schemes adapt to different single-threaded kNN solutions employed. We compare the schemes under 2 scenarios: (1) An update-heavy scenario using the New York road network with random update mode (NY-RU), m = 80K objects, query arrival rate λ_q = 1.25K, and a heavy update arrival rate λ_u = 20K. (2) A query-heavy scenario BJ-RU, m = 10K, λ_q = 20K, λ_u = 10K. Figures 5

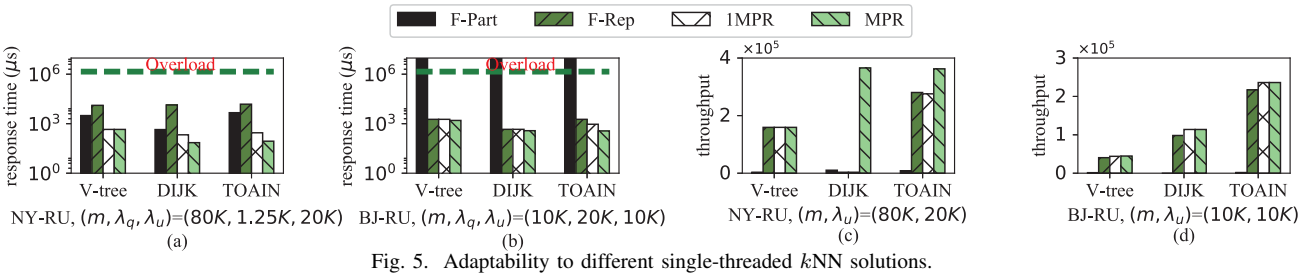

Fig. 5. Adaptability to different single-threaded kNN solutions.

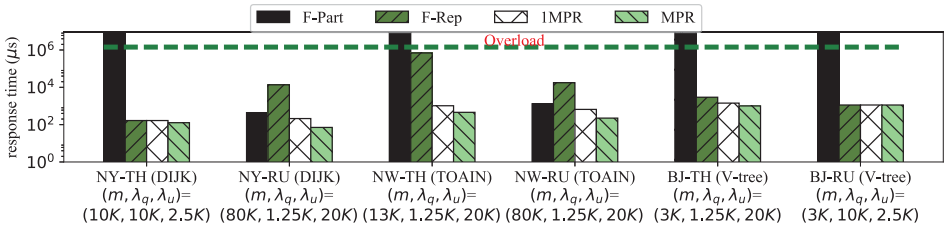

Fig. 6. Adaptability to different networks.

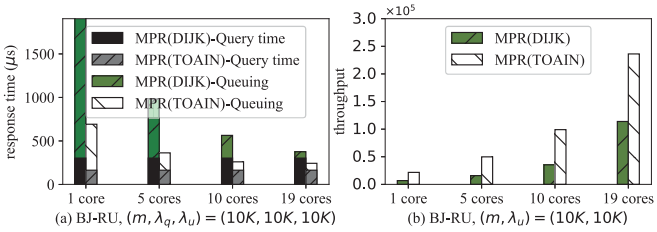

Fig. 7. Adaptability to different number of CPU cores.

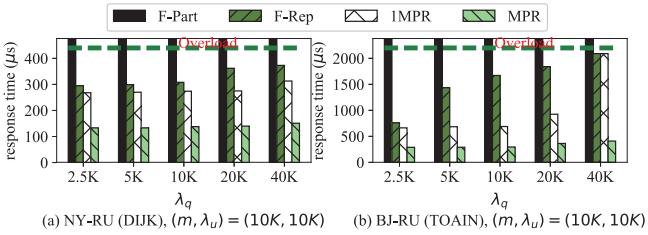

Fig. 9. Adaptability to different query loads.

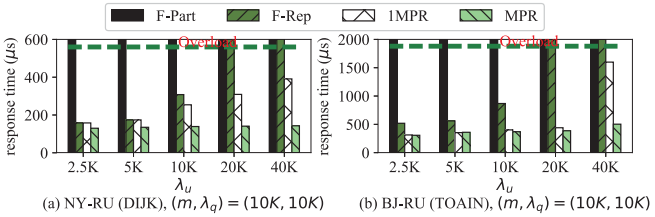

Fig. 8. Adaptability to different update loads.

(a) and (b) show the query response times of the four schemes when different kNN solutions are used for scenarios NY-RU and BJ-RU, respectively. Note that the response time axes are in log scale. Moreover, when a system is overloaded, the response time is illustrated by a tall bar that touches an "overload'ceiling"[3]. Recall that F-Part partitions the data and distributes updates over the w-cores, it is thus update-friendly. F-Rep, on the other hand, distributes queries over the cores and is thus query-friendly. For the update-heavy scenario (Figure 5(a)), we need an update-friendly scheme, and so F-Part outperforms F-Rep, giving smaller response times. Moreover, *Dijkstra* uses simple index with low update cost. Hence *Dijkstra* is also update-friendly. As a result, the response times given by the schemes are generally smaller when *Dijkstra* is used as the kNN solution. An interesting observation is that since V-tree is query-friendly (it speeds up query processing time by using an elaborate index) but not so update-friendly (compared with *Dijkstra*), the response times using V-tree in the update-heavy case (Figure 5(a)) are larger than *Dijkstra*. The opposite is true under the query-heavy scenario (Figure 5(b)). In particular, F-Part cannot handle the heavy query load resulting in system overloading. 1MPR and MPR, as seen from the figures, work very well with all the kNN solutions under both scenarios. Figures 5(c) and (d) show the schemes' performances under the two scenarios when throughput is the target measure for optimization. We see that MPR significantly outperforms the others. In particular, for scenario NY-RU(*Dijkstra*), MPR is the only scheme that can provide a significant throughput. This again shows the adaptability of MPR.

Our next set of results show how the schemes adapt to different combinations of road networks, update modes, and workloads. Figure 6 shows the response times given by the schemes under 6 scenarios. Comparing F-Rep and F-Part, we see that each outperforms the other in some cases. The reason is similar to that in our previous discussion. For example, the NY-RU scenario (2nd group of bars in Figure 6) is an update-heavy scenario (with λ_u = 20K). For this case, F-Part performs better than F-Rep because F-Part is update-friendly. From the figure, we see that MPR consistently performs much better than the other 3 schemes over all the scenarios.

Figure 7 shows how MPR scales with the number of CPU cores available in the system. We display the query response times (top) and throughputs (bottom) in the two sub-figures. *Dijkstra* and TOAIN are two kNN solutions employed. We see that under the environment tested, a single core machine cannot handle the loads and that results in system overload when the simple *Dijkstra* algorithm is used. MPR is able to

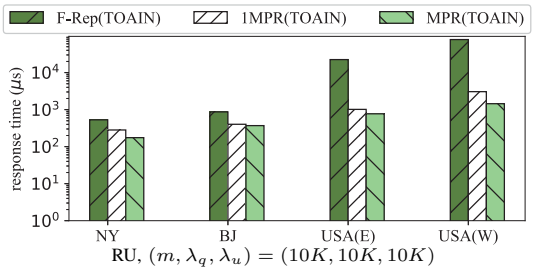

Fig. 10. Scalability with respect to network sizes.

take advantage of more CPU cores as query response time decreases and throughput increases when the number of CPU cores increases. In Figure 7(a), we break down response time into its two components, namely, the queuing delay and the query time. We see that MPR is able to significantly reduce the queuing delay when given more cores The improvements in response time and throughput are even more pronounced with *Dijkstra*.

Figures 8 and 9 show how the schemes adapt to different update and query loads, respectively. We pick two scenarios NY-RU and BJ-RU for illustration and vary either λ_u or λ_q in the figures. We make the following observations. First, the response times for the BJ network are generally higher than their counterparts for the NY network. This is because the BJ network is about 5 times larger than NY. Second, we see that F-Part cannot handle the loads and it suffers from overloading in all cases. Third, for the other 3 schemes, their response times increase with the loads. In particular, in Figure 8, we see that the response time under F-Rep increases significantly as λ_u increases. This is because F-Rep is not update-friendly and so it is very sensitive to the update load. In contrast, the response time under 1MPR increases relatively mildly. This is because 1MPR is able to adjust its configuration to handle a larger update load. For example, for the NY network (Figure 8(a)), 1MPR changes its (x, y) configuration from (1, 18) at λ_u = 2.5K to (5, 3) at λ_u = 40K. Moreover, we see that the response time under MPR is even less sensitive to λ_u. This is because MPR provides one more dimension (z) in its configuration, which allows the scheme to be even more adaptable to high load situations. In Figure 9, we see that the response time under F-Rep increases less dramatically with λ_q. This is because F-Rep is query-friendly. Again we see the advantage of MPR over the other schemes. It gives the best performance in response time for all the cases, outperforming the baseline schemes by wide margins.

We conduct a scalability experiment on the schemes by applying them on networks of various sizes. Figure 10 shows the response times under 4 networks (see Table I), whose sizes range from 0.7M edges to 15M edges. We see that MPR is the most scalable method compared with other schemes. The same conclusion can be drawn on throughput.

Finally, we conclude this section by a brief note on MPR's memory consumption. Among all of our experiment settings, the largest memory consumption is 39GB (for the largest network USA(W)). For a city-scale network, the largest consumption is 15GB (BJ). These are well within the capacity of a commodity multicore server.

VI. Conclusion

In this paper we conducted a comprehensive study on the problem of multiprocessing kNN queries on a multicore machine. We put forward the MPR framework, which offers a partitioning and replication strategy to utilize the multiple cores to serve queries and updates. MPR configures itself to achieve the best performance under a target macro measure by solving analytically an optimization problem. Through extensive experiments, we show that MPR is a highly flexible scheme and that it satisfies three adaptability requirements, namely, system adaptability, workload adaptability, and performance adaptability. It also works very well with existing single-threaded kNN solutions and offers an easy way of multiprocessing kNN computation.

Acknowledgement

Ben Kao was supported by Hong Kong University Grant Council grants 17253616 and 17254016. Reynold Cheng was supported by the Research Grants Council of Hong Kong (RGC Projects HKU 17229116, 106150091, and 17205115) the University of Hong Kong (Projects 104004572, 102009508, and 104004129), and the Innovation and Technology Commission of Hong Kong (ITF project MRP/029/18).

References

[1] https://sites.google.com/view/mpr-tech.
[2] https://www.10101111.com/.
[3] T. Abeywickrama, M. A. Cheema, and D. Taniar. k-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB*, 9(6):492–503, 2016.
[4] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.
[5] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.
[8] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian. ROAD: A new spatial object search framework for road networks. *TKDE*, 24(3):547–560, 2012.
[9] C. Li, Y. Gu, J. Qi, J. He, Q. Deng, and G. Yu. A gpu accelerated update efficient index for knn queries in road networks.
[10] S. Luo, B. Kao, G. Li, J. Hu, R. Cheng, and Y. Zheng. TOAIN: a throughput optimizing adaptive index for answering dynamic k nn queries on road networks. *PVLDB*, 11(5):594–606, 2018.
[11] S. Luo, Y. Luo, S. Zhou, G. Cong, and J. Guan. Disks: a system for distributed spatial group keyword search on road networks. *Proceedings of the VLDB Endowment*, 5(12):1966–1969, 2012.
[12] S. Luo, Y. Luo, S. Zhou, G. Cong, J. Guan, and Z. Yong. Distributed spatial keyword querying on road networks. In *EDBT*, pages 235–246, 2014.
[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
[14] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *PVLDB*, pages 802–813, 2003.
[15] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.
[16] B. Shen, Y. Zhao, G. Li, Q. Y. Zheng, Weimin, B. Yuan, and Y. Rao. V-tree: Efficient knn search on moving objects with road-network constraints. In *ICDE*, pages 871–882, 2016.
[17] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *TKDE*, 27(8):2175–2189, 2015.