Piccolo: Building Fast, Distributed Programs with Partitioned Tables

Russell Power
New York University

Jinyang Li
New York University

Abstract
Many applications can see massive speedups by distribut-
ing their computation across multiple machines. How-
ever, as the number of machines increases, so does the
difficulty of writing efficient programs - users must
tackle the problem of minimizing communication and
synchronization performed between hosts while also tak-
ing care to be robust against machine failures.

This paper presents Piccolo, a data-centric program-
ming model in which users organize computation into a
set of application kernels that share mutable distributed
in-memory state using a partitioned key-value table in-
terface. Piccolo eliminates fine-grained application-level
synchronization using user-defined accumulation func-
tions to combine concurrent updates on the same table
entry. By allowing programmers to specify simple lo-
cality policies, Piccolo’s run-time can execute a kernel
where its needed data is stored, resulting in excellent par-
allel performance.

Using Piccolo, we have implemented applications for
several problem domains, including the PageRank al-
gorithm, k-means clustering, n-body simulation, matrix
multiplication, and a distributed crawler. Experiments
using 100 EC2 instances and our own 12 machine cluster
show Piccolo to be significantly faster then existing data
flow models for many problems, while providing simi-
lar fault-tolerance guarantees and a convenient program-
ing interface.

1 Introduction
There is a rising demand for parallel applications that can
run on clusters of hundreds or thousands of commodity
machines. Many of these applications are CPU-intensive
and often require a large amount of memory - exam-
pies vary from large matrix eigenvalue problems such as
PageRank, to machine learning problems (k-means clus-
tering, SVM learning) to physical simulations. By aggre-
gating the CPU and memory resources of many physical
machines, a computation that takes hours to finish on a
single machine can be done in a few minutes. Further-
more, the size of the problem that can be solved within a
reasonable amount of time also drastically increases.

It is difficult to write parallel programs that run across
multiple machines. As the entire computation is divided
among multiple threads running on different machines,
one needs to coordinate these threads and share inter-
mediate results among them. For example, to compute
the PageRank score of one web page l, a thread needs
to access the PageRank scores of l’s “neighboring” web
pages, which may reside in the memory of threads run-
ning on different machines. For many scientists, the de
facto way of programming such an application today is to
use a message passing interface such as MPI [16] to ex-
plicitly exchange messages among application threads.
For many users, the communication-centric model pro-
vided by message-passing is too low-level an abstraction
- they fundamentally care about data and processing data,
as opposed to the whereabouts of data and how to get to
it.

Data-centric programming models, in which users are
presented with a simplified interface to access data but
no explicit communication mechanism, have proven a
much more convenient mechanism for expressing many
computations. As a result, MapReduce [13] and other
related systems [22, 42, 39, 30, 33] have become im-
mensely popular for developing parallel programs for
a cluster of machines. MapReduce and its relatives are
mostly designed to operate on persistent data streamed
from/to disks. As a result of this emphasis, their per-
formance on in-memory applications suffers. Distributed
shared memory [24, 27, 4, 12] and tuple spaces [9] allow
application threads to share distributed in-memory state
easily. However, their simple memory (or tuple) model
makes it difficult for programmers to optimize for good
application performance in a distributed environment.

This paper presents Piccolo, a data-centric program-
ing model for writing parallel applications that manip-
ulate large amounts of in-memory state across memory
machines. In Piccolo, programmers organize the computation around a series of application kernel functions, where each kernel is launched as multiple instances concurrently executing on many compute nodes. Kernel instances share intermediate state using a key-value based in-memory table whose entries are distributed across the memory of different compute nodes. Kernel instances share state exclusively via tables using `get` and `put` primitives. The underlying Piccolo run-time sends messages to read and modify table entries stored in the memory of remote nodes.

Piccolo borrows ideas from existing data-centric systems to enable a simple programming model with a high-performance distributed implementation. Piccolo enforces atomic operations on individual key-value pairs (as in tuple spaces [9]) and user-defined accumulation functions to automatically combine concurrent updates on the same key (similar to reduce functions in MapReduce [13]). The combination of these two techniques eliminates the need for fine-grained application-level synchronization for most applications. Additionally Piccolo exposes primitives for partitioning tables according to user specified functions. Users control how keys are assigned to partitions by defining a partitioning function [13] and the underlying run-time ensures that all entries in the same partition reside in the memory of one machine. By organizing computation as kernel instances and shared-state as user-controlled table partitions, programmers can specify simple locality policies to execute a kernel instance where its needed data is stored, resulting in dramatically better performance than possible with existing DSM systems.

We have built a run-time system to support the programming model of Piccolo. The run-time consists of one master (for coordination) and several worker processes (for storing in-memory table partitions and executing kernels). There are two challenges in building such a run-time: one is to schedule kernel executions in a load-balanced fashion across machines and another is to cope with machine failures. Our scheduler uses a simple work stealing heuristic to dynamically balance load among workers: if a worker finishes all its scheduled kernels, it steals a yet-to-be-started kernel instance from another worker. Furthermore, workers migrate table partitions among themselves as they steal work to adhere to user-specified locality policies. Piccolo provides a checkpoint/restore mechanism to recover from machine failures. It uses the Chandy-Lamport snapshot algorithm [11] to periodically generate a consistent snapshots of the execution state without pausing active computations. Upon machine failure, Piccolo recovers by re-starting the computation from its latest snapshot state.

Experiments have shown that Piccolo is fast and provides excellent scaling for many applications. The performance of PageRank and \( k \)-means on Piccolo is \( 25 \times \) and \( 5 \times \) faster than that of Hadoop. Computing a PageRank iteration on a 1B page web graph takes only 70 seconds on 100 EC2 instances. To illustrate the flexibility of Piccolo’s programming model, we also developed a fully functional distributed web crawler, which easily saturates our 100 Mbps internet uplink bandwidth by running on 12 machines (32 total cores).

The source code for the current Piccolo prototype, along with implementations of all applications, is available at piccolo.news.cs.nyu.edu.

The rest of the paper is organized as follows. Section 2 provides a description of the Piccolo programming model, followed by the design of Piccolo’s run-time (Section 3). We describe the set of applications we constructed using Piccolo in Section 4 Section 5 discusses our prototype implementation. We show Piccolo’s performance evaluation in Section 6 and present related work in Section 7.

2 Programming Model

Piccolo’s programming environment is exposed as a library to existing languages (our current implementation supports C++ and Python) and requires no change to underlying hardware, OS, or compiler. This section presents Piccolo’s programming model in terms of the application program structure (§2.1), the table interface and semantics (§2.2), user-specified locality policy (§2.3), and checkpoint/restore mechanism (§2.4). We conclude this section by showing how to create a concrete application, an implementation of the the PageRank algorithm, on top of Piccolo.

2.1 Program structure

Application programs written for Piccolo consist of `control` functions which are executed on a single machine, and `kernel` functions which are executed concurrently on many machines. Control functions create shared tables, launch multiple instances of a kernel function, and perform global synchronization. Kernel functions consist of sequential code which read from and write to tables to share state among multiple instances. By default, control functions execute in a single thread and a single thread is created for each kernel function. However, the programmer is free to create additional application threads in control or kernel functions as needed.

Kernel invocation: The programmer uses the `Run` function to launch a specified number \( m \) of kernel instances executing the desired kernel function on different machines. Each kernel instance is identified with an integer from 0 to \( m - 1 \) and it can retrieve its id using the `my_instance()` function.

Kernel synchronization: Currently, Piccolo does not support pair-wise synchronization among concur-
rent kernel instances. Rather, synchronization is accomplished via global barriers triggered by the control function. Upon returning from a barrier, all previous kernel invocations are guaranteed to have finished. We found that barriers are sufficient because the shared table interface with user-defined accumulation makes most locking operations unnecessary. This overall application structure, where control functions launch kernels across one or more global barriers, is reminiscent of a CUDA program [29] which also explicitly eschews support for pairwise thread synchronizations.

2.2 Table interface and semantics

Kernel instances organize the computation and share intermediate state through a key-value table. Table entries are spread across all nodes and each key-value pair resides in the memory of a single node. However, the table interface provides a uniform access model whether an underlying key-value pair is stored locally or on another machine.

Figure 1 shows the basic table API. The interface includes standard operations such as set, contains, put as well as Piccolo-specific functions like update, flush, get_iterator. The key and value types associated with a table can be arbitrary user-declared serializable types.

**User-defined accumulation:** Multiple kernel instances can issue concurrent updates to the same key. Users may desire different ways to automatically merge these updates, such as by applying a reduction function or simply collecting the updates in a set. In Piccolo, programmers associate a user-defined accumulator with each table. Piccolo executes the accumulator function to automatically combine concurrent updates on the same key. If programmers expect results to be independent from the ordering of updates, the accumulator must implement a commutative and associative function [41].

Piccolo provides a set of standard accumulators such as summation, multiplication and min/max. To define one’s own accumulator, the user specifies four functions: Initialize to initialize an accumulator for a newly created key, Accumulate to incorporate the effect of a single update operation, Merge to combine the contents of multiple accumulators on the same key, and View to present the current accumulator’s state as the table’s value type. This way of specifying an accumulator is similar to that created for DryadLINQ [41], with the exception of the additional View function. This function is necessary for Piccolo, as gets may be executed at any time and should reflect all updates accumulated so far.

**User-controlled Table Partitioning:** Piccolo uses a user-specified partition function [13] to divide the key-space into multiple partitions. Users can specify the number of partitions a table should have when creating it. The $p$ partitions of a table are named with integers $0...p−1$.

![Figure 1: Table Interface](image)

The user can scan all entries currently belonging to a partition using the get_iterator function (Figure 1). Piccolo does not reveal to the user which node stores a table partition, but simply guarantees that all table entries in a given partition are stored on the same machine. As we will see later, table partitioning is a key primitive for expressing user programs’ locality preferences.

**Table Semantics:** All table operations involving a single key-value pair are atomic from the application’s perspective. Write operations (e.g. update, put) destined for another machine can be buffered to avoid blocking applications. Scanning entries using an iterator is relatively efficient: Piccolo’s run-time pre-fetches remote table entries beyond the current iterator index.

Piccolo provides the following consistency guarantees, similar to those offered by release consistency [18, 25].

- All operations issued by a single kernel instance on the same key are applied in their issuing order. Operations issued by different kernel instances on the same key are applied in some total order [26].
- Upon a successful flush, all buffered writes done by the caller’s kernel instance will have been committed to their respective remote locations, and will be reflected in the response to subsequent gets by any kernel instance.
- Upon the completion of a global barrier by a control function, all kernel instances will have been completed and all their writes will have been applied.

2.3 Expressing locality preferences

Communicating with a different machine over the local network takes at least a few microseconds [31], which is almost two orders of magnitude slower than local memory accesses. Hence, frequent remote gets lead to abysmal application performance as kernels waste CPU
cycles waiting for a table entry to be fetched from another machine. The key to achieving good performance is to execute a kernel instance on the machine which stores most of its data. By organizing the computation as kernels and shared state as partitioned tables, Piccolo provides a simple way for programmers to express locality policies. Such policies enable the underlying Piccolo run-time to execute a kernel instance at a machine that stores its needed data, thus minimizing remote reads.

Piccolo supports two kinds of locality policies: (1) co-locate a kernel execution with some table partition, and (2) co-locate partitions of different tables. Recall the Run function takes as arguments a kernel function and number of desired instances to launch. Additionally, a programmer can specify a table argument to express their preference for co-locating the kernel execution with that table. The run-time honors this preference by executing the i-th kernel instance on the node storing the i-th partition of the specified table. If the number of table partitions (p) is greater than the number of kernel instances (m), the i-th instance is co-located with a subset of partitions i, i + m, i + 2m... Co-locating partitions of different tables is simple: Piccolo provides the GroupTables(T1, T2, ...) function for a user to express their preference to group the specified tables such that the i-th partition of T1, T2, ..., are stored on the same machine. For kernels that frequently read from more than one table, programmers can avoid remote reads by grouping multiple tables and co-locating kernel execution with one of the tables.

2.4 User-assisted Checkpoint and Restore

Piccolo handles machine failures using the checkpoint/restore mechanism. We decided not to support transparent checkpointing but instead require programmers to explicitly decide when and what to checkpoint. This design incurs additional programmer effort but is more efficient as it allows Piccolo to save only state needed for the program’s restoration.

Based on our experience writing applications on Piccolo, we arrived at two checkpointing APIs: one synchronous checkpoint (CpBarrier) and one asynchronous checkpoint (CpPeriodic). Both functions are invoked by some control function. Iterative applications (e.g. PageRank) launch kernels in multiple rounds separated by global barriers and can use synchronous checkpoints to save intermediate state every few rounds. On the other hand, applications with long running kernels (e.g. distributed crawler) use asynchronous checkpoints and co-locate partitions of different tables. Recall the other hand, applications with long running kernels co-locate a kernel execution with some table partition, stores its needed data, thus minimizing remote reads.

# source values for the next iteration
swap(curr, next)

Figure 2: PageRank Implementation

def PageRank(Config conf):
    graph = Table(pageid, [pageid]).init("/dfs/graph")
    curr = Table(pageid, double).init(
        graph.numPartitions(),
        SumAccumulator, SitePartitioner)
    next = Table(pageid, double).init(
        graph.numPartitions(),
        SumAccumulator, SitePartitioner)
    GroupTables(curr, next, graph)
    if conf.restore():
        last_iter = curr.restore_from_checkpoint()
    else:
        last_iter = 0
    # run 50 iterations
    for iteration in range(last_iter, 50):
        Run(PKernel, instances=curr_pr.numPartitions(),
            locality=LOC_REQUIRED(curr),
            args=(curr, next, graph))
        # checkpoint every 5 iterations, storing the
        # current iteration alongside checkpoint data
        if i % 5 == 0:
            CpBarrier(tables=curr, additional=[iteration])
        else:
            Barrier()
    # the values accumulated into 'next' become the
    swap(curr, next)

def PRKernel(Table(pageid, double) curr, Table(pageid, double) next, Table(pageid, [pageid]) graph_partition):
    # current iteration alongside checkpoint data
    if i % 5 == 0:
        CpBarrier(tables=curr, additional=[iteration])
    else:
        Barrier()
    # the values accumulated into 'next' become the
    swap(curr, next)

def PRKernel(Table(pageid, double) curr, Table(pageid, double) next, Table(pageid, [pageid]) graph_partition):
    # current iteration alongside checkpoint data
    if i % 5 == 0:
        CpBarrier(tables=curr, additional=[iteration])
    else:
        Barrier()
    # the values accumulated into 'next' become the
    swap(curr, next)

PageRank takes as arguments a list of tables, a time interval for periodic checkpointing, and a kernel callback function used for checkpointing optional user data. Piccolo invokes kernel checkpoint callbacks immediately after the completion of a get or update operation. Piccolo ensures that the checkpointing of user state and tables appear atomic. To recover from a machine failure, Piccolo simply invokes a user-specified restore function to load the latest checkpointed table and user state.

2.5 Putting it together: PageRank

We use PageRank as a concrete example on how applications are constructed using Piccolo. The PageRank computation [8] takes as input a sparse web graph and computes a score for each page. The computation proceeds in multiple iterations: page i’s rank value in the k-th iteration (p_i^{(k)}) is the sum of the normalized ranks of its incoming neighbors in the previous iteration: p_i^{(k)} =

def PRKernel(Table(pageid, double) curr, Table(pageid, double) next, Table(pageid, [pageid]) graph_partition):
    # current iteration alongside checkpoint data
    if i % 5 == 0:
        CpBarrier(tables=curr, additional=[iteration])
    else:
        Barrier()
    # the values accumulated into 'next' become the
    swap(curr, next)
The complete PageRank implementation in Piccolo is shown in Figure 2. The input link graph is represented as a set of outgoing links, src \( \rightarrow \) dst, for each page (src). The graph is loaded into the shared in-memory table (graph) from a distributed file system. For a link graph too large to fit in memory, Piccolo also supports a DiskTable interface to stream link table partitions from disk using the iterator API. The intermediate rank values are kept in two tables: curr for the ranks to be read in the current iteration, next for the ranks to be written. The control function (PageRank) iteratively launches \( p \) PRKernel kernel instances where \( p \) is the number of table partitions in graph (which is identical to that of curr and next). The kernel instance \( i \) scans all pages in the \( i \)-th partition of graph, reads their ranks in curr, and generates updates for next.

To cope with concurrent updates to next, the program associates the Sum accumulation function with next, which correctly combines updates as desired by the PageRank computation. The overall computation proceeds in rounds using a global barrier between PRKernel invocations.

To optimize for locality, the program groups tables graph, curr, next together and expresses preference for co-locating PRKernel executions with the curr table. As a result, none of the kernel instances need to perform any remote reads. As a bonus, the program uses the partition function, ParByDomain, to assign the URLs in the same domain to the same partition. As more links exist among pages in the same domain, such partitioning also reduces the number of remote updates.

The checkpoint/restore mechanism is easy to use: the program performs synchronous checkpoint to save the next table every five iterations and loads the latest checkpointed table into next upon recovery from a failure.

3 System Design

This section describes the run-time design for executing Piccolo programs on a large collection of machines interconnected with high-speed Ethernet.

3.1 Overview

Piccolo’s execution environment consists of one master process and \( N \) worker processes. Piccolo relies on the single master to simplify global decision making such as determining the set of active workers, assigning table partitions to workers etc. Figure 3 illustrates the overall interactions among workers and the master when executing a Piccolo program. As Figure 3 shows, each worker is responsible for storing one or more table partitions in its memory and handles operations involving entries in partitions in its ownership. Worker processes execute kernel instances and the master process executes the user control thread by itself. Additionally, the master decides how table partitions are assigned to workers and schedules kernel instances for execution on workers. Having a single master does not introduce a performance bottleneck: the master informs all workers of the current partition assignment so that workers need not consult the master to perform the performance-critical get/put/update operations, as shown in Figure 3.

The master begins the execution of a Piccolo program by invoking the entry function in the control thread. Upon each table creation API call, the master decides upon a partition assignment. The master informs all workers of the partition assignment and each worker initializes its set of partitions accordingly. Upon each Run API call to execute \( m \) kernel instances, the master prepares \( m \) tasks, one for each kernel instance. The master schedules these tasks for execution on workers based on user’s locality preference. Each worker runs a single kernel instance at a time, resulting in at most \( N \) concurrent kernel instances. Each worker notifies the master upon task completion and the master assigns additional tasks if they are available. Upon encountering a global barrier, the master blocks the control thread until all active tasks are completed.

During kernel execution, a worker buffers update operations destined for remote workers, combines them using user-defined accumulators and flushes them to remote workers after a short timeout. To handle a get or put operation, the worker flushes accumulated updates on the same key before sending the operation to the remote worker. Each owner applies operations (including accumulated updates) in their received order. Piccolo does not perform caching but supports a limited form of pre-fetching: after each get_iterator API call, the
worker pre-fetches a portion of table entries beyond the current iterator value.

Two main challenges arise in the above basic design. First, how to assign tasks in a load-balanced fashion so as to reduce the overall wait time on global barriers? This is particularly important for iterative applications that incur a global barrier at each iteration of the computation. The second challenge is to perform efficient checkpointing and restoration of table state. In the rest of this Section, we detail how Piccolo address both challenges.

### 3.2 Load-balanced Task Scheduling

The amount of time it takes a worker to finish executing a kernel instance can vary greatly, resulting in unbalanced load across workers. Two factors can cause some workers to run slower than others: heterogeneous hardware configurations and variable-sized computation inputs. Piccolo’s scheduling freedom is limited by two constraints: First, no running tasks should be killed. As a running kernel instance modifies global table state, re-executing a terminated kernel instance requires performing an expensive restore operation from a saved checkpoint. Therefore, once a kernel instance is starts, it is better to let the task complete than terminating it halfway for re-scheduling. By contrast, MapReduce systems do not have this constraint [23] as reducers do not start aggregation until all mappers are finished. The second constraint comes from the need to honor user locality preference. Specifically, if a kernel instance is to be moved from one worker to another, its co-located table partitions must also be transferred across those workers.

Scheduling without load-balancing works as follows. At table creation time, the master assigns table partitions to all workers using a simple round-robin assignment for empty memory tables. For tables loaded from a distributed file, the master chooses an assignment that minimizes inter-rack transfer while keeping the number of partitions roughly balanced among workers. The master schedules \( m \) tasks according to locality preference, namely, assign task \( i \) to a worker storing partition \( i \).

**Load-balance via work stealing**: Piccolo performs a simple form of load-balancing: the master observes the progress of different workers and instructs a worker \( w_{idle} \) that has finished all its assignments to steal a not-yet-started task \( i \) from the worker \( w_{busy} \) with the most remaining tasks. Since scheduling can be viewed as a variant of the classic NP-hard knapsack problem, we adopt the greedy heuristic of packing the biggest item first, i.e. scheduling the largest task first. To implement this heuristic, the master collects partition size information from all workers at table loading time or at each global barrier. Using the locality preference, we approximate the input size of a task using the size of a kernel instance’s co-located table partition. The master instructs each worker to execute its assigned tasks in decreasing order of estimated task sizes. Additionally, during work stealing, \( w_{idle} \) chooses the biggest task among \( w_{busy} \)’s remaining tasks.

**Table partition migration**: Because of user locality preference, the corresponding table partition \( i \) for the stolen task needs to be transferred from \( w_{busy} \) to \( w_{idle} \) before execution starts. Since migration occurs while other active tasks are sending operations to partition \( i \), Piccolo must take care not to lose, re-order, or duplicate operations from any worker on a given key during migration in order to preserve table semantics.

The master coordinates the process of migrating partition \( i \) from \( w \) to \( w' \). The migration proceeds in three phases. In the first phase, the master sends message \( M_1 \) to \( w' \) to inform its new ownership of \( i \). In the second phase, the master sends message \( M_2 \) to all workers to adjust their partition assignment of \( i \) from \( w \) to \( w' \). Upon the receipt of \( M_2 \), every worker begins to send table operations destined for partition \( i \) to \( w' \). Worker \( w' \) simply buffers these operations without applying them. Additionally, upon receiving \( M_2 \), the previous owner \( w \) freezes the current partition \( i \) and buffers all future received operations should they arrive after the table is frozen. Subsequently, the new owner \( w' \) transfers the frozen state of partition \( i \) from \( w \). Each worker sends an acknowledgment to the master after it has finished the second phase. After the master has received acknowledgments from all workers, it proceeds to the third phase and sends \( M_3 \) to \( w' \) to complete migration. Upon receiving \( M_3 \), \( w' \) transfers buffered operations from \( w \) and applies them to the frozen partition state. After buffered operations from all workers are applied, \( w' \) completes the migration by unfreezing partition \( i \).

As can be seen, the migration process does not block any update operations and thus incurs little latency overhead for most kernels. The normal checkpoint/recovery mechanism is used to cope with faults that might occur the migration.

### 3.3 Fault Tolerance

Piccolo relies on user-assisted checkpointing and restoration to cope with both master and worker failures during a program run. The Piccolo run-time saves a checkpoint of program state (including tables and other user-data) on a distributed file system and restores from the latest completed checkpoint to recover from a failure.

**Checkpoint**: Piccolo supports synchronous (CBarrier) and asynchronous (CPertodic) checkpointing. Checkpointing must be consistent and should incur low-overhead. To reduce overhead, Piccolo needs to overlap kernel computation with checkpoint saving. To ensure consistency, Piccolo must determine a global snapshot of the system in the face of concurrently
running kernels and the control thread.

We use the Chandy-Lamport (CL) distributed snapshot algorithm [11] to perform checkpointing. To save a CL snapshot, each process records its own state and two processes incident on a communication channel cooperate to save the channel state. In Piccolo, channel state can be efficiently captured using only table modification messages because kernels communicate with each other exclusively via tables. We describe how Piccolo applies the CL algorithm to perform both types of checkpointing below.

To begin a checkpoint, the master chooses a new checkpoint epoch number \( E \) based on an integer counter and sends the start checkpoint message \( \text{Start}_E \) to all workers. Upon receiving the start message, worker \( w \) immediately “freezes” the current state of its responsible tables partitions and buffers future table operations. Once the frozen table partitions are serialized to stable storage, \( w \) sends the marker message \( \text{M}_E,w \) to all other workers. Worker \( w \) then enters a logging state in which it logs all buffered operations to a replay file (in addition to applying them). Once \( w \) has received markers from all other works \( \forall w', \forall w' \neq w \), it flushes the replay logs and sends \( \text{Fin}_E,w \) to the master. The master considers the checkpoint done once it has received \( \text{Fin}_E,w \) from all workers.

For asynchronous checkpoints, the master initiates checkpoints periodically based on a timer. To record any user-data consistently with table state, each worker atomically freezes table state and invokes user checkpoint callback functions upon receiving the start checkpoint message. Synchronous checkpoints provide the semantics that recorded table state and user-data is equivalent to those immediately after the global barrier. Therefore, for synchronous checkpointing, each worker may start after it has completed all its assigned tasks before sending the checkpoint marker \( \text{M}_E,w \) to all other workers. Additionally, the master should save any optional user-data only after it has received \( \text{Fin}_E,w \) from all workers. There is a trade-off in deciding when to start a synchronous checkpoint. If the master starts the checkpoint too early, e.g. while workers still have many remaining tasks, replay files become unnecessarily large. On the other hand, if the master delays checkpointing until all workers have finished, it misses opportunities to overlap kernel computation with checkpointing. Piccolo uses a heuristic to balance this trade-off: the master begins a synchronous checkpoint as soon as one of the workers has finished all its assign tasks.

To simplify the design, the master does not initiate checkpointing while there is active table migration and vice-versa.

**Restore:** Upon detecting any worker failure, the master restarts the computation without the failed node by restoring from the last completed checkpointed. If the master has failed, users need to designate a new replacement master for recovery, as Piccolo does not checkpoint the internal state of the master: the replacement master is free to choose a different partition assignment and task schedule during restoration.

## 4 More Applications

In addition to PageRank, we have implemented four other applications: a distributed web crawler, k-means, n-body, matrix multiplication. This section describes how implementations optimize for locality using Piccolo’s programming model.

### 4.1 Distributed Web Crawler

Apart from iterative parallel computations, Piccolo can be used by applications to distribute and coordinate fine-grained tasks among many machines. To demonstrate this usage we implemented a distributed web crawler using Piccolo. The basic crawler operation is simple: starting from a few starting URLs, the crawler repeatedly downloads a page which it parses to discover new URLs to fetch. A practical crawler must also satisfy other important constraints: (1) honor the robots.txt file of each web site, (2) refrain from overwhelming a site by capturing fetches to a site at a fixed rate, and (3) avoid repeated fetches of the same URL.

Our implementation uses three co-located tables:

- The URL table stores the crawling state ToFetch, Fetching, Blacklisted, Done for each URL. The crawler fetches a page \( p \) in ToFetch state and sets the page state to Fetching. After the crawler has finished parsing \( p \) and extracting its outgoing links, it further sets \( p \)'s state to Done.
- The Politeness table tracks the last time a page was downloaded for each site.
- The Robots table stores the processed robots file for each site.

A single node crawler is often limited by available memory, as crawled URLs must reside in memory in order for the crawler to maintain reasonable performance. Using Piccolo, a distributed crawler can effectively aggregate the memory of many machines to crawl large numbers of pages efficiently.

Our crawler is implemented in Python in order to utilize the comprehensive set Python web-related libraries. The crawler spawns \( m \) kernel instances, one for each machine. A simplified crawler kernel implementation is shown in Figure 4.1 (omitting details for processing robots.txt files). Each kernel scans its local URL table partitions to find ToFetch URLs and processes them using a pool of helper threads. As all three tables are
Apart from a distributed crawler, we have also implemented a number of conventional applications: in machine learning (k-means), physical simulation (n-body) and numerical linear algebra (matrix multiplication). Below, we discuss these implementations and how they generalize to other applications which share similar program structures.

**k-means.** The k-means algorithm is an iterative algorithm to group n data points into k clusters in a multi-dimensional space to minimize the “distance” between points and their assigned clusters [5]. k-means clustering can be performed on hundreds of millions of data points (e.g. a large document or picture collection) and hundreds or thousands of clusters (e.g. each topic of interest).

$k$-means represents a program structure found in many machine learning algorithms, wherein a small model is being optimized against a large state space. In the case of k-means, the positions of a relatively small number of centroids are being optimized by feedback from a large number of points. Piccolo is well-suited for this structure by co-locating kernel execution with training points, kernel instances generate updates for the refined model partitioned across many machines. Many popular machine learning and clustering algorithms, such as expectation maximization and neural networks [6], have implementations that exhibit this structure.

**n-body.** This application models the dynamics of a set of particles. The n-body problem consists of simulating many particles over discrete time-steps - during each time-step the velocity and position of each particle is updated based on its current velocity and the force interaction of other particles. We implemented an n-body simulation intended for short distances [36], where particles further than a threshold distance (r) apart are assumed to have no effect on each other. To improve the performance of our implementation, we divided space into cubes with edge length r; each particle can only be affected by forces from other particles in its own and surrounding boxes.

When compared with k-means, physics simulations [15] represent a different structure in which the model (particle positions) being updated is as large as the data points themselves (particles). To parallelize this structure, the model is divided into blocks which are further partitioned among machines. This structure works well on Piccolo because the computation for each subdivision of the model requires only a few random remote gets and generates accumulatable updates. For example, the belief-propagation algorithm used in computer vision only requires interactions among neighboring pixels. The Barnes-Hut n-body approximation requires $O(\log n)$ remote reads to generate accumulatable updates for particles in a hierarchy of grids.

**Matrix multiplication.** Computing $C = AB$ where $A$ and $B$ are two large matrices is a common primitive in numerical linear algebra. The input and output matrices are divided into blocks of size $b \times b$ and stored in three tables. The kernel is co-located with table $C$. Each instance computes $C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$. As a small optimization, we co-locate tables $A, B$ with $C$.

Matrix multiplication has a similar structure to many other common dense matrix applications such as the LU decomposition [34] and dense Cholesky factorization which divide matrices into a few large blocks. Compared to n-body, each kernel instance incurs many more remote reads (proportional to the square root of the number of machines).

**PageRank.** The PageRank algorithm computes the eigenvalues of a large sparse matrix, i.e. the web link
graph. Its structure is similar to other iterative methods for solving linear systems of equations such as the iterative conjugate gradient method [34]. Like PageRank, each CG kernel does not perform remote reads when co-located with inputs but generates a large number (proportional to the matrix dimension) of partial sums.

5 Implementation

Piccolo has been implemented in C++. We provide both C++ and Python APIs so that Users can write kernel and control functions in either C++ or Python. We use SWIG [3] for constructing a Python interface to Piccolo. Our implementation re-uses a number of existing libraries, such as OpenMPI for communication, Google’s protocol buffers for object serialization, and LZO for compressing on-disk tables.

All the parallel computations (PageRank, k-means, n-body and matrix multiplication) are implemented using the C++ Piccolo API. The distributed crawler is implemented using the Python Piccolo API.

6 Evaluation

We tested the performance of Piccolo on the applications described in Section 4. We constructed PageRank and k-means implementations in Hadoop to compare our performance against that achieved in a bulk data-flow model.

The highlights of our results are:

- Piccolo is fast. PageRank and k-means are 5 and 25× faster than those on Hadoop. When compared against the results published for DryadLinq [42], in which a PageRank iteration on a 900M page graph were performed in 69 seconds, Piccolo finishes an iteration for a 1B page graph in 70 seconds on EC2, while using 1/5 the number of CPU cores.

- Piccolo scales well. For all applications evaluated, increasing the number of workers shows a nearly linear reduction in the computation time. Our 100-instance EC2 experiments on PageRank also demonstrates excellent scaling.

- Piccolo can help a non-convention application like the crawler achieve good parallel performance. Our crawler can download and parse web pages at 100Mbps, limited only by the Internet bandwidth of our cluster.

6.1 Test Setup

Most experiments were performed using our small local cluster of 12 machines (64 total CPU cores): 6 of the machines have 1 quad-core Intel Xeon X3360 (2.83GHz) processors with 4GB memory, the other 6 machines have 2 quad-core Xeon E5520 (2.27GHz) processors with 8GB memory. All machines are connected via a commodity gigabit ethernet switch. Our EC2 experiments involve 100 “large instances” each with 7.5GB memory and 2 “virtual cores” where each virtual core is equivalent to a 2007-era single core 2.5GHz Intel Xeon processor. In all experiments, we pinned each Piccolo worker to use a single physical or virtual core.

For our scaling experiments, we vary the input size of different applications. Table 5 shows the default and maximum input size used for each application. We generate the web link graph for PageRank based on the statistics of a web graph of 100M crawled UK pages as part of the Web Graph Project[7]. Specifically, we extract the distributions of the number of pages in each site and the ratio of intra/inter-site links and generate a web graph of any size by sampling from the site size distribution until the desired number of pages is reached; outgoing links are then generated for each page in a site based on the distribution of the ratio of intra/inter-site links. For other applications, we use randomly generated inputs.

6.2 Scaling Performance

We first evaluate how the total application running time decreases with increasing number of workers (N). Figure 6 shows application speedup as N ranges from 8 to 64 for the default input size in Table 5. All applications are CPU-bound and exhibit linear speedup with N. Ideally, all applications (except for PageRank) have perfectly balanced table partitions and should achieve ideal speedup. To have a reasonable runtime at N = 8, we choose a relatively small default input size. As the input is divided among more workers, Piccolo’s overhead is no longer negligible (e.g. PageRank finishes each iteration in 8 seconds with N = 64.), resulting in 20% less than ideal speedup. PageRank’s table partitions are not balanced and work stealing becomes important for its scaling (see § 6.6).

We also evaluate the ability for Piccolo to scale with increasing computation. We adjust the input size to keep the amount of computation per worker fixed with increasing N. For PageRank and k-mean, the input size is increased linearly with N and for matrix multiplication, the edge size increases as \(O(N^{1/3})\). We did not show n-body results because its computation scales somewhere between constant and quadratic depending on the box size, making it difficult to find the right input size to achieve a fixed amount of computation per worker. The

<table>
<thead>
<tr>
<th>Application</th>
<th>Default input size</th>
<th>Maximum input size</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank</td>
<td>100M pages</td>
<td>1B pages</td>
</tr>
<tr>
<td>k-means</td>
<td>25M points, 100 clusters</td>
<td>1B points, 100 clusters</td>
</tr>
<tr>
<td>n-body</td>
<td>100K points</td>
<td>10M points</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>edge size = 2500</td>
<td>edge size = 6000</td>
</tr>
</tbody>
</table>

Figure 5: Application input sizes.
ideal scaling has constant running time as input size increase with $N$. As Figure 7 shows, the achieved scaling for all applications is within 25% of the ideal number.

### 6.3 Comparison with Hadoop

We compared Piccolo’s performance with Hadoop for the PageRank and $k$-means benchmarks. We made extensive efforts to optimize the Hadoop implementations of these applications, including using raw memory comparisons, using primitive types to avoid Java’s boxing and unboxing overhead, optimizing various tuning parameters etc. We ignore the startup time and the time taken between jobs (iterations) when showing Hadoop’s performance numbers.

For the Hadoop implementation of PageRank, as with Piccolo, we partition the input link graph by site. During execution, each map task has locality with the partition of graph it is operating on. Mappers process the joined graph and PageRank score inputs, and use a combiner to aggregate partial results. Our Hadoop $k$-means implementation is highly optimized. Each mapper fetches all 100 centroids from the previous iteration, computes the cluster assignment of each point in its input stream, and uses a local hash map to aggregate the updates for each cluster. As a result, a reducer only needs to aggregate one update from each mapper to generate the new centroid.

Figure 8 shows the running time of Piccolo and Hadoop using the default input size. Piccolo significantly outperforms Hadoop on both benchmarks (25× for PageRank and 5× for $k$-means with $N=64$). The performance difference between Hadoop and Piccolo is smaller for $k$-means because of our optimized $k$-means implementation; the structure of PageRank does not admit a similar optimization.

### 6.4 EC2 Scaling

We investigated how Piccolo scales with a larger number of machines using 100 EC2 instances. Figure 9 shows the scaling of PageRank and $k$-means on EC2 as we increase their input size with $N$. We were somewhat surprised to see that the resulting scaling on EC2 is better than achieved on our small local testbed. Our local testbed’s CPU performance exhibits quite some variability between experimental runs, impacting scaling. After further investigation, we believe the source for such variability is likely due to dynamic CPU frequency scaling.

At $N = 200$, PageRank finishes in 70 seconds for a 1B page link graph. On a similar sized graph (900M pages), our local testbed achieves comparable performance (80 seconds) with many fewer workers ($N = 64$), due to the higher performing cores on our local testbed.

### 6.5 Distributed Crawler

We evaluated our distributed crawler implementation using various numbers of workers. The URL table was initialized with a seed set of 1000 URLs. At the end of a 30 minutes run of the experiment, we measured the number of pages crawled and bytes downloaded. Figure 10 shows the crawler throughput in MBytes/sec as $N$ increases from 1 to 64. The crawler spends most CPU time in the Python code for parsing HTML and URLs so that its throughput scales linearly with $N$.

At $N = 32$,
the crawler downloads at 100Mbps which is the bandwidth limit of our internet uplink. At \( N = 64 \), the crawler throughput decreases: as more workers are competing for the limited uplink bandwidth, there are more timeouts in downloading web pages.

Even capped by the 100Mbps uplink limit, our crawler will be able to retrieve 1 billion web pages over 28 days. We can trade off CPU time for better crawling throughput by changing our implementation to download gzipped pages.

6.6 Work Stealing

We evaluated the usefulness of our work stealing implementation using the PageRank benchmark; evaluating the performance with and without work stealing enabled. PageRank provides a good basis for testing due to the fairly large variance in the distribution of shard sizes present: the largest shard for the 900M page corpus is 5 times the size of the smallest.

The results of the work stealing evaluation are shown in figure 11. The savings affected by work stealing are substantial - the time per iteration is cut nearly in half when work stealing is enabled. Upon examination, the reason for the disparity is, as expected, the imbalance in the input partition sizes - when run without work stealing, the computation is forced for long periods waiting for the workers assigned more data to catch up. A smaller component of the difference is associated with our somewhat heterogenous cluster environment - the frequency of our processors differs by up to 25%. While the PageRank computation is mostly memory bound, the speed differential between machines can still be discerned.

6.7 Checkpointing

We evaluated the overhead associated with checkpointing using the PageRank computation. Compared to \( k \)-means and \( n \)-body problems, Pagerank has a larger table that needs to be checkpointed, making it a more effective test of the overhead of checkpointing. In our experiment, each worker wrote its checkpointed table partitions to the local disk. Figure 12 shows the relative time of PageRank with checkpointing disabled and enabled. We also show the overhead of the naive implementation of synchronous checkpointing where the master starts checkpointing only after all workers have finished. As the figure shows, overhead of the checkpoint is negligible (2%) and the optimization of starting checkpointing early results in significant reduction of overhead.
7 Related Work

Parallel models for distributed environments

Message-passing systems such as MPI [16] and Parallel Virtual Machine (PVM [38]) have been a popular mechanism for constructing distributed programs for many years. The MPI specification, finalized in the mid-90’s provides a simple API, providing for both point-to-point and broadcast communication between nodes. High performance MPI libraries are available for many supercomputer networking environments, such as InfiniBand and Myrinet. PVM, introduced in 1989, provides much the same messaging mechanism as MPI, while providing additional support for cluster-wide process management.

The complexity of programming for explicit message passing models, which require the user to manage explicit communications themselves drove a wave of research in the area of distributed shared memory (DSM) systems [25, 24, 27, 4]. Most DSM systems aim to provide transparent memory access and programs on top of DSMs often incur many fine-grained application-level synchronizations and remote memory reads. While initially promising, DSM research has fallen off as the ratio of network latency to local CPU performance has widened, making naive remote accesses and synchronization prohibitively expensive.

Parallel Global Address Space (PGAS) extensions [12, 28, 40] try to ameliorate DSM’s latency problems by allowing users to express affinities of portions of shared memory with a particular thread, thereby reducing the frequency of remote memory references. They retain the low level (raw memory) interface common to DSM’s. As a result, applications written for PGAS systems tend to require complicated synchronization logic when operating on non-primitive data-types, or in order to aggregate several values (for instance, computing the sum of a memory location with multiple writers).

Tuple spaces, as seen in coordination languages such as Linda [9] and more recently in the form of JavaSpaces [17], expose users to a global tuple-space accessible from all participating threads. Tuple spaces support atomic primitives for reading and writing tuples as well as simple queries for finding matching tuples. As tuple spaces’ target usage is for loosely-coupled work coordination, they do not support locality of execution nor accumulation operations. As a result, they cannot provide good performance for closely-coupled distributed programs such as PageRank.

There are many programming models for writing parallel programs on a single machine, such for OpenMP [21, 2], CUDA [29], OpenCL [20] etc. These local programming models rely on the availability of low-latency memory access and/or synchronization between threads, features not available in distributed environments.

MapReduce and data flow programming

In recent years, MapReduce has emerged as the de facto programming paradigm for parallel data processing [13]. Its wide-spread popularity has spawned much research interest, ranging from generalizing MapReduce to data flow programming [22] to building high-level query languages such as DryadLINQ [42], Hive [39], Pig [30], Sawzall [33]. FlumeJava [10] provides a set of collection abstractions and parallel execution primitives which are optimized and compiled down to a sequence of MapReduce operations.

At its heart, MapReduce is a batch processing system for analyzing and transforming large quantities of data streamed from disks. Its success stems from restricting programmers to only writing operators that manipulate data streams in exchange for a run-time that can automatically parallelize jobs and cope with failures. Thus, for applications that predominantly manipulate in-memory state, MapReduce and its relatives may be too heavy weight. As MapReduce does not provide direct access to globally shared state, implementing a distributed crawler becomes hard because the crawler needs prompt access to recently discovered URLs.

Distributed data structures

The concept of distributed data structures typically refers to frameworks that provide a flexible and scalable data storage or caching interface. Examples of these include DDS [19], Memcached [32], the recently proposed RamCloud [31], and many key-value stores based on distributed hash tables [1, 14, 37, 35]. These systems do not seek to provide a locality or work distribution model. They are targeted at loosely-coupled distributed applications like a web-serving system and are not designed for more tightly-coupled applications like a distributed crawler or PageRank.

8 Conclusion

In this paper we have presented Piccolo, a flexible system for work distribution and communication across large numbers of general purpose computers. Piccolo combines the features of popular existing programming models to provide a convenient and efficient table based interface for high performance computing.

Our testing using common problems in scientific and distributing computation has shown Piccolo to have excellent performance while providing a comfortable and programming environment familiar to users.

References


