Piccolo: Building Fast, Distributed Programs with Partitioned Tables

Russell Power Jinyang Li

New York University
http://news.cs.nyu.edu/piccolo

Abstract

Piccolo is a new data-centric programming model for writing parallel in-memory applications in data centers. Unlike existing data-flow models, Piccolo allows computation running on different machines to share distributed, mutable state via a key-value table interface. Piccolo enables efficient application implementations. In particular, applications can specify locality policies to exploit the locality of shared state access and Piccolo's run-time automatically resolves write-write conflicts using user-defined accumulation functions.

Using Piccolo, we have implemented applications for several problem domains, including the PageRank algorithm, *k*-means clustering and a distributed crawler. Experiments using 100 Amazon EC2 instances and a 12 machine cluster show Piccolo to be faster than existing data flow models for many problems, while providing similar fault-tolerance guarantees and a convenient programming interface.

1 Introduction

With the increased availability of data centers and cloud platforms, programmers from different problem domains face the task of writing parallel applications that run across many nodes. These application range from machine learning problems (k-means clustering, neural networks training), graph algorithms (PageRank), scientific computation etc. Many of these applications extensively access and mutate shared intermediate state stored in memory.

It is difficult to parallelize in-memory computation across many machines. As the entire computation is divided among multiple threads running on different machines, one needs to coordinate these threads and share intermediate results among them. For example, to compute the PageRank score of web page *p*, a thread needs to access the PageRank scores of *p*'s "neighboring" web pages, which may reside in the memory of threads running on different machines. Traditionally, parallel in-memory applications have been built using message-passing primitives such as MPI [21]. For many users, the communication-centric model provided by message-passing is too low-level an abstraction - they fundamentally care about data and processing data, as opposed to the location of data and how to get to it.

Data-centric programming models [19, 27, 1], in which users are presented with a simplified interface to access data but no explicit communication mechanism, have proven a convenient and popular mechanism for expressing many computations. MapReduce and Dryad [27] provide a data-flow programming model that does not expose any globally shared state. While the data-flow model is ideally suited for bulk-processing of on-disk data, it is not a natural fit for in-memory computation: applications have no online access to intermediate state and often have to emulate shared memory access by joining multiple data streams. Distributed shared memory [29, 32, 7, 17] and tuple spaces [13] allow sharing of distributed in-memory state. 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 programming model for writing parallel in-memory applications across many 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 distributed, mutable state using a set of in-memory tables whose entries reside in the memory of different compute nodes. Kernel instances share state exclusively via the key-value table interface with *get* and *put* primitives. The underlying Piccolo runtime sends messages to read and modify table entries stored in the memory of remote nodes.

By exposing shared global state, the programming model of Piccolo offers several attractive features. First, it allows for natural and efficient implementations for ap-

plications that require sharing of intermediate state such as k-means computation, n-body simulation, PageRank calculation etc. Second, Piccolo enables online applications that require immediate access to modified shared state. For example, a distributed crawler can learn of newly discovered pages quickly as a result of state updates done by ongoing web crawls.

Piccolo borrows ideas from existing data-centric systems to enable efficient application implementations. Piccolo enforces atomic operations on individual key-value pairs and uses user-defined accumulation functions to automatically combine concurrent updates on the same key (similar to reduce functions in MapReduce [19]). The combination of these two techniques eliminates the need for fine-grained application-level synchronization for most applications. Piccolo allows applications to exploit locality of access to shared state. Users control how table entries are partitioned across machines by defining a partitioning function [19]. Based on users' locality policies, the underlying run-time can schedule a kernel instance where its needed table partitions are stored, thereby reducing expensive remote table access.

We have built a run-time system consisting of one master (for coordination) and several worker processes (for storing in-memory table partitions and executing kernels). The run-time uses a simple work stealing heuristic to dynamically balance the load of kernel execution among workers. Piccolo provides a global checkpoint/restore mechanism to recover from machine failures. The run-time uses the Chandy-Lamport snapshot algorithm [15] to periodically generate a consistent snapshots of the execution state without pausing active computations. Upon machine failure, Piccolo recovers by restarting 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 $11\times$ and $4\times$ faster than that of Hadoop. Computing a PageRank iteration for a 1 billion-page web graph takes only 70 seconds on 100 EC2 instances. Our distributed web crawler can easily saturate a 100 Mbps internet uplink when running on 12 machines.

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 OS or compiler. This section describes the programming model in terms of how to structure application programs (§2.1), share intermediate state via key/value tables (§2.2), optimize for locality of access (§2.3), and recover from failures(§2.4). We conclude this section by showing how to implement the PageRank algorithm on top of Piccolo (§2.5).

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 concurrently executing kernel instances. By default, control functions execute in a single thread and a single thread is created for executing each kernel instance. 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 has an identifier $0 \cdots m-1$ which can be retrieved using the `my_instance` function.

Kernel synchronization: The programmer invokes a global barrier from within a control function to wait for the completion of all previously launched kernels. Currently, Piccolo does not support pair-wise synchronization among concurrent kernel instances. We found that global barriers are sufficient because Piccolo's shared table interface makes most fine-grained locking operations unnecessary. This overall application structure, where control functions launch kernels across one or more global barriers, is reminiscent of the CUDA model [36] which also explicitly eschews support for pair-wise thread synchronization.

2.2 Table interface and semantics

Concurrent kernel instances share intermediate state across machine through key-value based in-memory tables. Table entries are spread across all nodes and each key-value pair resides in the memory of a single node. Each table is associated with explicit key and value types which can be arbitrary user-declared serializable types. As Figure 1 shows, the key-value interface provides a uniform access model whether the underlying table en-

```
Table<Key, Value>:
  clear()
  contains(Key)
  get(Key)
  put(Key, Value)

  # updates the existing entry via
  # user-defined accumulation.
  update(Key, Value)

  # Commit any buffered updates/puts
  flush()

  # Return an iterator on a table partition
  get_iterator(Partition)
```

Figure 1: Shared Table Interface

try is stored locally or on another machine. The table APIs include standard operations such as `get`, `put` as well as Piccolo-specific functions like `update`, `flush`, `get_iterator`. Only control functions can create tables; both control and kernel functions can invoke any table operation.

User-defined accumulation: Multiple kernel instances can issue concurrent `update`s to the same key. To resolve such write-write conflict, the programmer can associate a user-defined accumulation function with each table. Piccolo executes the accumulator during run-time to combine concurrent `update`s on the same key. If the programmer expects results to be independent from the ordering of updates, the accumulator must be a commutative and associative function [52].

Piccolo provides a set of standard accumulators such as summation, multiplication and min/max. To define an 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 return the current accumulator state reflecting all `update`s accumulated so far. Accumulator functions have no access to global state except for the corresponding table entry being updated.

User-controlled Table Partitioning: Piccolo uses a user-specified *partition function* [19] to divide the key-space into partitions. Table partitioning is a key primitive for expressing user programs' locality preferences. The programmer specifies the number of partitions (p) when creating a table. The p partitions of a table are named with integers $0...p-1$. Kernel functions can scan all entries in a given table partition using the `get_iterator` function (see Figure 1).

Piccolo does not reveal to the programmer which node stores a table partition, but guarantees that all table entries in a given partition are stored on the same machine. Although the run-time aims to have a load-balanced assignment of table partitions to machines, it is the programmer's responsibility to ensure that the largest table partition fits in the available memory of a single machine. This can usually be achieved by specifying a the number of partitions to be much larger than the number of machines.

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 kernel execution. In the face of buffered remote writes, Piccolo provides the following guarantees:

- 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 [31].
- 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 `get`s by any kernel instance.
- Upon the completion of a global barrier, all kernel instances will have been completed and all their writes will have been applied.

2.3 Expressing locality preferences

While writes to remote table entries can be buffered at the local node, the communication latency involved in fetching remote table entries cannot be effectively hidden. Therefore, the key to achieving good application performance is to minimize remote `get`s by exploiting locality of access. 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 on a machine that stores most of 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. When launching some kernel, the programmer can specify a table argument in the `Run` function to express their preference for co-locating the kernel execution with that table. The programmer usually launches the same number of kernel instances as the number of partitions in the specified table. The run-time schedules the i-th kernel instance to execute on the machine that stores the i-th partition of the specified table. To optimize for kernels that read from more than one table, the programmer uses the `GroupTables(T1,T2,..)` function to co-locate multiple tables. The run-time assigns the i-th partition of T1,T2,...

to be stored on the same machine. As a result, by co-locating kernel execution with one of the tables, the programmer can avoid remote reads for kernels that read from the same partition of multiple tables.

2.4 User-assisted checkpoint and restore

Piccolo handles machine failures via a global checkpoint/restore mechanism. The mechanism currently implemented is not fully automatic - Piccolo saves a consistent global snapshot of all shared table state, but relies on users to save additional information to recover the position of their kernel and control function execution. We believe this design makes a reasonable trade-off. In practice, the programming efforts required for checkpointing user information are relatively small. On the other hand, our design avoids the overhead and complexities involved in automatically checkpointing C/C++ executables.

Based on our experience of writing applications, we arrived at two checkpointing APIs: one synchronous (`CpBarrier`) and one asynchronous (`CpPeriodic`). Both functions are invoked from some control function. Synchronous checkpoints are well-suited for iterative applications (e.g. PageRank) which launch kernels in multiple rounds separated by global barriers and desire to save intermediate state every few rounds. On the other hand, applications with long running kernels (e.g. a distributed crawler) need to use asynchronous checkpoints to save their state periodically.

`CpBarrier` takes as arguments a list of tables and a dictionary of user data to be saved as part of the checkpoint. Typical user data contain the value of some iterator in the control thread. For example in PageRank, the programmer would like to record the number of PageRank iterations computed so far as part of the global checkpoint. `CpBarrier` performs a global barrier and ensures that the checkpointed state is equivalent to the state of execution at the barrier.

`CpPeriodic` takes as arguments a list of tables, a time interval for periodic checkpointing, and a kernel callback function `CheckpointCallback`. This callback is invoked for all active kernels on a node immediately after that node has checkpointed the state for its assigned table partitions. The callback function provides a way for the programmer to save the necessary data required to restore running kernel instances. Oftentimes this is the position of an iterator over the partition that is being processed by a kernel instance. When restoring, Piccolo reloads the table state on all nodes, and invokes kernel instances with the dictionary saved during the checkpoint.

2.5 Putting it together: PageRank

As a concrete example, we show how to implement PageRank using Piccolo. The PageRank algorithm [11] takes as input a sparse web graph and computes a rank value 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, i.e. $p_i^{(k)} = \sum_{\forall j \in In_i} \frac{p_j^{(k-1)}}{|Out_j|}$, where Out_j denotes page j's outgoing neighbors.

```
tuple PageID(site, page)
const PropagationFactor = 0.85

def PRKernel(Table(PageID,double) curr,
             Table(PageID,double) next,
             Table(PageID,[PageID]) graph_partition):
  for page, outlinks in
      graph.get_iterator(my_instance()):
    rank = curr[page]
    update = PropagationFactor * rank / len(outlinks)
    for target in outlinks:
      next.update(target, update)

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 i in range(last_iter, 50):
    Run(PRKernel,
        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,
                {iteration=i})
    else: Barrier()

    # the values accumulated into 'next' become the
    # source values for the next iteration
    swap(curr,next)
```

Figure 2: PageRank Implementation

The complete PageRank implementation in Piccolo is shown in Figure 2. The input web graph is represented as a set of outgoing links, $page \rightarrow target$, for each page. The graph is loaded into the shared in-memory table (`graph`) from a distributed file system. For link graphs too large to fit in memory, Piccolo also supports a read-only `DiskTable` interface for streaming data from disk.

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

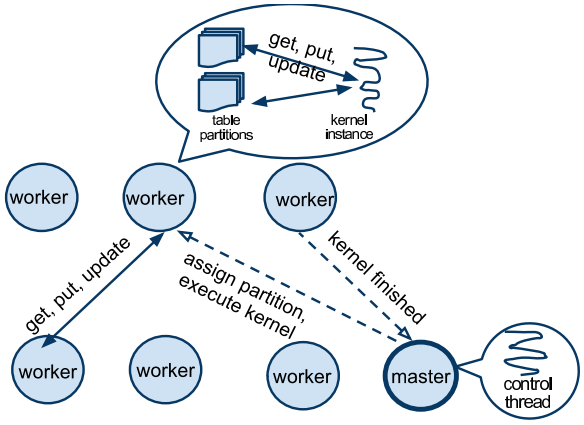

Figure 3: The interactions between master and workers in executing a Piccolo program.

(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. For each $page \rightarrow target$ link, the kernel instance reads the rank value of *page* in curr, and generates updates for next to increment *target*'s rank value for the next iteration.

Since the program generates concurrent updates to the same key in next, it associates the Sum accumulator 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. In addition, the program uses the partition function, SitePartitioner, to assign the URLs in the same domain to the same partition. As pages in the same domain tend to link to one another frequently, such partitioning significantly reduces the number of remote updates.

Checkpointing/restoration is straightforward: the control thread performs a synchronous checkpoint to save the next table every five iterations and loads the latest checkpointed table to recover from failure.

3 System Design

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

3.1 Overview

Piccolo's execution environment consists of one *master* process and many *worker* processes, each executing on a potentially different machine. Figure 3 illustrates the overall interactions among workers and the master when executing a Piccolo program. As Figure 3 shows, the master executes the user control thread by itself and schedules kernel instances to execute on workers. Additionally, the master decides how table partitions are assigned to workers. Each worker is responsible for storing assigned table partitions in its memory and handling table operations associated with those partitions. 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 performance-critical table operations.

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 on a partition assignment. The master informs all workers of the partition assignment and each worker initializes its set of partitions, which are all empty at startup. 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 preferences. Each worker runs a single kernel instance at a time and notifies the master upon task completion. The master instructs each completed worker to proceed with an additional task if it is available. Upon encountering a global barrier, the master blocks the control thread until all active tasks are finished.

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 addresses both challenges.

3.2 Load-balanced Task Scheduling

Basic scheduling without load-balancing works as follows. At table creation time, the master assigns table partitions to all workers using a simple round-robin assign-

ment 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 the specified locality preference, namely, it assigns task i to execute on a worker storing partition i.

This initial schedule may not be ideal. Due to heterogeneous hardware configurations or variable-sized computation inputs, workers can take varying amounts of time to finish assigned tasks, resulting in load imbalance and non-optimal use of machines. Therefore, the runtime needs to load-balance kernel executions beyond the initial schedule.

Piccolo's scheduling freedom is limited by two constraints: First, no running tasks should be killed. As a running kernel instance modifies shared table state, re-executing a terminated kernel instance requires performing an expensive restore operation from a saved checkpoint. Therefore, once a kernel instance is started, it is better to let the task complete than terminating it halfway for re-scheduling. By contrast, MapReduce systems do not have this constraint [28] as reducers do not start aggregation until all mappers are finished. The second constraint comes from the need to honor user locality preferences. 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.

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 assigned tasks to steal a not-yet-started task i from the worker (w_{busy}) with the most remaining tasks. We adopt the greedy heuristic of scheduling larger tasks first. To implement this heuristic, the master estimates the input size of each task by the number of keys in its corresponding table partition. The master collects partition size information from all workers at table loading time as well as at each global barrier. The master instructs each worker to execute its assigned tasks in decreasing order of estimated task sizes. Additionally, the idle worker w_{idle} always steals the biggest task among w_{busy}'s remaining tasks.

Table partition migration: Because of user locality preference, worker w_{idle} needs to transfer the corresponding table partition i from w_{busy} before it executes stolen task i. Since table 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 in order to preserve table semantics. Piccolo uses a multi-phase migration process that does not require suspending any active tasks.

The master coordinates the process of migrating partition i from w_a to w_b, which proceeds in two phases. In the first phase, the master sends message M_1 to all workers indicating the new ownership of i. Upon receiving M_1, all workers flush their buffered operations for i to w_a and begin to send subsequent requests for i to w_b. Upon the receipt of M_1, w_a "pauses" updates to i, and begins to forward requests received from other workers for i to w_b. w_a then transfers the paused state for i to w_b. During this phase, worker w_b buffers all requests for i received from w_a or other workers but does not yet handle them.

After the master has received acknowledgments from all workers that the first phase is complete, it sends M_2 to w_a and w_b to complete migration. Upon receiving M_2, w_a flushes any pending operations destined for i to w_b and discards the paused state for partition i. w_b first handles buffered operations received from w_a in order and then resumes normal operation on 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 during migration.

3.3 Fault Tolerance

Piccolo relies on user-assisted checkpoint and restore to cope with both master and worker failures during program execution. 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 needs to save a consistent global checkpoint with low overhead. To ensure consistency, Piccolo must determine a global snapshot of the program state. To reduce overhead, the run-time must carry out checkpointing in the face of actively running kernel instances or the control thread.

We use the Chandy-Lamport (CL) distributed snapshot algorithm [15] 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 as kernels communicate with each other exclusively via tables.

To begin a checkpoint, the master chooses a new checkpoint epoch number (E) and sends the start checkpoint message $Start_E$ to all workers. Upon receiving the start message, worker w immediately takes a snapshot of the current state of its responsible table partitions and buffers future table operations (in addition to applying them). Once the table partitions in the snapshot are written to stable storage, w sends the marker message $M_{E,w}$ to all other workers. Worker w then enters a logging state in which it logs all buffered operations to a replay

file. Once w has received markers from all other workers ($M_{E,w'}, \forall w' \neq w$), it writes the replay log to stable storage and sends $Fin_{E,w}$ to the master. The master considers the checkpointing done once it has received $Fin_{E,w}$ from all workers.

For asynchronous checkpoints, the master initiates checkpoints periodically based on a timer. To record user-data consistently with recorded table state, each worker atomically takes a snapshot of table state and invokes the checkpoint callback function to save any additional user state for its currently running kernel instance. Synchronous checkpoints provide the semantics that checkpointed state is equivalent to those immediately after the global barrier. Therefore, for synchronous checkpointing, each worker waits until it has completed all its assigned tasks before sending the checkpoint marker $M_{E,w}$ to all other workers. Furthermore, the master saves user-data in the control thread only after it has received $Fin_{E,w}$ 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 assigned 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 resets the state of all workers and restores computation from the last completed global checkpoint. Piccolo does not checkpoint the internal state of the master - if the master is restarted, restoration occurs as normal, however, 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 summarizes how Piccolo's programming model enables efficient implementation for these applications.

4.1 Distributed Web Crawler

Apart from iterative computations such as PageRank, 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. The basic crawler operation is simple: beginning from a few initial URLs, the crawler repeatedly downloads a page and parses it 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 capping fetches to a site at a fixed rate, and (3) avoid repeated fetches of the same URL.

```
#local variables kept by each kernel instance
fetch_pool = Queue()
crawl_output = OutputLog('./crawl.data')

def FetcherThread():
  while 1:
    url = fetch_pool.get()
    txt = download_url(url)
    crawl_output.add(url, txt)

    for l in get_links(txt):
      url_table.update(l, ShouldFetch)
    url_table.update(url, Done)


def CrawlKernel(Table(URL,CrawlState) url_table):
  for i in range(20)
    t = FetcherThread()
    t.start()

  while 1:
    for url, status in url_table.my_partition :
      if status == ShouldFetch
        #omit checking domain in robots table
        #omit checking domain in politeness table
        url_table.update(url, Fetching)
        fetch_pool.add(url)
```

Figure 4: Snippet of the crawler implementation.

Our implementation uses three co-located tables:

- The *url_table* stores the crawling state *ToFetch, Fetching, Blacklisted, Done* for each URL. For each URL p in *ToFetch* state, the crawler fetches the corresponding web page and sets p's state to *Fetching*. After the crawler has finished parsing p and extracting its outgoing links, it 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.

The crawler spawns m kernel instances, one for each machine. Our implementation is done in Python in order to utilize Python's web-related libraries. Figure 4 shows the simplified crawler kernel (omitting details for processing robots.txt and capping per-site download rate). 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 partitioned according to the `SitePartitioner` function and co-located with each other, a kernel instance can efficiently check for the politeness information and robots entries before downloading a URL. Our implementation uses the max accumulator to resolve write-write conflicts on the same

URL in `url_table` according to $Done > Blacklisted > Fetching > ToFetch$. This allows the simple and elegant operation shown in Figure 4, where kernels rediscovering an already-fetched URL p can request updating p's state to *ToFetch* and still arrive at the correct state for p.

Consistent global checkpointing is important for the crawler's recovery. Without global checkpointing, the recovered crawler may find a page p to be *Done* but does not see any of p's extracted links in the `url_table`, possibly causing those URLs to never be crawled. Our implementation performs asynchronous checkpointing every 10 minutes so that the crawler loses no more than 10 minutes worth of progress due to node failure. Restoring from the last checkpoint can result in some pages being crawled more than once (those lost since the last checkpoint), but the checkpoint mechanism guarantees that no pages will "fall through the cracks."

4.2 Parallel computation

k-means. The k-means algorithm is an iterative computation for grouping n data points into k clusters in a multi-dimensional space. Our implementation stores the assigned centers for data points and the positions of centers in shared tables. Each kernel instance processes a subset of data points to compute new center assignments for those data points and update center positions for the next iteration using the summation accumulator.

n-body. This application simulates the dynamics of a set of particles over many discrete time-steps. We implemented an n-body simulation intended for short distances [43], where particles further than a threshold distance (r) apart are assumed to have no effect on each other. During each time-step, a kernel instance processes a subset of particles: it updates a particle's velocity and position based on its current velocity and the positions of other particles within r distance away. Our implementation uses a partition function to divide space into cubes so that a kernel instance mostly performs local reads in order to retrieve those particles within r distance away.

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 $m \times m$ blocks stored in three tables. Our implementation co-locates tables A, B, C. Each kernel instance processes a partition of table C by computing $C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$.

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 [6] 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 API.

6 Evaluation

We tested the performance of Piccolo on the applications described in Section 4. Some applications, such as PageRank and k-means, can also be implemented using the existing data-flow model and we compared the performance of Piccolo with that of Hadoop for these applications.

The highlights of our results are:

- Piccolo is fast. PageRank and k-means are $11\times$ and $4\times$ faster than those on Hadoop. When compared against the results published for DryadLinq [53], 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 experiment on PageRank also demonstrates good scaling.

- Piccolo can help a non-conventional application like the crawler to achieve good parallel performance. Our crawler, despite being implemented in Python, manages to saturate the Internet bandwidth of our cluster.

6.1 Test Setup

Most experiments were performed using our local cluster of 12 machines: 6 of the machines have 1 quad-core Intel Xeon X3360 (2.83GHz) processor 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 created one worker process per core and pinned each worker to use that core.

For 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 pages in UK[9]. Specifically, we

Application	Default input size	Maximum input size
PageRank	100M pages	1B pages
k-means	25M points, 100 clusters	1B points, 100 clusters
n-body	100K points	10M points
Matrix Multiply	edge size = 2500	edge size = 6000

Figure 5: Application input sizes

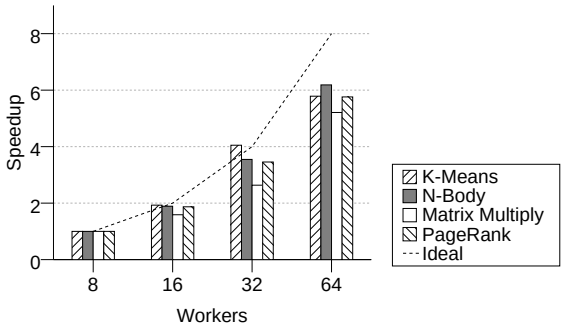

Figure 6: Scaling performance (fixed default input size)

extract the distributions for the number of pages in each site and the ratio of intra/inter-site links. We 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

Figure 6 shows application speedup as the number of workers (N) increases from 8 to 64 for the default input size. All applications are CPU-bound and exhibit good speedup with increasing N. Ideally, all applications (except for PageRank) have perfectly balanced table partitions and should achieve linear speedup. However, to have reasonable running time at N=8, we choose a relatively small default input size. Thus, as N increases to 64, Piccolo's overhead is no longer negligible relative to applications' own computation (e.g. k-means finishes each iteration in 1.4 seconds at 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 § 6.5).

We also evaluate how applications scale with increasing input size by adjusting input size to keep the amount of computation per worker fixed with increasing N. We scale the input size linearly with N for PageRank and k-means. For matrix multiplication, the edge size increases as $O(N^{1/3})$. We do not show results for n-body because it is difficult to scale input size to ensure a fixed amount of computation per worker. For these experiments, the ideal scaling has constant running time as input size increases

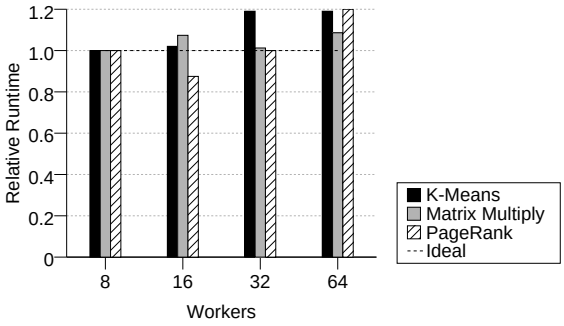

Figure 7: Scaling input size.

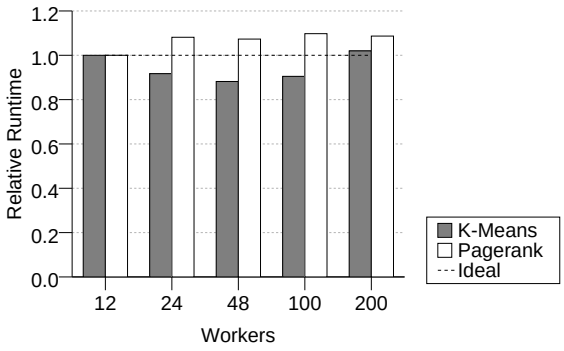

Figure 8: Scaling input size on EC2.

with N. As Figure 7 shows, the achieved scaling for all applications is within 20% of the ideal number.

6.3 EC2

We investigated how Piccolo scales with a larger number of machines using 100 EC2 instances. Figure 8 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 exhibited quite some variability, 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.

6.4 Comparison with Other Frameworks

Comparison with Hadoop: We implemented PageRank and k-means in Hadoop to compare their performance against that of Piccolo. The rest of our applications, including the distributed web crawler, n-body and matrix

multiplication, do not have any straightforward implementation with Hadoop's data-flow model.

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 join the 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 via Hadoop File System (HDFS), 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.

We made extensive efforts to optimize the performance of PageRank and *k*-means on Hadoop including changes to Hadoop itself. Our optimizations include using raw memory comparisons, using primitive types to avoid Java's boxing and unboxing overhead, disabling checksumming, improving Hadoop's join implementation etc. Figure 9 shows the running time of Piccolo and Hadoop using the default input size. Piccolo significantly outperforms Hadoop on both benchmarks (11× for PageRank and 4× 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.

Although we expected to see some performance difference because Hadoop is implemented in Java while Piccolo in C++, the order of magnitude difference came as a surprise. We profiled the PageRank implementation on Hadoop to find the contributing factors. The leading causes for the slowdown are: (1) sorting keys in the map phase (2) serializing and de-serializing data streams and (3) reading and writing to HDFS. Key sorting alone accounted for nearly 50% of the runtime in the PageRank benchmark, and serialization another 15%. In contrast, with Piccolo, the need for (1) is eliminated and the overhead associated with (2) and (3) is greatly reduced. PageRank rank values are stored in memory and are available across iterations without being serialized to a distributed file system. In addition, as most outgoing links point to other pages at the same site, a kernel instance ends up performing most updates directly to locally stored table data, thereby avoiding serialization for those updates entirely.

Comparison with MPI: We compared the the performance of matrix multiplication using Piccolo to a third-party MPI-based implementation [2]. The MPI version uses Cannon's algorithm for blocked matrix multiplication and uses MPI specific communication primitives to handle data broadcast and the simultaneous sending and

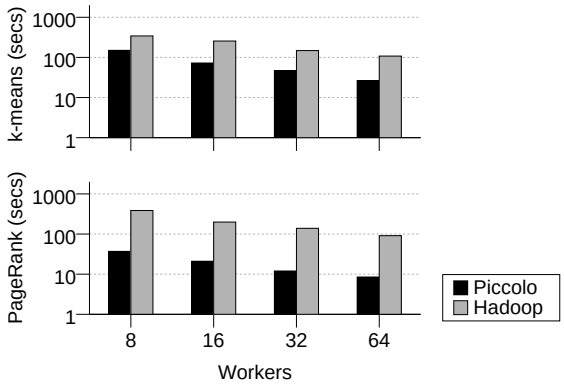

Figure 9: Per-iteration running time of PageRank and *k*-means in Hadoop and Piccolo (fixed default input size).

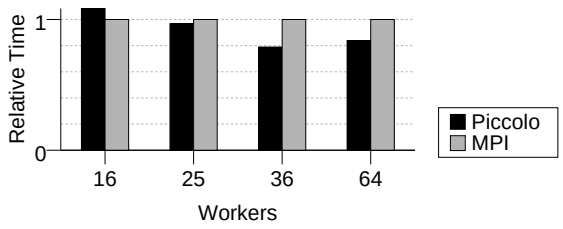

Figure 10: Runtime of matrix multiply, scaled relative to MPI.

receiving of data. For Piccolo, we implemented the naïve blocked multiplication algorithm, using our distributed tables to handle the communication of matrix state. As Piccolo relies on MPI primitives for communication, we do not expect to see performance advantage, but are more interested in quantifying the amount of overhead incurred.

Figure 10 shows that the running time of the Piccolo implementation is no more than 10% of the MPI implementation. We were surprised to see that our Piccolo implementation out-performed the MPI version in experiments with more workers. Upon inspection, we found that this was due to slight performance differences between machines in our cluster; as the MPI implementation has many more synchronization points than that of Piccolo, it is forced to wait for slower nodes to catch up.

6.5 Work Stealing and Slow Machines

The PageRank benchmark provides a good basis for testing the effect of work stealing because the web graph partitions have highly variable sizes: the largest partition for the 900M-page graph is 5 times the size of the smallest. Using the same benchmark, we also tested how performance changed when one worker was operating slower then the rest. To do so, we ran a CPU-intensive program on one core that resulted in the worker bound to that core

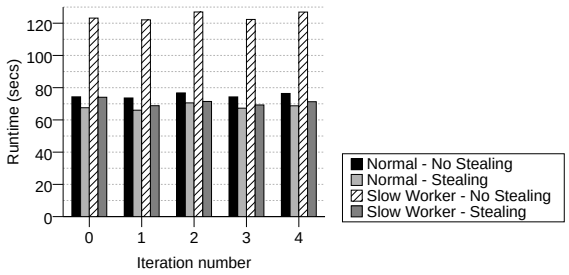

Figure 11: Effect of Work Stealing and Slow Workers

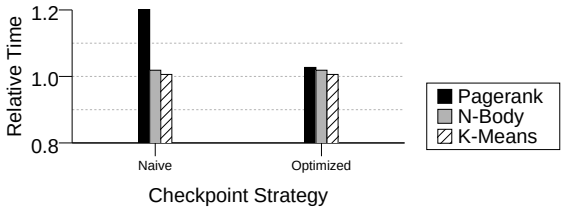

Figure 12: Checkpoint overhead. Per-iteration runtime is scaled relative to without checkpointing.

having only 50% of the CPU time of the other workers.

The results of these tests are shown in Figure 11. Work stealing improves running time by 10% when all machines are operating normally. The improvement is due to the imbalance in the input partition sizes - when run without work stealing, the computation waits longer for the workers processing more data to catch up.

The effect of slow workers on the computation is more dramatic. With work-stealing disabled, the runtime is nearly double that of the normal computation, as each iteration must wait for the slowest worker to complete all assigned tasks. Enabling work stealing improves the situation dramatically - the computation time is reduced to less then 5% over that of the non-slow case.

6.6 Checkpointing

We evaluated the checkpointing overhead using the PageRank, *k*-means and *n*-body problems. Compared to the other problems, PageRank has a larger table that needs to be checkpointed, making it a more demanding test of checkpoint/restore performance. In our experiment, each worker wrote its checkpointed table partitions to the local disk. Figure 12 shows the runtime when checkpointing is enabled relative to when there is no checkpointing. For the naïve synchronous checkpointing strategy, the master starts checkpointing only after all workers have finished. For the optimized strategy, the master initiates the checkpoint as soon as one of the workers has finished. As the figure shows, overhead of the optimized checkpointing strategy is quite negligible ($\sim$2%) and the optimization of starting checkpointing early results in significant reduction of overhead for the larger PageRank checkpoint.

Limitations of global checkpoint and restore: The global nature of Piccolo's failure recovery mechanism raises the question of scalability. As the of a cluster increases, failure becomes more frequent; this causes more frequent checkpointing and restoration which consume a larger fraction of the overall computation time. While we lacked the machine resources to directly test the performance of Piccolo on thousands of machines, we estimate scalability limit of Piccolo's checkpointing mechanism

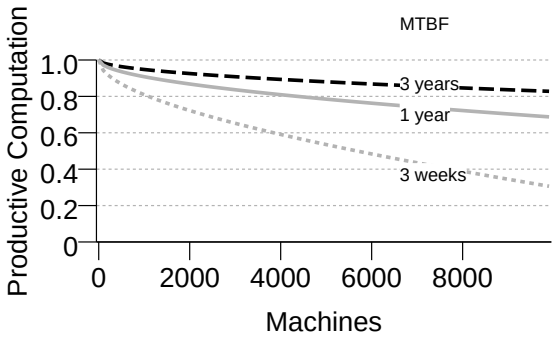

Figure 13: Expected scaling for large clusters.

based on expected machine uptime.

We consider a hypothetical cluster of machines with 16GB of RAM and 4 disk drives. We measured the time taken to checkpoint and restore such a machine in the "worst case" - a computation whose table state uses all available system memory. We estimate the fraction of time a Piccolo computation would spend working productively (not in a checkpoint or restore state), for varying numbers of machines and failure rates. In our model, we assume that machine failures arrive at a constant interval defined by the failure rate and the number of machines in a cluster. While this is a simplification of real-life failure behavior, it is a worst-case scenario for the restore mechanism, and as such provides a useful lower bound. The expected efficiency based on our model is shown in Figure 13. For well maintained data-centers that we are familiar with, the average machine uptime is typically around 1 year. For these data-centers, the global checkpointing mechanism can efficiently scale up to a few thousand machines.

6.7 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 14 shows the crawler's web page download throughput in

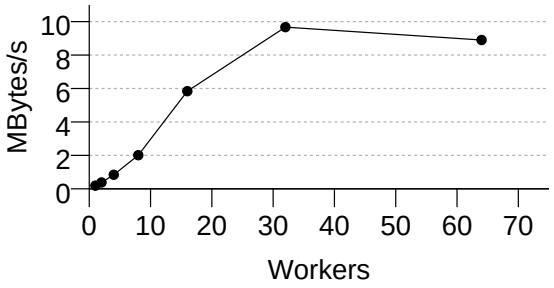

Figure 14: Crawler throughput

MBytes/sec as N increases from 1 to 64. The crawler spends most CPU time in the Python code for parsing HTML and URLs. Therefore, its throughput scales approximately linearly with N. At N=32, the crawler download throughput peaks at $\sim$10MB/s which is limited by our 100-Mbps Internet uplink. There are highly optimized single-server crawler implementations that can sustain higher download rates than 100Mbps [49]. However, our Piccolo-based crawler could potentially scale to even higher download rates despite being built using Python.

7 Related Work

Communication-oriented models: Communication-based primitives such as MPI [21] and Parallel Virtual Machine (PVM [46]) have been popular for constructing distributed programs for many years. MPI and PVM offer extensive messaging mechanisms including unicast and broadcast as well as support for creating and managing remote processes in a distributed environment. There has been continuous research on developing experimental features for MPI, such as optimization of collective operations [3], fault-tolerance via machine virtualization [34] and the use of hybrid checkpoint and logging for recovery [10]. MPI has been used to build very high performance applications - its support of explicit communication allows considerable flexibility in writing applications to take advantage of a wide variety of network topologies in supercomputing environments. This flexibility has a cost in the form of complexity - users must explicitly manage communication and synchronization of state between workers, which can become difficult to do while attempting to retain efficient and correct execution.

BSP (Bulk Synchronous Parallel) is a high-level communication-oriented model [50]. In this model, threads execute on different processors with local memory, communicate with each other using messages, and perform global-barrier synchronization. BSP implementations are typically realized using MPI [25]. Recently, the BSP model has been adopted in the Pregel framework for parallelizing work on large graphs [33].

Distributed shared-memory: The complexity of programming for communication-oriented models drove a wave of research in the area of distributed shared memory (DSM) systems [30, 29, 32, 7]. Most DSM systems aim to provide *transparent* memory access, which causes programs written for DSMs to incur many fine-grained synchronization events 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 naïve remote accesses and synchronization prohibitively expensive.

Parallel Global Address Space (PGAS) [17, 35, 51] are a set of language extensions to realize a distributed shared address space. These extensions try to ameliorate the latency problems of DSM 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 (flat memory) interface common to DSM. As a result, applications written for PGAS systems still require fine-grained synchronization when operating on non-primitive datatypes, 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 [13] and more recently JavaSpaces [22], expose to users a global tuple-space accessible from all participating threads. Although tuple spaces provide atomic primitives for reading and writing tuples, they are not intended for high-frequency access. As such, there is no support for locality optimization nor write-write conflict resolution.

MapReduce and Dataflow models: In recent years, MapReduce has emerged as a popular programming model for parallel data processing [19]. There are many recent efforts inspired by MapReduce ranging from generalizing MapReduce to support the join operation [27], improving MapReduce's pipelining performance [16], building high-level languages on top of MapReduce (e.g. DryadLINQ [53], Hive [48], Pig [37] and Sawzall [40]). FlumeJava [14] provides a set of collection abstractions and parallel execution primitives which are optimized and compiled down to a sequence of MapReduce operations.

The programming models of MapReduce [19] and Dryad [27] are instances of stream processing, or data-flow models. Because of MapReduce's popularity, programmers start using it to build in-memory iterative applications such as PageRank, even though the data-flow model is not a natural fit for these applications. Spark [54] proposes to add distributed read-only in-memory cache to improve the performance of

MapReduce-based iterative computations.

Single-machine shared memory models: Many programming models are available for parallelizing execution on a single machine. In this setting, there exists a physically-shared memory among computing cores supporting low-latency memory access and fast synchronization between threads of computation, which are not available in a distributed environment. Although there are also popular streaming/data-flow models [44, 47, 12], most parallel models for a single machine are based on shared-memory. For the GPU platform, there are CUDA [36] and OpenCL [24]. For multi-core CPUs, Cilk [8] and more recently, Intel's Thread Building Blocks [41] provide support for low-overhead thread creation and dispatching of tasks at a fine level. OpenMP [18] is a popular shared-memory model among the scientific computing community: it allows users to target sections of code for parallel execution and provides synchronization and reduction primitives. Recently, there have been efforts to support OpenMP programs across a cluster of machines [26, 5]. However, based on software distributed shared memory, the resulting implementations suffer from the same limitations of DSMs and PGAS systems.

Distributed data structures: The goal of distributed data structures is to provide a flexible and scalable data storage or caching interface. Examples of these include DDS [23], Memcached [39], the recently proposed RamCloud [38], and many key-value stores based on distributed hash tables [4, 20, 45, 42]. These systems do not seek to provide a computation model, but rather are targeted towards loosely-coupled distributed applications such as web serving.

8 Conclusion

Parallel in-memory application need to access and share intermediate state that reside on different machines. Piccolo provides a programming model that supports the sharing of mutable, distributed in-memory state via a key/value table interface. Piccolo helps applications achieve high performance by optimizing for locality of access to shared state and having the run-time automatically resolve write-write conflicts using application-specified accumulation functions.

Acknowledgments

Yasemin Avcular and Christopher Mitchell ran some of the Hadoop experiments. We thank the many people who have improved this work through discussion and reviews: the members of NeWS group at NYU, Frank Dabek, Rob Fergus, Michael Freedman, Robert Grimm, Wilson Hsieh, Frans Kaashoek, Jinyuan Li, Robert Morris, Sam Roweis, Torsten Suel, Junfeng Yang, Nickolai Zeldovich.

References

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Example matrix multiplication implementation using mpi. `http://www.cs.umanitoba.ca/~comp4510/examples.html`.

[3] ALMÁSI, G., HEIDELBERGER, P., ARCHER, C. J., MARTORELL, X., ERWAY, C. C., MOREIRA, J. E., STEINMACHER-BUROW, B., AND ZHENG, Y. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the 19th annual international conference on Supercomputing* (New York, NY, USA, 2005), ICS '05, ACM, pp. 253–262.

[4] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[5] BASUMALLIK, A., MIN, S.-J., AND EIGENMANN, R. Programming distributed memory sytems using OpenMP. *Parallel and Distributed Processing Symposium, International 0* (2007), 207.

[6] BEAZLEY, D. M. Automated scientific software scripting with SWIG. *Future Gener. Comput. Syst. 19* (July 2003), 599–609.

[7] BERSHAD, B. N., ZEKAUSKAS, M. J., AND SAWDON, W. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference* (1993).

[8] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 1995), ACM, pp. 207–216.

[9] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[10] BOSILCA, G., BOUTEILLER, A., CAPPELLO, F., DJILALI, S., FEDAK, G., GERMAIN, C., HERAULT, T., LEMARINIER, P., LODYGENSKY, O., MAGNIETTE, F., NERI, V., AND SELIKHOV, A. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 2002), Supercomputing '02, IEEE Computer Society Press, pp. 1–18.

[11] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems 30*, 1-7 (1998), 107 – 117. Proceedings of the Seventh International World Wide Web Conference.

[12] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers* (2004), ACM, p. 786.

[13] CARRIERO, N., AND GELERNTER, D. Linda in context. *Commun. ACM 32*, 4 (1989), 444–458.

[14] CHAMBERS, C., RANIWALA, A., PERRY, F., ADAMS, S., HENRY, R. R., BRADSHAW, R., AND WEIZENBAUM, N. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI - ACM SIGPLAN 2010* (2010).

[15] CHANDY, K. M., AND LAMPORT, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3* (1985), 63–75.

[16] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTEIN, J. MapReduce online. In *NSDI* (2010).

[17] CONSORTIUM, U. UPC language specifications, v1.2. Tech. rep., Lawrence Berkeley National Lab, 2005.

[18] DAGUM, L., AND MENON, R. Open MP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering 5*, 1 (1998), 46–55.

[19] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (2004).

[20] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles* (Oct. 2007), pp. 205–220.

[21] FORUM, M. MPI 2.0 standard, 1997.

[22] FREEMAN, E., ARNOLD, K., AND HUPFER, S. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[23] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation* (Berkeley, CA, USA, 2000), USENIX Association, pp. 22–22.

[24] GROUP, K. O. W. The OpenCL specification. Tech. rep., 2009.

[25] HILL, J., MCCOLL, W., STEFANESCU, D., GOUDREAU, M., LANG, K., RAO, S., SUEL, T., TSANTILAS, T., AND BISSELING, H. Bsplib: The bsp programming library. *Parallel Computing 24* (1998).

[26] HOEFLINGER, J. P. Extending OpenMP to clusters. Tech. rep., Intel, 2009.

[27] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)* (2007).

[28] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *SOSP* (2010).

[29] JOHNSON, K. L., KAASHOEK, M. F., AND WALLACH, D. A. CRL: High-performance all-software distributed shared memory. In *SOSP* (1995).

[30] KELEHER, P., COX, A. L., AND ZWAENEPOEL, W. Lazy release consistency for software distributed shared memory. In *In Proceedings of the 19th Annual International Symposium on Computer Architecture* (1992).

[31] LAMPORT, L. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers 28*, 9 (1979).

[32] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS) 7* (1989), 321–359.

[33] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), ACM, pp. 135–146.

[34] NAGARAJAN, A. B., MUELLER, F., ENGELMANN, C., AND SCOTT, S. L. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing* (New York, NY, USA, 2007), ICS '07, ACM, pp. 23–32.

[35] NUMRICH, R. W., AND REID, J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum 17* (August 1998), 1–31.

[36] NVIDIA. CUDA programming guide (ver 3.0).

[37] OLSON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD* (2008).

[38] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIERES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBERL, S., STRATMANN, E., AND STUTSMAN, R. The case for RAMclouds: Scalable high-performance storage entirely in DRAM. In *Operating system review* (Dec. 2009).

[39] PHILLIPS, L., AND FITZPATRICK, B. Livejournal's backend and memcached: Past, present, and future. In *LISA* (2004), USENIX.

[40] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. In *Scientific Programming* (2005).

[41] REINDERS, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[42] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *18th IFIP/ACM International Conference on Distributed Systems Platforms* (Nov. 2001).

[43] SINGH, J. P., WEBER, W.-D., AND GUPTA, A. SPLASH: Stanford parallel applications for shared-memory. Tech. rep., Stanford University, 1991.

[44] STEPHENS, R. A survey of stream processing, 1995.

[45] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* (2002), 149–160.

[46] SUNDERAM, V. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience* (1990), 315–339.

[47] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. StreamIt: A language for streaming applications. In *Compiler Construction* (2002), Springer, pp. 49–84.

[48] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2* (August 2009), 1626–1629.

[49] TSANG LEE, H., LEONARD, D., WANG, X., AND LOGUINOV, D. Irlbot: Scaling to 6 billion pages and beyond. In *WWW Conference* (2008).

[50] VALIANT, L. A bridging model for parallel computation. *Communications of the ACM 33* (1990).

[51] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., GRAHAM, P. H. S., GAY, D., COLELLA, P., AND AIKEN, A. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience 10*, 11 (1998).

[52] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[53] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)* (2008).

[54] ZAHARIA, M., CHOWDHURY, N. M. M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster Computing with Working Sets. Tech. Rep. UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.