Abstractions for In-memory Distributed Computation

by

Russell Power

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Mathematics

New York University

May 2014

Professor Jinyang Li

Abstract

The recent cloud computing revolution has changed the distributed computing landscape, making the resources of entire datacenters available to ordinary users. This process has been greatly aided by dataflow style frameworks such as MapReduce which expose simple model for programs, allowing for efficient, fault-tolerant execution across many machines. While the MapReduce model has proved to be effective for many applications, there are a wide class of applications which are difficult to write or inefficient in such a model. This includes many familiar and important applications such as PageRank, matrix factorization and a number of machine learning algorithms. In lieu of a good framework for building these applications, users resort to writing applications "by-hand", using MPI or RPC, a difficult and error-prone construction.

This thesis presents 2 complementary frameworks, Piccolo and Spartan, which help programmers to write in-memory distributed applications not served well by existing approaches.

Piccolo presents a new data-centric programming model for in-memory applications. Unlike data-flow models, Piccolo allows programs running on different machines to share distributed, mutable state via a key-value table interface. This design allows for both high-performance and additional flexibility. Piccolo makes novel use of commutative updates to efficiently resolve write-write conflicts. We find Piccolo provides an efficient backend for a wide-range of applications: from PageRank and matrix multiplication to web-crawling.

While Piccolo provides an efficient backend for distributed computation, it can still be somewhat cumbersome to write programs using it directly. To address this, we created Spartan. Spartan implements a distributed implementation of the NumPy array language, and fully supports important array language features such as spatial indexing (slicing), fancy indexing and broadcasting. A key feature of Spartan is its use of a small number of simple, powerful *high-level operators* to provide most functionality. Not only do these operators dramatically simplify the design and implementation of Spartan, they also allow users to implement new functionality with ease.

We evaluate Piccolo and Spartan on a wide range of applications and find that they both perform significantly better than existing approaches.

Contents

List of Figures

List of Tables

Listings

1
Introduction

Distributed computing – leveraging large numbers of machines to solve problems – is no longer a niche application. Publicly available services such as Amazon EC2 and Google's Compute Engine allow users to effectively rent hundreds or thousands of machines for short periods of time. This is a radical change: even 10 years ago, distributed computing was the exclusive domain of large software companies, government research labs and well-financed universities. Programmers can now leverage these machine resources to to speed up their applications or to run them on more data.

Of course, having access to machines takes us only so far: programmers also need to write code which uses these machines to solve their problem. Unfortunately, programming distributed applications is notoriously difficult. Developers must deal with a large number of challenges, including:

- *Coordination.* Data and computation must be partitioned across machines to maximize effiency. As accessing information from a remote machine is orders of magnitude slower than reading local data, care must be taken to minimize communication.

- *Synchronization.* Programming distributed systems requires handling what amounts to "multi-threading on steroids": updates to shared data from thousands of machines must be synchronized efficiently and correctly.

- *Hardware failures.* Individual machines and datacenter networks are reliable, but when running on thousands of machines, failures are frequent. For long running computations, some mechanism must be provided to ensure that a single machine failure does not result in the entire computation being restarted.

- *Slow machines.* Even in a datacenter where every machine is theoretically identical, programmers must account for some machines running slower than others. This is often due to other processes sharing the machine, but it can also be the result of unexpected hardware defects. [1]

To address the difficulties of writing distributed applications, a variety of distributed computation systems have been created. These systems allow programmers to write code for a simple model of execution; the system translates operations on the model to actually run on a cluster of machines. The goal of any of these systems to insulate users from the vagaries of programming "raw" distributed systems, while at the same time offering reasonably high performance. These systems can be divided into two main groups: distributed shared memory (DSM) and dataflow.

DSM is the first distributed programming model to arise, and it continues to be actively researched; modern implementations include UPC [33], Chapel [24] and X10 [28]. (We include key-value models such as Linda and JavaSpaces in the DSM category). DSM models all provide a mechanism for starting new processes, and allow processes to communicate using a *shared address space*. Intuitively, these models extend the concept multi-threading, frequently seen on a single

[1] A frequent culprit is overheating, which causes CPU's to slow down to avoid damage. As an example of a more difficult case: the author has personally seen an issue where writing to certain memory locations triggered a CPU bug which disabled caching. This would (seemingly at-random) cause programs to run an order of magnitude slower.

machine; enabling "threads" to be run on many machines in parallel. Modern implementations take into consideration issues such as locality of reference (reading/writing from a local machine is an order-of-magnitude faster than from a remote machine).

Dataflow frameworks (popularized by MapReduce [36] and its open-source implementation Hadoop [1]), eschew the idea of any *direct* communication between processes. (This is unlike DSM, where processes can interact with one another via the shared address space). Instead, dataflow frameworks are organized around sparse (key-value) collections. To manipulate these collections, a small number of *data parallel operators* such as `map`, `shuffle`, and `reduce` are supplied. Programmers combine applications of these data-parallel operators to transform their data and compute a desired result.

Given the large number of competing frameworks available, we ned some way to compare models with one another: what makes a distributed frameworks "good"? To begin, we need to consider a problem we want to solve; a system may be efficient and easy to use for one problem (running grep across a million web pages), but bad for another (computing the Cholesky factorization of a matrix). For our purposes, we consider implementing the PageRank algorithm for ranking web pages, as shown in Listing 1.1.

```
# graph is a list of pages, each of which contains a number of links
# pages are identified by a 64 bit number

def pagerank(graph):
  current = random(graph.size)
  while not converged:
    next = [0...]
    for page in graph:
      for link in page:
        next[link] += current[page.id] / page.num_links
    swap(current, next)
  return current
```

Listing 1.1: PageRank algorithm

PageRank computes a rank for each web page in the graph and attempts to assign higher ranks to "good" pages[2].

This algorithm is fairly simple to represent in a single threaded context; this gives us reason to hope that a distributed version will not be too complicated (at least conceptually). It also

[2]The definition is recursive: a good page is a one that has many links from other good pages. (The result of the PageRank computation is actually the eigen-vector of the graph matrix.)

exhibits features which should make it relatively easy to parallelize: it runs over the entire graph of data at a time, and updates should be spread out across all pages (assuming a non-adversarial graph). What happens when we try to write this application using our existing models?

Let's consider how we might write this program using a DSM model. At first glance, this seems straightforward: we store our `graph`, `current`, and `next` collections in our shared memory space, and have each process iterate over some disjoint portion of the graph, accumulating ranks into the next array. We ignore the complication of partitioning the data to minimize remote reads (all modern DSM systems support this in some fashion)[3]. Our version of this application look like Listing 1.2.

```
# graph, current, and next are stored in a shared address space
def dsm_pagerank():
  while not converged:
    next.clear()
    forall(pages p in graph):
      for link in p:
        next[link] += current[p.id] / p.num_links
    swap(current, next)
  return current
```

Listing 1.2: DSM PageRank Implementation

Here we are using the `forall` operator to run our inner loop in parallel across our graph. This implementation looks good, but it is hiding a critical problem. Consider the following line of our code:

```
next[link] += current[p.id] / p.num_links
```

If we expand the +=, we see we are doing a read and a write:

```
next[link] = next[link] + current[p.id] / p.num_links
```

This line has two problems. First, our process is reading from an effectively random memory location, virtually ensuring we have to talk to a remote machine to fetch the value of `next[link]`. Second, many processes may be updating the value of `next[link]` simultaneously; without some form of synchronization, these updates will conflict and our computation will be incorrect.

[3] In a typical system, page identifiers are *sparse*: they are spread out across the address space. To handle this, we must use hash maps or a similar structure to manage our rank data. Coordinating updates to such a data structure across many processes is both slow and nightmarishly complex. This disqualifies most DSM systems (including X10 and UPC) from being used for our problem. Some DSM systems (Linda, Javaspaces, and Chapel) do support hashmaps as a builtin structure, so we will assume we are using such a model.

On a local machine, we can resolve these issues by either locking the `next` array to protect the read and write, or using an atomic operation such as `atomic_add` to update the value in-place. Most DSM models offer similar operations, but the problem is that these primitives no longer scale in a distributed context. On a single machine, locking or atomic operations are resolved at the level of a the CPU cache, with latencies under 10 nanoseconds. In a distributed system, these operations require network communication between hosts with a minimum latency of 2 microseconds: several orders of magnitude slower[4]. These issues effectively bottleneck the inner loop of our application, making it impossible to scale efficiently[5].

DSM frameworks struggle to implement PageRank efficiently. By contrast, using a dataflow model we can write our PageRank computation in a relatively elegant and performant fashion. This is illustrated by the (simplified) implementation shown in Listing 1.3.

```
# graph, current and next are stored on a distributed file system
def mr_pagerank():
  while not converged:
    joined = join(graph, current)

    # output an update for each link in the graph
    updates = map(joined,
      fn(page, pr):
        for link in page:
          yield link, current[page.id] / p.num_links
    )

    # shuffle so that all updates for a given page are together
    grouped = group_by_key(updates)

    # sum together all of the updates for a page
    next = reduce(updates, fn(page, updates): (page, sum(updates)))

    # write out our data so we can read it for the
    # next iteration
    next.write_to_disk()
```

Listing 1.3: MapReduce PageRank Implementation

Note that the use of the data parallel operators completely hides the difficulty of handling concurrent updates: the `reduce` operator simply manages this for us internally. While this implementation seems more scalable than our attempt with DSM, we still have a problem. Dataflow systems like MapReduce require that data be written back out to a distributed file system before

[4]This number is for a high-performance network (e.g. Infiniband). On a more common ethernet based network, latencies are another order of magnitude higher.

[5]Technically, programmers could manually buffer updates using local memory, and exchange data manually between workers at fixed intervals. But at this point, our model isn't really saving us any effort versus implementing our application by hand.

it can be used. In this case, we must write our `next` collection out after each iteration. This forces us to serialize and replicate our data to disk, which turns out to be prohibitively expensive relative to the cost of actually computing the new ranks[6].

The skeptical reader will likely be thinking that our PageRank application has been carefully chosen and is exceptional in some way. If so, we needn't worry about the fact that our current models don't help with writing it: we can simply implement it manually once, and use our framework of choice for other applications. Unfortunately, PageRank is actually emblematic of an entire class of important applications, including:

- Matrix factorization algorithms like Singular Value Decomposition (SVD) and Principal Component Analysis (PCA).
- Eigen-vector computation (e.g. PageRank)
- Many machine learning algorithms, including regression, neural networks, clustering and nearest neighbors.
- n-body simulation
- Continuously running programs, such as web crawlers.

These applications share 3 properties which distinguish them:

- *State fits in memory.* We can store the mutable state of our application in the total memory available on our cluster. This implies that serializing data can be avoided while processing.
- *Iterative or continuous operation.* As these applications require many passes over a data set (or run perpetually), the overhead of serializing data between iterations is an important performance concern.
- *Concurrent updates to state.* Each of these applications requires multiple processes to make updates to shared state. For correctness, we need some way of efficiently synchronizing these updates.

[6]Since the time the work in this thesis was published, other systems have addressed some, but not all, of the limitations of dataflow frameworks. We discuss this more in Section 6.

A system which can efficiently model these types of operations would allow us to implement any of the above applications easily. In this thesis, we present two complementary approaches which dramatically simplify writing these applications. We first present Piccolo, a framework for distributed in-memory applications. Piccolo combines features from dataflow frameworks and DSM, allowing programmers to efficiently implement applications like our PageRank example above. The Piccolo model provides allows programmers to create in-memory key-value tables and introduces the concept of "accumulators" which allow concurrent updates without the need for expensive locking or atomic operations. We demonstrate how Piccolo can be used to implement a variety of in-memory distributed applications efficiently.

Second, we present Spartan, a distributed array language implementation. Spartan closes the "semantic gap" between the Piccolo interface and the type of code users want to write. It provides a distributed implementation of the popular NumPy [64] array language extension to Python. Spartan uses a number of techniques including lazy evaluation, high-level operators and expression graph optimization, to efficiently execute high-level array language applications. We used Spartan to implement a variety of array programs from machine learning and finance, and show that it performs competitively with existing solutions while offering an easy-to-use interface.

1.1 Piccolo

We designed Piccolo to address the limitations of existing DSM and dataflow frameworks for writing in-memory computations. Piccolo extends the basic DSM model, allowing users to create one or more typed, key-value tables which are used to share data. Tables are partitioned across machines in a cluster; to help minimize communication, users can control the partitioning by providing a partitioning function. Parallelism in Piccolo is provided via *kernel functions*: these are user defined functions which are executed in parallel on each partition of a table. Kernel instances access data and communicate with one another by reading and writing to tables.

Piccolo tables support the usual (put, get, remove) interface, but also add a new method: *update*. The *update* method, combined with a user-defined *accumulators*, enables Piccolo to automatically combine concurrent updates for the same key (similar to reduce functions in MapReduce [36]). The use of accumulators eliminates the need for fine-grained synchronization for many applications, which as we described above, is a critical performance limitation in other DSM systems.

In addition to the table interface described above, the Piccolo runtime supports dynamic load-balancing of data, even while applications are running, and provides a global checkpoint/restore mechanism to recover from machine failures. The Chandy-Lamport snapshot algorithm [26] is used to generate a consistent snapshot of the execution state without pausing active computations. Upon machine failure, Piccolo recovers by re-starting the computation from its latest snapshot state.

We show that Piccolo can be used to write natural and efficient implementations of a variety of applications, like linear regression, k-means computation, n-body simulation, and PageRank. Piccolo also enables online applications, such as a distributed web crawler, that require immediate access to modified state; a type of operation which is simply not available in dataflow based systems.

Our experiments have shown that Piccolo is fast and provides excellent scaling for many applications. For example, computing a PageRank iteration for a 1 billion-page web graph takes only 70 seconds on 100 EC2 instances (10 times faster than an optimized Hadoop implementation), and a distributed web crawler implemented in Piccolo easily saturates a 100 Mbps internet uplink when running on 12 machines.

1.2 Spartan

Piccolo provides a flexible, efficient platform suitable for in-memory applications. Nevertheless, writing a Piccolo program involved a non-trivial amount of work to define tables and accumulators, and manually write kernels. The cumbersome nature of writing to the raw backend is not limited to Piccolo – we see it with MapReduce style systems as well. One approach taken by many projects is to design a simpler or higher-level language interface which is then compiled down to a MapReduce core. Examples of this include Pig [66], Spark [93], Scalding [87] and DryadLINQ [92].

While a similar approach was not unreasonable for Piccolo, we felt it was more promising to start from the other direction. That is, rather than design a language around our backend, could we take an existing, popular language and use a Piccolo style backend to make an efficient distributed implementation?

As it turns out, a candidate language exists and is widely used: NumPy [64]. NumPy is a array language extension to the Python programming language, and borrows heavily from the syntax and conventions of the popular Matlab commercial language. Array languages rely heavily on intrinsic operations which process entire arrays at a time – an effective source of implicit parallelism. NumPy does not have a distributed implementation (despite some partial attempts [35]), and it is widely used for exactly the type of iterative, non-trivial applications we are interested in.

Implementing a distributed version of an array language such as NumPy presents some interesting challenges. First, NumPy is not a small language: it contains well over 100 builtin operations for examining array contents (slicing, equality comparisons `isnan, etc`), computing aggregate statistics (sum, min, max, mean, mode) and selecting out values of interest (where, nonzero, argmin, argmax). Writing Piccolo implementations of all of these operators by-hand would be immensely time-consuming. To avoid this, we need some way to encapsulate (*abstract?*) common patterns (mapping, filtering, reducing, etc.), and re-use them to simplify writing operations.

Second, idiomatic array programs generate large numbers of expensive temporary variables if they are not optimized. While it is a minor problem for local programs, in a distributed system is an larger issue, as constructing even small arrays requires multiple network round-trips. For instance, consider a simple function which computes a an arbitrary statistic over 2 arrays shown in Listing 1.4. A naïve implementation of such a program would evaluate each expression in isolation (first evaluate `x + y`, then squaring, then evaluating `x - y`, etc). This results in 4 expensive temporary arrays being created. An optimized implementation should combine all of the expressions into a single map over the x and y arrays.

Finally, many users rely on extensions to NumPy itself; most of these extensions are written in a language such as C or Fortran and manipulate the internal state of NumPy arrays. While automatically converting such code to run on a distributed implementation is beyond our scope, we still need to provide some mechanism for users to add new functionality to the system, ideally in a way which cooperates with existing methods and optimizations. An accumulative Piccolo-

```
# each operation actually element-wise over arrays.
# e.g. "x + y" is equivalent to:
# z = new_array(length(x))
# for i in 0 -> length(x):
#   z[i] = x[i] + y[i]

def compute_stat(x, y):
  return sqrt((x + y) ** 2 / (x - y))
```
Listing 1.4: Sample Array Program

style backend is used to store distributed array data, execute parallel operations and coordinate updates to arrays.

Spartan uses three techniques to address these challenges and provide an efficient distributed implementation of NumPy: lazy evaluation, high-level operators and a comprehensive set of expression optimizations.

Instead of directly evaluating operations such as $a+b$, Spartan uses lazy evaluation to capture users intentions in the form of an *expression graph*. The expression graph is then *lowered* to a comprehensive set of *high-level operators*. These operators encapsulate common array operations such as mapping, reducing, slicing and filtering and dramatically simplify the implementation of common builtin functions. They serve another important purpose as well, as they enable array programs to be effectively optimized, which turns out to be critical for performance.

After the expression graph has been converted to use the high-level operators, a number of optimizations can be applied. Optimizations such as map and reduce fusion and common-subexpression elimination are used to eliminate expensive temporary values. A final optimization pass enables Spartan's integration with Parakeet [77], an optimizing compiler for single-machine NumPy programs. This integration allows Spartan programs to run seamlessly on GPU accelerators and run faster, even on a single machine, than their NumPy equivalents.

Our evaluation of Spartan shows that it can provide the same raw performance as Piccolo (up to 5 times faster than existing in-memory frameworks such as Spark) while presenting users with a familiar and easy-to-use interface.

1.3 Contributions

The contributions of this thesis are:

- The development of the Piccolo computation framework, which provides a flexible key-value storage system with support for accumulators, dynamic load-balancing and an efficient checkpointing system.

- The design and implementation of the Spartan distributed array language. Spartan combines the use of lazy evaluation, high-level operators, a number of optimization passes and a Piccolo-like accumulative backend to provide a complete and efficient implementation of the core of NumPy array language.

The remainder of this thesis is organized as follows: in Chapter 2, we describe the design and implementation of the Piccolo distributed computation framework. Chapter 3 demonstrates Piccolo's performance on a number of applications. Chapter 4 gives a brief introduction to array languages, and details the design of the Spartan distributed array language. Chapter 5 shows Spartan's performance on a number of common applications. In Chapter 6 we describe related work. We conclude with a review of the presented work.

2 Piccolo Design

2.1 Overview

Piccolo was designed to address the limitations we found in existing frameworks when it came to writing distributed in-memory computations, such as PageRank, *k*-means and *n*-body simulation. In particular, we desired the following features:

- Applications should be able to access and manipulate shared state at all times during a computation.
- Multiple processes must be able to update a common data item in an efficient and consistent manner.

- Users should be able to specify locality policies to minimize remote data access.
- The system should provide fault-tolerance with a minimum of user effort.

To accomplish these goals, Piccolo combines ideas from both DSM and data-flow frameworks. In Piccolo, users store and access application data using typed, key-value tables. These tables are split into a number of partitions, which are then distributed across all of the machines involved in an application. Partitioning can be controlled by the user by specifying a partition function; this allows users to ensure locality of reference for data. Piccolo provides *accumulators* (similar to *reducers* in data-flow systems) as a mechanism for synchronizing updates to shared table entries. Finally, Piccolo provides an efficient checkpointing mechanism which requires little user effort to use.

The remainder of this chapter details the design and implementation of the Piccolo framework. Section 2.2 describes the programming provided for users, this is followed by the design of the Piccolo runtime (Section 2.3), and a brief description of our implementation (Section 2.4).

2.2 Programming Model

2.2.1 Program structure

Application programs written for Piccolo consist a *control* function which is executed a single machine, and *kernel* functions which are executed concurrently on many machines. Control functions create shared tables, launch tasks (instances of a kernel function which run in parallel), and perform global synchronization. Kernel functions consist of sequential code; kernel functions use shared tables to cooperate and share state among concurrently executing tasks.

2.2.2 Controller Interface

The user-supplied controller function is responsible for creating and destroying tables and running kernels. The API used by the controller function to accomplish these tasks is shown in Listing 2.1. The controller function performs 3 basic operations:

Creating tables: The `create` and load methods are used to create a new table or load an existing table from disk. Tables must be created by the controller function prior to use.

```
Master:
  Table<Key, Value> load(key_type, value_type)
  Table<Key, Value> create(key_type, value_type, accumulator, partitioner)
  destroy(table)
  group([tables])

  run(table, kernel_fn, args)

  barrier()
  cp_barrier([tables], userdata)
  cp_continuous([tables], userdata, frequency)

  userdata restore_from_checkpoint()
```

Listing 2.1: Controller Interface

Piccolo tables are *typed* – these are arbitrary user-defined types (e.g. string, float, int, etc.). Both key and value types must be serializable (for storing and moving table data). Having types helps ensure program correctness and also enables Piccolo to efficiently serialize and combine table data. Tables are also *partitioned*; table data is split across workers – each worker is responsible for one or more partitions of a table.

The usage of the *accumulator* and *partitioner* arguments is described later. Pre-existing data located on a shared filesystem (e.g GFS) can be loaded into Piccolo using the `load` method. Tables loaded this way are read-only.

Kernel invocation: The programmer uses the `run` function to launch kernel instances executing the desired kernel function. One kernel instance is launched for each partition of the supplied table; instances are run *with locality*: kernel instance i runs on the worker which holds table partition i. Kernel functions can obtain references to and manipulate existing tables; they cannot create new tables. Each kernel instance has an identifier $0 \cdots m-1$ which can be retrieved using the `my_instance` function; typically this identifier is used to iterate over a local portion of the table.

Kernel synchronization: The programmer invokes a global barrier from within a control function to wait for the completion of all previously launched kernels. Piccolo does not support locking of individual table entries. We have found that Piccolo's use of accumulation functions (described below) eliminates most of the cases where locking operations would otherwise be required. This overall application structure, where control functions launch kernels across one or more global barriers, is reminiscent of the CUDA model [63] which also explicitly eschews support

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

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

  # Commit any buffered updates/puts
  void flush()

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

Listing 2.2: Shared Table Interface

for pair-wise thread synchronization. The usage of the *cp_barrier* and *cp_continous* methods is described in the fault tolerance section 2.3.2.

2.2.3 Table interface and semantics

Concurrent kernel instances share intermediate state across machines using 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. As Figure 2.2 shows, the key-value interface provides a uniform access model whether the underlying table entry is stored locally or on another machine. The table APIs include the expected `get`, `put` and `remove` operations, but also includes Piccolo-specific functions like `update`, `flush` and `get_iterator`. Only control functions can create tables; both control and kernel functions can read and write to tables.

User-defined accumulation: In many programs, multiple kernel instances can issue concurrent `update`s to the same key. To resolve such write-write conflicts, Piccolo allows programmers to associate a user-defined accumulation function with each table. Piccolo executes the accumulator during run-time to combine concurrent updates 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 [91]. The interface used for accumulators is shown in Figure 2.3.

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`

```
Accumulator<Value>
  initialize()
  update(Value)
  merge(Accumulator<Value>)
  Value view()
```

Listing 2.3: Accumulator Interface

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; they can only access their local state and supplied update value.

By using accumulator functions, Piccolo applications can avoid most, if not all, of the situations where locking would be required in another framework.

Table Partitioning: Piccolo uses a user-specified *partition function* [36] 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 2.2).

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 are 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 [51].

- 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, all kernel instances will have been completed and all their writes will have been applied.

2.2.4 Expressing locality preferences

While accumulators allow writes to remote table entries to be buffered and combined, the communication latency involved in *reading* remote table entries cannot be effectively hidden. Given this, a key factor in achieving good application performance is to minimize remote `gets` by exploiting locality of access. 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 a kernel, the programmer specifies the table in the `run` function to express their preference for co-locating the kernel execution with that table. To optimize for kernels that read from more than one table, the programmer uses the `group(T1,T2,..)` function to co-locate multiple tables. The run-time then ensures the i-th partition of T1,T2,... is 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.

2.2.5 User-assisted checkpoint and restore

Piccolo handles machine failures via a global checkpoint/restore mechanism. The mechanism 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 effort required to checkpoint user information is relatively small. In exchange for this small amount of user effort, 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 (`cp_barrier`) and one asynchronous (`cp_periodic`). Both functions are invoked from the controller 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.

`cp_barrier` takes as arguments a list of tables and a dictionary of user data to be saved as part of the checkpoint. Typical user data might 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. `cp_barrier` performs a global barrier and ensures that the checkpointed state is equivalent to the state of execution at the barrier.

`cp_periodic` takes as arguments a list of tables, a time interval for periodic checkpointing, and a kernel callback function `cp_callback`. 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.

2.2.6 Putting it together: PageRank

As a concrete example, we show how to implement PageRank using Piccolo. The PageRank algorithm [20] 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.

The complete PageRank implementation in Piccolo is shown in Figure 2.4 (we use Python syntax for brevity). 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

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

def PRKernel(curr, next, graph):
  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(Master m):
  # graph is partitioned by site
  graph = m.load("/dfs/graph")

  curr = m.create(PageId, double, graph.num_partitions(),
                  sum_accum, partition_by_site)
  next = m.create(PageId, double, graph.num_partitions(),
                  sum_accum, partition_by_site)

  # make sure partitions are aligned
  m.group(graph, curr, next)

  meta = master.restore_from_checkpoint()
  if meta:
    last_iter = meta["iteration"]
  else:
    last_iter = 0

  # run 50 iterations
  for i in range(last_iter, 50):
    m.run(graph, PRKernel, args=(curr, next, graph))

    # checkpoint every 5 iterations, storing the
    # current iteration alongside checkpoint data
    if i % 5 == 0:
      m.cp_barrier(tables=curr, {iteration=i})
    else:
      m.barrier()

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

Listing 2.4: PageRank Implementation

file system. For link graphs too large to fit in memory, Piccolo also supports a read-only 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 (`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, `partition_by_site`, 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.

2.3 System Design

Piccolo's execution environment consists of one *master* process and many *worker* processes, each executing on a potentially different machine. Figure 2.1 illustrates the overall interactions among workers and the master when executing a Piccolo program. As Figure 2.1 shows, the master executes the user control thread by itself and schedules kernel instances to execute on workers. The master also decides how table partitions are assigned to workers. Each worker is responsible for storing assigned table partitions in its memory and handling table operations

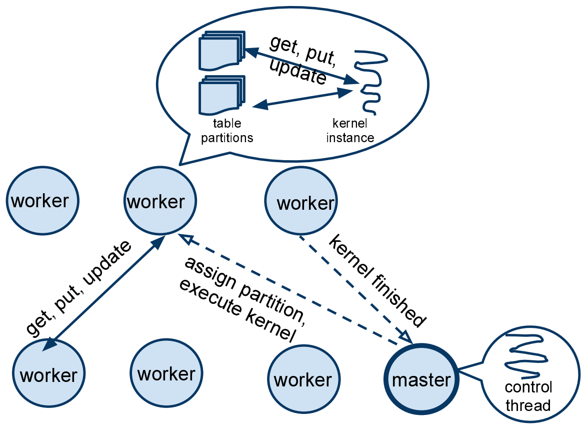

Figure 2.1: Master-worker interaction during a Piccolo program.

associated with those partitions. Having a single master does not introduce a performance bottleneck: master operations are all very light-weight, and 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 user-defined controller function. 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. When `run(table, kernel_fn)` is invoked the master prepares m tasks, one for each partition of the table. The master schedules these tasks for execution on workers based on user's locality preferences. Workers run a single kernel instance at a time and notifies the master when a kernel instance completes; this process continues until every kernel instance is finished. 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 design. First, how can we assign tasks in order to minimize the amount of time wasted waiting for global barriers? This is particularly important for iterative applications that incur a global barrier at each iteration of the computation. The second challenge is how we can perform efficient checkpointing and restoration of table state. In the rest of this Section, we detail how Piccolo addresses both challenges.

2.3.1 Load-balanced Task Scheduling

We first describe the basic scheduling algorithm without load-balancing. 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 filesystem, the master chooses an assignment that minimizes inter-rack transfer while keeping the number of partitions roughly balanced among workers. When running kernels, the master schedules kernel instances to ensure access locality: namely, it assigns task i to execute on a worker storing table 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 run-time needs to load-balance kernel executions after 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 [47] 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-balancing 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 preferences, worker w_{idle} needs to transfer one or more table partitions 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_{begin} 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_{begin}, 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_{commit} to w_a and w_b to complete migration. Upon receiving M_{commit}, 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.

This table 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.

2.3.2 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 ensure low overhead, the run-time must carry out checkpointing in the face of actively running kernel instances or the control thread. Fortunately, an algorithm already exists which allows Piccolo to provide both of these properties: the Chandy-Lamport distributed snapshot algorithm [26]. Piccolo uses the Chandy-Lamport (CL) algorithm 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), then time is wasted writing lots of data to replay files, which also 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.

2.4 Implementation

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. SWIG [14] is used to help construct the 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.

3
Piccolo Evaluation

3.1 Applications

In addition to PageRank, we implemented a variety of other applications in Piccolo, 4 of which we describe here: a distributed web crawler, k-means, n-body and matrix multiplication. We also describe how Piccolo's programming model enables efficient implementation for these applications.

3.1.1 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,

```
#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)
```

Listing 3.1: Snippet of crawler implementation

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.

Our implementation uses three co-located tables:

- The *url_table* stores the crawling state for each URL: one of *ToFetch, Fetching, Blacklisted, Done*. 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. Listing 3.1 shows the simplified crawler kernel (omitting details for processing robots.txt and capping per-site download rate). Each kernel scans its local *url_table* partition to find *ToFetch* URLs and processes them using a pool of helper threads. As all three tables are partitioned according to the *partition_by_site* 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 Listing 3.1, where kernels re-discovering an already-fetched URL u can simply attempt to update u's state to *ToFetch* without checking the current value; the correct behavior is ensured by the accumulation function.

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."

3.1.2 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.

3.1.3 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 [78], 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.

3.1.4 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 $C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$.

3.2 Overview

We tested the performance of Piccolo on the applications described above. 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 [92], 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.

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

Table 3.1: Application input sizes

3.3 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 3.1 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[18]. Specifically, we 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.

3.4 Scaling Performance

Figure 3.1 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

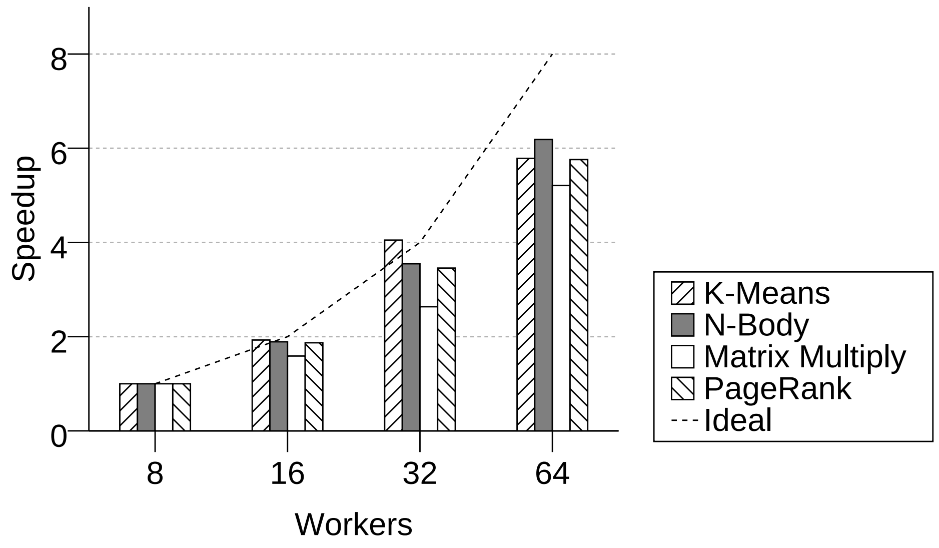

Figure 3.1: Scaling performance (fixed default input size)

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 § 3.7).

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 with N. As Figure 3.2 shows, the achieved scaling for all applications is within 20% of the ideal number.

3.5 EC2

We investigated how Piccolo scales with a larger number of machines using 100 EC2 instances. Figure 3.3 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

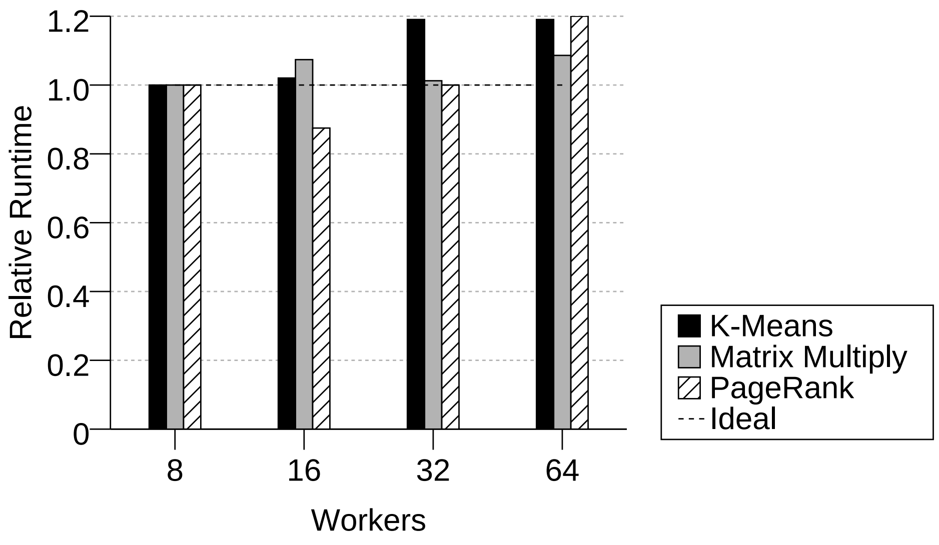

Figure 3.2: Scaling input size.

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.

3.6 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

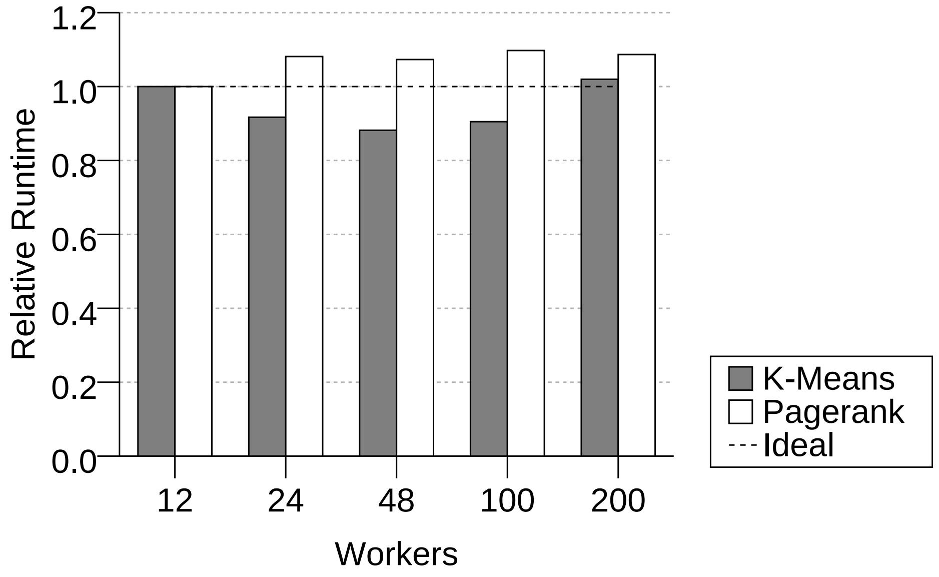

Figure 3.3: Scaling input size on EC2.

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 3.4 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,

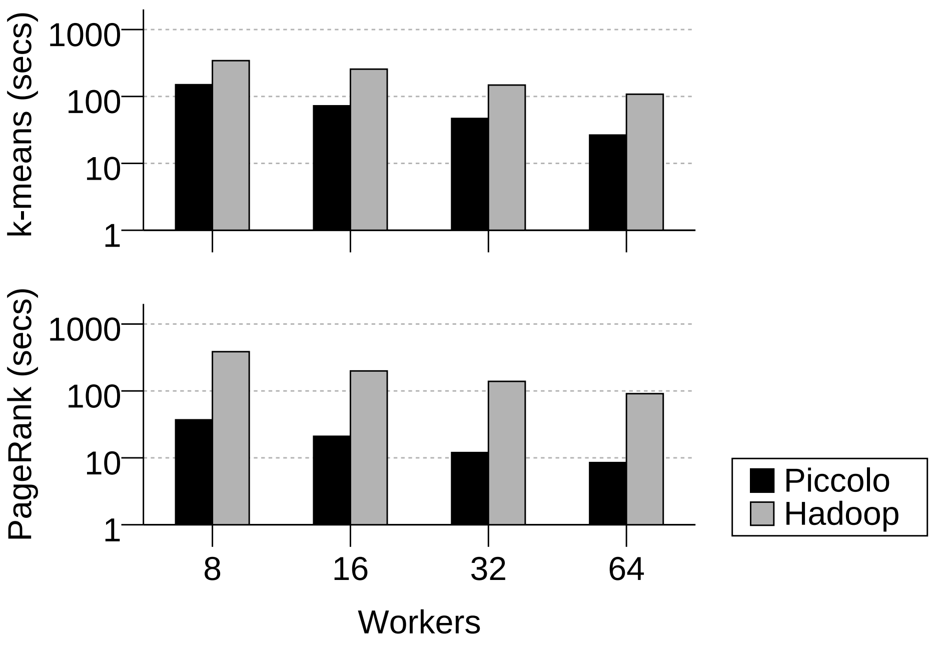

Figure 3.4: PageRank and *k*-means in Hadoop and Piccolo (fixed default input size).

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 [6]. 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 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 3.5 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 outperformed the MPI version in experiments with more workers. Upon inspection, we found that

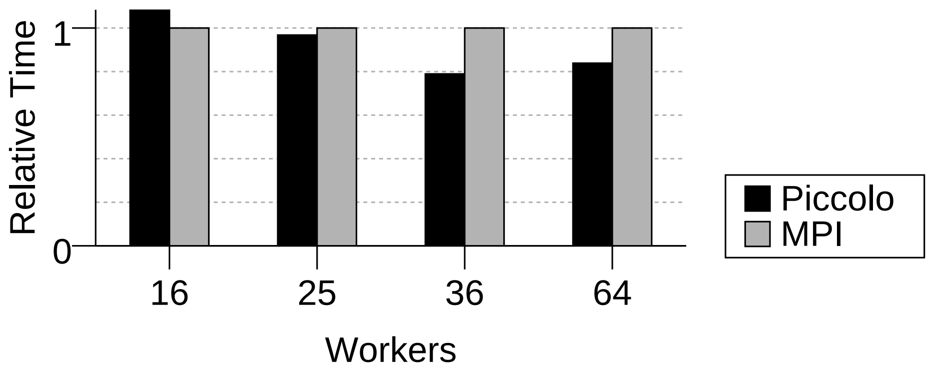

Figure 3.5: Runtime of matrix multiply, scaled relative to MPI.

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.

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

The results of these tests are shown in Figure 3.6. 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.

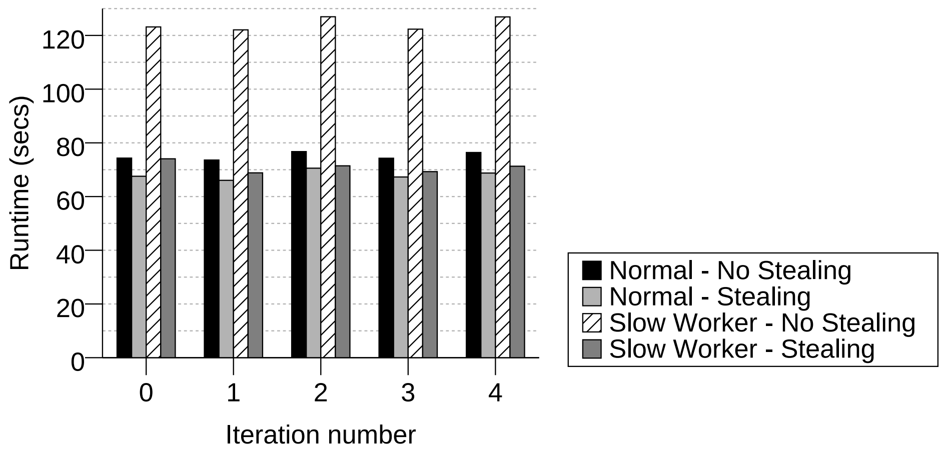

Figure 3.6: Effect of Work Stealing and Slow Workers

3.8 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 3.7 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 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

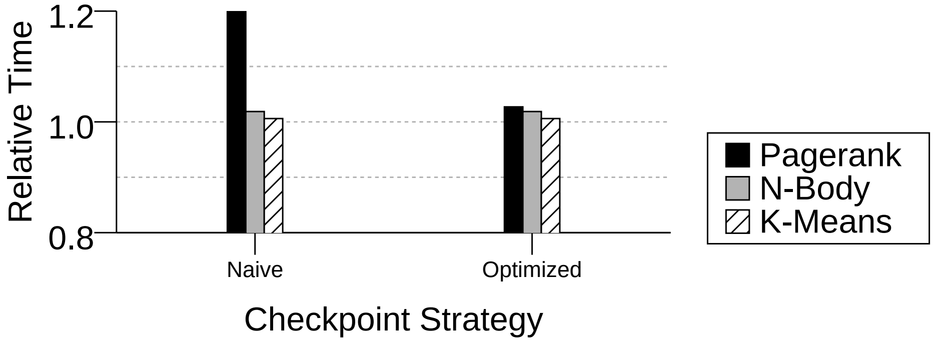

Figure 3.7: Checkpoint overhead scaled relative to without checkpointing.

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 3.8. For well maintained datacenters that we are familiar with, the average machine uptime is typically around 1 year. For these datacenters, the global checkpointing mechanism can efficiently scale up to a few thousand machines.

3.9 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 3.9 shows the crawler's web page download 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. Therefore, its throughput scales approximately linearly with N. At N=32, the crawler download throughput peaks at ~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 [86]. However, our Piccolo-based crawler could potentially scale to even higher download rates despite

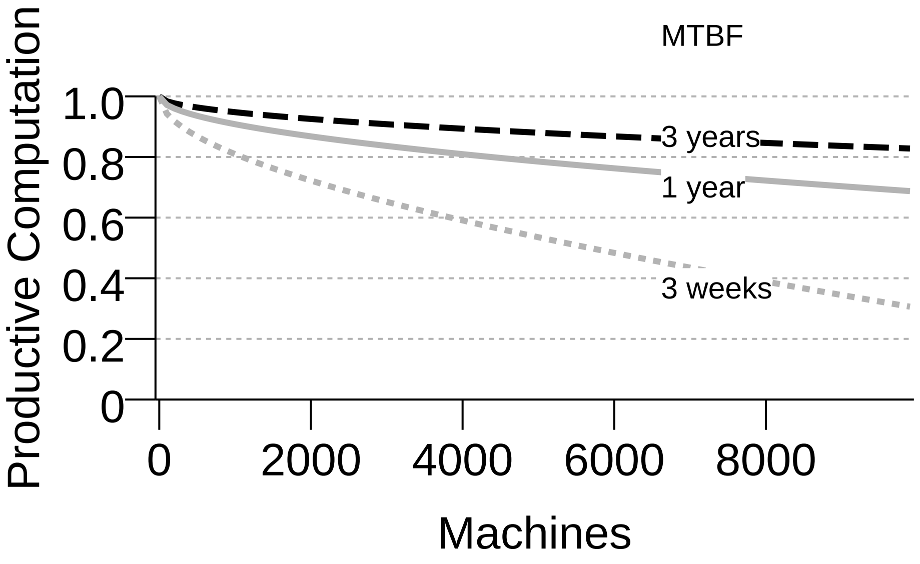

Figure 3.8: Expected scaling for large clusters.

being built using Python.

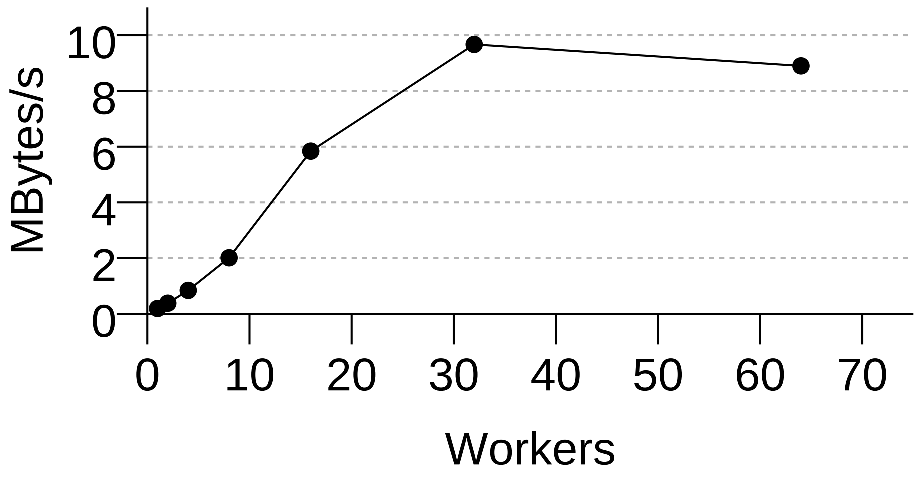

Figure 3.9: Crawler throughput

4

Spartan Design

4.1 Overview

Piccolo proved to be a flexible and efficient platform for writing many in-memory applications. But writing an efficient Piccolo program still required a significant amount of “boilerplate” effort to allocate and manage tables, and to define the accumulators and kernels required for a computation.

We designed Spartan to address this boilerplate problem by providing users with a high-level language which hid most of the tedium associated with writing raw Piccolo programs. Spartan is a distributed array language which provides used all of the expected array language operations such as array-wide operations, slicing, filtering, and reductions. Spartan uses lazy evaluation, a common intermediate form and an extended version of the Piccolo accumulative backend to enable an efficient implementation of many interesting array language applications.

In this chapter we demonstrate the utility of array languages and the challenges associated with distributing them. We then describe the model, design and implementation of the Spartan language. Finally we demonstrate the performance of Spartan for a number of interesting array problems.

4.2 Motivation

To illustrate how array languages can be useful for data analysis problems, consider the following scenario: we are a financial analyst (a "quant"), and we are interested in determining if change in the *spread* (the difference between the cost of buying something and the price at which it can be sold) can help up predict future prices. Using an array language like NumPy [64], we can very quickly sketch out a linear predictor for prices and test our theory, as shown in Listing 4.1.

```
# ask - price you can buy at
# bid - price you can sell at
# t - how far forward to predict
def predict_price(ask, bid, t)
  # element-wise difference
  spread = ask - bid

  # element-wise average of ask and bid
  midprice = (ask + bid) / 2

  # slices allow for cheaply extracting parts of an array
  d_spread = spread[t:] - spread[:-t]

  # find prices ‘t‘ steps in the future of d_spread
  d_spread = d_spread[:-t]
  future_price = midprice[2*t:]

  # compute a univariate linear predictor
  regression = mean(future_price / d_spread)
  prediction = regression * d_spread

  error = mean(abs(prediction - future_price))
  return error
```

Listing 4.1: Spread Prediction Example

This same example is equally easy to write in R [83], Matlab [57], Julia [4] or any other array-oriented language. These languages are all frequently used for data analysis problems and the types of in-memory problems which Piccolo targets: unfortunately, all of these languages only run on a single machine, which greatly restricts their applicability to larger problems. Having a distributed implementation of such a language would greatly extend its reach.

4.3 Array Language Features

Array languages provide several compact and efficient ways to express operations on arrays. As you might expect from their name, they are distinguished from other languages by their flexible slicing mechanism and array-wide operations. Below we give a brief overview of the some of the important features of array languages: a more complete discussion can be found in Rubinsteyn's thesis [77].

Builtin Functions

Array languages come builtin with an extensive library of functions for array manipulation and analysis: part of the appeal of these languages is having powerful analysis tools "at one's fingertips". These include basic concepts such as adding and subtracting arrays, reductions over arrays (*sum, min, max, mode*), functions to locate regions of interest (*where, nonzero, argmin, argmax*), and common array operations such as matrix multiplication. Also important are more involved operations such as matrix factorization, Fourier transformations and eigenvalue analysis. NumPy has over one hundred builtin operations, many of which are commonly used in array programs. Array-wide operations and reductions are particularly common.

Array-wide operations: Array languages extend common arithmetic and boolean operations to work on arrays ($a + b$ adds a and b element-wise). When one array has fewer dimensions than another, *broadcasting* is used to extend the smaller array, this generalizes the natural idea of adding arrays and scalars (e.g. $a + 1$).

```
# element-wise addition
c = array([1,2]) + array([3,4}) # c = [4, 6]

# broadcasting scalar values
d = 5 * c # d = [20, 30]
```

Reductions: Reduction operations are used to extract global information from arrays; these include operations such as sum, min, max, argmin (the index of the element with the minimum value), argmax, etc. Reductions can be performed over the entire array (returning a single scalar value), or over an axis of the array, in which case they collapse the array along that axis. Some operations we can perform using these operators include:

```
# create a new random 100x100 array
array = rand((100, 100))

# sum every element
array.sum()

# sum over rows
array.sum(axis=0)

# sum over columns
array.sum(axis=1)

# index of the minimum value
array.argmin()
```

Slicing

Array languages allow multiple array elements to be selected simultaneously through the use of multi-dimensional slice expressions. A slice consists of a starting offset, and ending offset and a step size. If a slice is omitted for a dimension, than all of the elements in that dimension are selected. Some examples of slicing are shown below:

```
array[4]     # 5th element of array
array[:20]   # first 20 elements of array
array[-20:]  # last 20 elements of array
array[::2]   # every other element
array[::-1]  # array reversed
```

Slicing is not limited to a single dimension. For example the following code computes a simple blurring of an image:

```
blur = (img[0:-2,0:-2] +
        img[1:-1,1:-1] +
        img[1:-1,0:-2] +
        img[0:-2,1:-1]) / 4
```

An important feature of slices is that they are arrays themselves in every respect: any operation that can be performed on a top-level array can also be performed on a slice.

Fancy indexing

Slicing allows users to extract fixed chapters of an array; "fancy" indexing can be viewed as extending slicing by making it data dependent. Fancy-indexing allows indexing an array with a mask or an array of indices. This allows users to re-order and filter arrays based on data:

```
nonzero = array[array > 0]

# shuffle two arrays in the same way
rand_idx = random(x.shape)
x = x[rand_idx]
y = y[rand_idx]
```

4.4 Challenges

If a distributed array language is to be successful, it needs to support the full set of functions expected by users and it should be as fast (or nearly as fast) as hand-written solutions. This implies a number of challenges:

Builtin functions Array programs depend on a large library of builtin functions, and the writing of each of these functions in isolation would be a very time-consuming process. In addition, users often make use of additional extension libraries to handle problems not addressed by the core language. Many of these extensions are written in C++ or Fortran: this makes them effectively impossible to automatically distribute. In order to address these issues, we need a way to simplify writing builtin functions and enable users to create their own extensions easily.

Temporary elision The idiomatic style of array programs leads to the creation of a large number of expensive temporary arrays if executed naïvely. (This is an issue even for single machine array languages, and has led to the creation of tools like numexpr [8]). For example, consider the following line from our `predict_price` application:

```
midprice = (ask + bid) / 2
```

In this example, without optimization, 2 expensive temporary arrays will be generated in the process of computing *midprice*. Since this type of element-wise operation is very common in array programs, we need some way optimize away the creation of these temporaries.

Slicing In single-machine array languages, slices do not require any data to be copied, and are therefore very cheap. As slices are used extensively in array applications, a distributed implementation should offer the same no-copy guarantee.

4.5 Overall Approach

The goal of Spartan is to overcome the challenges listed above while still providing the same level of convenience found in a single-machine programming language. To accomplish this, Spartan adopts a layered approach, which splits execution into frontend and backend steps.

The execution of an array program in Spartan consists of 4 parts: *capturing* user code into expression graphs, lowering expression graphs to a common set of *high-level operators*, *optimizing* the high-level operation graph, and *executing* operations on the backend.

Array-language frontend: Users interact with Spartan using a variant of the NumPy array language. Spartan uses a *lazy evaluation* strategy to avoid evaluating user expressions until necessary; the result of this lazy evaluation is an expression graph. By evaluating expressions lazily, Spartan can avoid creating temporary arrays for expressions which are never used; it also allows for optimizations to be performed on the expression graph before evaluation.

Intermediate representation: Rather than directly implementing every array possible op-

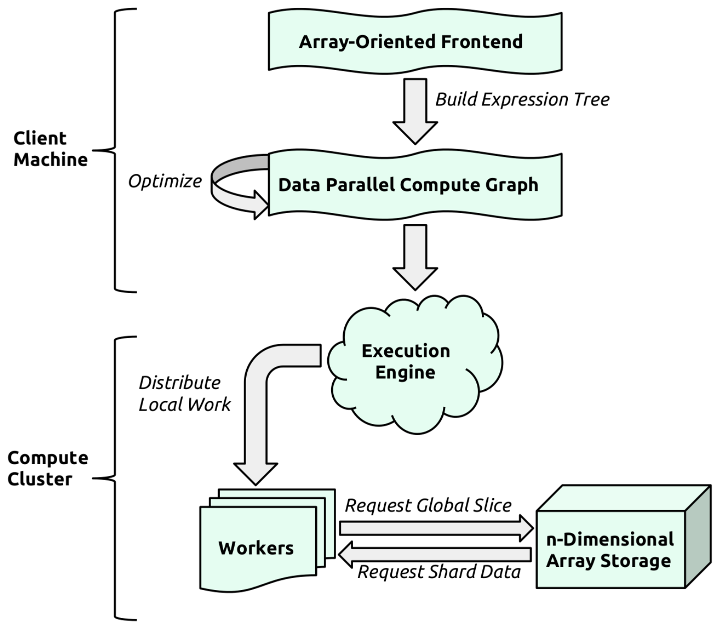

Figure 4.1: Spartan System Design

eration (which would both reduce expressiveness and require many distinct implementations), Spartan uses a common set of *high-level operators* to capture common patterns for array operations: user expressions are lowered to applications of one or more of these operators. These operators include the usual suspects (*map* and *reduce*), as well as additional operations for prefix scans, slicing and filtering.

Optimization: The use of high-level operators not only simplifies the implementation of many functions, it also provides a consistent *intermediate representation.* The use of an intermediate form allows a number of simple, yet powerful *fusion* optimizations to be applied. These optimizations combine expression nodes together using algebraic rewrite rules, effectively eliminating most unnecessary temporary variables.

Distributed array backend: Once an expression graph has been optimized, it can be executed against the distributed array backend. The backend "understands" multi-dimensional arrays natively, which enables efficient fetching and updating of arbitrary array regions, and zero-copy slices. The backend also exposes methods to parallelize operations over arrays, and leverages the *accumulator* concept from Piccolo to allow for consistent concurrent updates to

```
# x: N (examples) x 100 (features)
x = load('examples.bin')
# y: N x 1
y = load('predictions.bin')

w = random((100, 1))
epsilon = 1e-6
for i in range(100):
  yp = dot(x, w)
  grad = sum(x * (yp - y), axis=0)
  w = w - grad * epsilon
```

Listing 4.2: Linear Regression

arrays.

This layered approach has many advantages. First, it enables a very modular design, simplifying the task of adding new functionality. Users can update Spartan without needing to understand or modify the backend: they need only express their extension using one or more high-level operators. Moreover, such extensions will automatically be optimized by the frontend. It also greatly simplifies writing new optimizations: optimization writers only need to consider a small number of high-level operators, rather then the entire set of user functions.

The remainder of this chapter details the programming model and design of the Spartan system.

4.6 Detailed Design

Listing 4.2 shows a simple linear regression application written in Spartan. The remainder of this chapter will show how Spartan transforms and executes such an application.

4.6.1 Distributed Arrays

The fundamental data structure for Spartan is the *tiled array*. A tiled array is an n-dimensional array which has been divided into one or more contiguous, non-overlapping pieces (tiles); tiles are spread across machines in a cluster. In Spartan, tiles are uniquely identified by their *extent*: the upper-left and lower-right corners of the tile in an array. The Spartan backend handles storage and processing of these arrays.

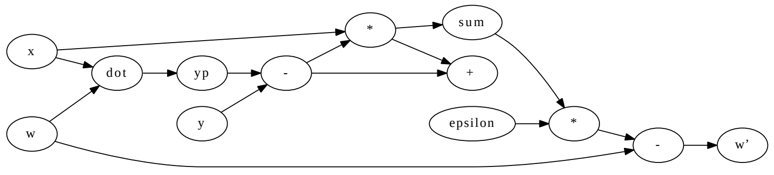

Figure 4.2: Linear Regression DAG

4.6.2 Programming Model

Programs in Spartan are written using a variant of the NumPy array language; NumPy itself is an extension to the Python programming language. As with NumPy, Spartan expressions can be freely inter-mixed with normal Python code.

To create a new array, users use one of the array creation options: bulk loading data from a distributed file system, copying a local array into a distributed one, or using one of the builtin array creation routines (e.g. *empty*, *zeros*, *ones*, *rand*). Each of these returns an array expression which can then be manipulated. Once an array has been loaded or created, users can manipulate it as if it were an ordinary NumPy array – slicing, broadcasting, basic arithmetic operations and reductions all work as expected.

4.6.3 Lazy Evaluation

Spartan uses lazy evaluation to capture user expressions before evaluating them. This results in a an expression graph, where each expression in such a graph represents a high-level user expression.

This lazy evaluation strategy is critical: it allows Spartan to inspect and optimize the resulting expression graph before running operations. The expression graph resulting from one execution of the linear regression loop is shown in Figure 4.2.

Expressions are evaluated only when *forced*: this occurs in a few situations: when a value is used for flow control (*if* or *while* statements), when a user explicitly calls the `force` function, when a value must be returned to the user for display (e.g. `print(sum(a))`), or when the user requests an array be serialized to disk. It should be noted that this can implicitly cause a form of loop unrolling to take place: as long as there is no data dependent control flow, expression graphs will simply continue growing. In theory this could cause memory issues (with very large

Operator	Usage
empty(shape, dtype)	Array initialization routines.
map(map_fn, array1, array2...)	Arithmetic operators, builtins.
filter(fn_or_mask, array)	Boolean indexing.
reduce(reducer_fn, accum_fn, array, axis)	Sum,min,max,etc.
scan(fn, array, axis)	Cumulative sum.
shuffle(map_fn, accum_fn, array)	Generic operations.
slice(array, region)	Slicing
stencil(array, filter, stride, axes)	Image processing.

Table 4.1: High-level operators

loops); if this becomes an issue, a user can always explicitly force an expression to collapse the graph.

Expressions in Spartan are *immutable*. This restriction simplifies program analysis and optimization, by ensuring that expression graphs are acyclic. This immutability is in contrast to traditional NumPy arrays, which can be modified. As in practice, most array programs do not mutate arrays, we find this to be an acceptable limitation.

4.6.4 High-level Operators

The next step in converting an expression graph to an executable form is to *lower* user expressions to a consistent set of high-level operators. This lowering operation has two advantages: it greatly simplifies writing many common functions, as almost all can be expressed using one of the high-level operators. Second, using a set of common operations enables optimizations to be applied to the graph, improving performance.

The list of high level operators and their example usage is given in Table 4.1. All operators which take a function to map over data (map, reduce, scan and shuffle) expect a function which can process an entire tile of data at a time. We give a brief description of each operator here:

- `empty(shape, type)` Create a new, empty array of the given shape and data type (float, int, etc.).
- `map(fn, array...)` Map a function over one or more arrays, producing a new array as a result. Input arrays are broadcast to ensure they have the same shape, and a join is performed across all of the arrays.
- `reduce(reduce_fn, accum_fn, array, axis)` Reduce an array over a given axis.

Operation	Translation
yp - y	`map(lambda a, b: a - b, [yp, y])`
sum(z, axis=0)	`reduce(lambda t: sum(t, axis=0), add, z, 0)`
a[0:10, :]	`slice(a, [0:10])`
val[mask]	`filter(mask, val)`

Table 4.2: Example applications of operators

This operator is used to implement all of the builtin reduction functions (min, max, argmin, argmax, sum, mean, and mode).

- `scan(reduce_fn, accum_fn, array, axis)` Compute a cumulative reduction over an axis of an array.
- `slice(region, array)` Extract a region from array.
- `stencil(array, filter, stride, axis)` This is an example of a more specialized operator. The stencil scans a window across an array along one or more axes and multiplies each window with one or more *filters*. This is a basic building block of many image processing algorithms and convolution neural networks.
- `filter(fn_or_mask, array)` Apply a filter to array. The filter can be either a function which returns a boolean value, or a boolean array.
- `shuffle(map_fn, accum_fn, array)` This operator serves as a sort of "catch-all" for functions that are not addressed well by the other operators. The shuffle operator allows a user to create a target array of an arbitrary shape. The mapper_fn is called for each *tile* of the source array and returns a list of (region, data) pairs; these are then accumulated into the target array using the supplied `accum_fn`. It is used for the matrix multiplication and array reshaping builtins, and also as a building block for many user-defined operations.

At first glance, the set of operators available may seem overly verbose: we could implement almost all of them using just the shuffle operator. Making these operations explicit has two important benefits. First, using separate operators allows Spartan to provide higher-performance implementations for operations such as slice and stencil. Additionally, the use of separate operators enables the map and reduce fusion optimizations which are critical for performance.

With these high level operators defined, the process of lowering expressions is straightforward. A few example translations are shown in Table 4.2.

4.6.5 Optimization

After the initial compilation step, we are left with a DAG of high-level operations. While this can be immediately executed against the backend, we can greatly improve application performance by applying a number of optimizations to the graph at this point.

As is common in most compiler frameworks, each Spartan optimization is structured as a *pass*: an optimization takes as input a DAG of operations, and returns (a potentially unmodified) DAG which computes the same final result. Spartan uses a number of optimization passes. Fusion rules such as *map fusion* and *reduce fusion* combine consecutive operations together into one aggregate operation, which reduces temporary variables. This type of optimization is also found in lazy functional languages such as Haskell [2]. The *cache collapsing* operation prevents fusion operators from re-evaluating expressions which have already been evaluated. *Parakeet generation* executes after the fusion optimizations have been performed. It inspects fused operations and compiles the fused operations into equivalent Parakeet [77] code.

Map-map fusion Idiomatic use of array expressions results in large numbers of *temporary arrays*. In the linear regression application, we can see an example of this during the gradient calculation `grad = sum(x * (yp - y))`. Without optimization, this will result in the creation of 2 temporary arrays before we begin evaluating the sum. An extreme case of this can be seen in the Black-Scholes implementation found in the evaluation chapter; in this case map-map fusion results in 33 fewer temporaries.

The map folding optimization eliminates these common temporaries by identifying and combining consecutive map tasks together. Since map operations preserve the shape of the input, fusion is a straightforward process: when the optimizer encounters a sub-tree of the form: $map(g, map(f, x))$, it replaces it with a single equivalent map. The replacement rule is:

$$map(g, map(f, x)) \rightarrow map(g \odot f, x)$$

Reduce-map fusion Fusing map and reduce operations is similar. If the input to a reduction is a map operation, then the map operation can be performed locally as part of the reduction, instead of creating a temporary. The replacement performed is:

$$reduce(g, map(f, x)) \rightarrow reduce(g \odot f, x)$$

Collapsing Cached Expressions This optimization implements a form of common subexpression elimination (CSE). When an expression is evaluated by Spartan, the result is cached until the expression itself is garbage collected (when it is no longer referenced by a user variable or another expression graph.) This optimization replaces previously evaluated expressions in the expression graph with their computed value. Without this optimization, the map and reduce fusion optimizations can end up fusing already evaluated expressions, resulting in redundant work being performed.

Parakeet Code Generation The fusion operations greatly reduce the number of distributed temporaries created, but the local evaluation on each worker will still result in *local* temporaries being created. This is a known performance issue with NumPy, and projects such as numexpr [8] have been written to help address it. Unfortunately, tools such as numexpr are rather limited in their expressiveness; they cannot handle the full set of local expressions output from Spartan. Recently, the Parakeet [77] optimizing compiler has been released. Parakeet supports a large subset of NumPy and can target single core, multi-core and (experimentally) GPU accelerators.

The Parakeet optimization pass compiles fused map and reduce operations to Parakeet compatible Python code; this code is then compiled to C or CUDA by Parakeet. By leveraging the Parakeet GPU backend, Spartan can transparently run array language code on a distributed GPU cluster.

Non-Idempotent Operations The fusion operations all assume that expressions are idempotent, and can be evaluated multiple times in the graph affecting correctness. This is not true for some operations (e.g *random*), in which case fusing must be disabled. Currently this handled by explicitly black-listing non-idempotent functions.

4.6.6 Backend

Once the optimization passes are finished, the optimized DAG is ready for execution on the backend. The backend of Spartan is responsible for storing and accessing arrays and executing operations provided by the frontend. The basic design of the Spartan backend is similar to that of

```
Master:
  new_array(shape, type, tile_hint)

Array:
  fetch(selector)
  update(extent, data, accum)
  map(fn)

Tile:
  fetch(selector)
  update(extent, data, accum)

make_slice(array, region)
broadcast(array, shape)
```

Listing 4.3: Backend API

Piccolo, but whereas Piccolo manages key-value tables the Spartan backend manages distributed arrays; arrays have a slightly more complex interface in order to efficiently handle indexing and slicing operations. The backend API is shown in Listing 4.3.

The backend consists of a *master*, which manages array metadata and coordinates execution and *workers* which hold array data and execute individual tasks. Only the master process can create new arrays or map a function over an array; arrays can be retrieved and updates by the master or any worker.

Tiles At the lowest level, the Spartan backend tracks individual *tiles* of an array. Tiles hold a contiguous rectangular region of array data, and are spread across workers. Tiles have 2 representations: dense and sparse. The sparse representation holds a dictionary from array indices to values. The dense representation holds an array of values and a mask indicating which values have been initialized.

The `fetch` method returns data from a tile matching a *selector*. The *selector* can be either a function (which takes the tile data and returns a result), a slice to be applied to the local tile data, or a boolean or index array.

The `update` method *merges* the current data for a tile with that of the update, using the specified accumulation function. As users are allowed to update arbitrary portions of a tile, a mask is used for dense tiles to distinguish between uninitialized slots (which are replaced with the new data) and previously initialized data (which are combined using an accumulation function).

Creating arrays Many primitives need the ability to create a new array; this is provided by the `new_array` operation. Given a shape and a data type (e.g. float, double, int, etc), this operation creates a new distributed array and returns a reference to the user. If a *tile_hint* is given, each tile will be of shape *tile_hint*. (Tile hints may be supplied by the user, or inferred from the type of operation being performed). If no tile hint is provided a heuristic method is used which attempts to construct a generically "good" tile shape. Tiles are uniquely identified by their extent: this is the upper-left and lower-right corners where the tile resides in the array.

Spartan uses reference counting on the master to automatically garbage collect unused arrays; when an array is collected, the master sends a request to each worker to remove any tiles belonging to the dead array.

When an array is created, tiles are uninitialized (the data pointers are set to NULL). This allows empty tiles to be transmitted and stored efficiently, and speeds up creating sparse or empty arrays. Users specify the type of tile to be be used when creating arrays; by default Spartan uses the dense tile format.

Tile alignment A very frequent operation in array applications is a map over multiple arrays: element-wise operations such as $a + b$, for example, result in such a map. For efficiency, the backend must ensure that these operations do not result in excess communication. The backend handles this by ensuring that if any 2 arrays A and B have the same shape and tile layout, then each tile t from array A and B will share the same worker.

Accessing and modifying arrays Arrays expose 2 methods for reading and updating entries, which are natural extensions of the individual tile operations:

The `fetch` method takes a selector and returns a local array containing the data matching the selector. As with the tile fetch operation, a selector can be a slice, or an array. A selector can span multiple tiles; in this case the overlapping region of each tile is determined and fetched, and the resulting parts are assembled together before returning to the user.

The `update` operation updates an existing array in place. As with the `fetch` operation, we first determine which tiles the given slice overlaps with. The tile data is then split into pieces and the sent to appropriate workers in parallel. Update operations re-use the accumulator concept from Piccolo to allow for buffering and to avoid the need to lock or otherwise synchronize array updates.

`broadcast` promotes a given array to have the requested shape, adding or increasing dimensions as necessary. Rather than creating a copy of the input array, broadcast returns a *broadcast object*, which mimics the distributed array interface, and translates `fetch` operations by dropping dimensions as appropriate. Broadcast objects do not support the update or map operations.

The `make_slice` function returns a *slice object* which represents a slice of an existing array. Slice objects behave like normal arrays: they can be mapped over, updated and fetched from; these operations are translated and applied to their base array.

Concurrency The `map` operator enables parallel processing over an array. The map operator invokes a user-provided function *with locality* on each tile of the array, ensuring that all operations run locally with the machine that holds the tile data. The user function is supplied with the extent currently being processed; it can use this to look up array data as required. The use of accumulation functions for array updates eliminates the need for special-purpose reduction operations or locking of array elements.

Fault Tolerance For fault tolerance, Spartan uses a the Chandy-Lamport global checkpointing protocol [27]. As most array language operations are fast to evaluate, it is generally sufficient to save intermediate array data at the application level.

4.6.7 Implementing High-level Operators

Given this set of backend functions, implementing our high-level operators is straightforward. For example, the `map` operator broadcasts input arrays to have a consistent shape, creates a new target array, and then maps over the input arrays, writing the result of the user function to the target. The full implementation of the map operator is shown in Listing 4.4.

This example also shows how the *tile_hint* parameter can be used to minimize communication. In this case, we choose to tile our output array in the same manner as our input; this ensures that all writes to our output array will be local (as the backend ensures tiles are aligned between arrays).

The `reduce` operator is similar: a new array is created to hold the output, and an accumulation function is attached to combine updates as shown in Listing 4.5. The *slice* operator simply creates a slice wrapper object, as shown in Listing 4.6.

```
def map(map_fn, [X, Y, Z]):
  # map over the largest array to minimize remote fetches (if any)
  largest = find_largest([X, Y, Z])

  # coerce arrays to have the same shape
  Xb, Yb, Zb = X.broadcast(largest), Y.broadcast(largest), Z.broadcast(largest)

  # create a target array
  out = new_array(Xb.shape, tile_hint=Xb.tile_shape)

  def _mapper(extent):
    map_result = map_fn(Xb.fetch(extent),
                        Yb.fetch(extent),
                        Zb.fetch(extent))
    out.update(extent, map_result)

  # invoke our mapper function on each tile
  largest.map(_mapper)
```

Listing 4.4: Map Implementation

```
def reduce(reduce_fn, accum_fn, X, axis):
  # create an uninitialized target array with one dimension removed
  Y = new_array(shape_for_reduce(X, axis),
               tile_hint=tile_for_reduce(X))

  def _reducer(ex):
    reduce_result = reduce_fn(fetch(X, ex))

    # compute where to put the reduced value
    output_ex = extent_for_reduce(ex, axis)
    Y.update(output_ex, reduce_result, accum_fn)

  # invoke our reducer function on each tile
  X.map(_reducer)
```

Listing 4.5: Reduce Implementation

```
def slice_op(array, region):
  return make_slice(array, region)
```

Listing 4.6: Slice Operator Implementation

4.7 Implementation

Spartan is implemented in Python and consists of approximately 6000 lines of code. Most of this is devoted to the expression language and optimizations. Spartan aims to replicate the NumPy API as closely as possible, to reduce the effort of porting existing NumPy applications to Spartan.

The original Spartan backend was built by extending Piccolo [71] tables, and inherited the Chandy-Lamport checkpointing facility from Piccolo. The backend has since been replaced, but we have not yet re-implemented the checkpointing protocol.

5
Spartan Evaluation

In this section, we implement a variety of applications in Spartan and evaluate the performance of Spartan on each. We also demonstrate how the Spartan optimizations can dramatically impact the performance of certain applications.

5.1 Test Setup

We evaluate the performance of Spartan on our local cluster. This is a heterogeneous setup of 11 machines: 6 dual-processor AMD Opteron machines with 16GB of RAM and 16 cores, and 5 dual-processor Intel Xeon machines with 8GB of RAM and 8 cores. As with the Piccolo evaluation, we create one worker process per core, and pin processes to a single core.

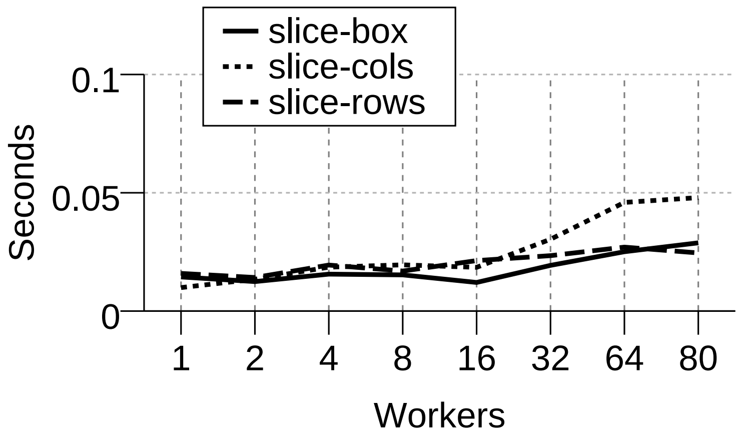

Figure 5.1: Slicing Performance

5.2 Micro-benchmarks

We begin with some micro-benchmarks which evaluate Spartan's performance for a few small tasks in isolation.

Slicing The slicing benchmarks measure the amount of time to slice out and compute the sum over 100 rows, columns or a 100x100 box of an input array of size 10000x1000**numworkers*. (The amount of work performed by the *slice-cols* operation increases with the number of workers). This simulates the common activity of a user extracting and computing some statistic over a portion of their dataset, e.g: `array[x:y, a:b].sum()`.

For this example, the array has been tiled to have contiguous rows. Even with 80 workers (a 800 million element array), these basic slicing operations are very fast, taking less than 50 milliseconds. They also scale well as the number of workers and data size increases. This makes common slicing operations suitable for use in an interactive environment.

Stencil Figure 5.2 shows the performance of the stencil operation on our local cluster, for 16 images of size 128x128, and 32 5x5 filters. We increase the size of the images along with the number of workers. After the initial jump from 1 to 2 workers (execution on one worker requires no communication), the performance remains relatively stable: this reflects the fact that very little communication is taking place, as the input and output arrays have been aligned consistently.

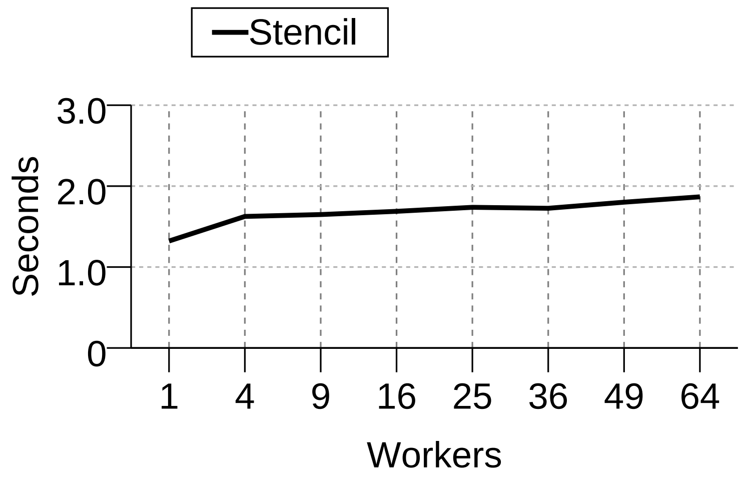

Figure 5.2: Stencil Performance

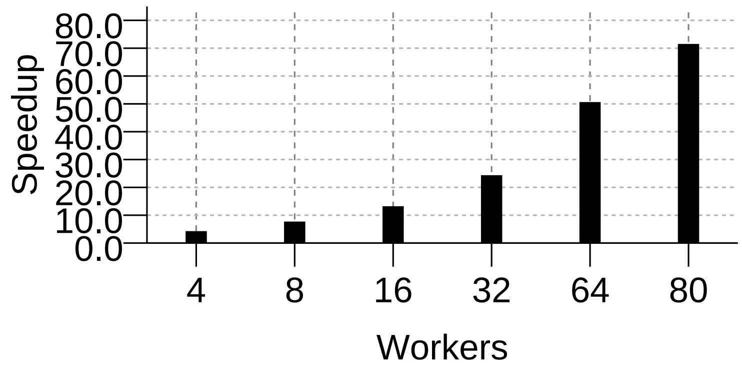

Figure 5.3: Matrix Multiplication Speedup

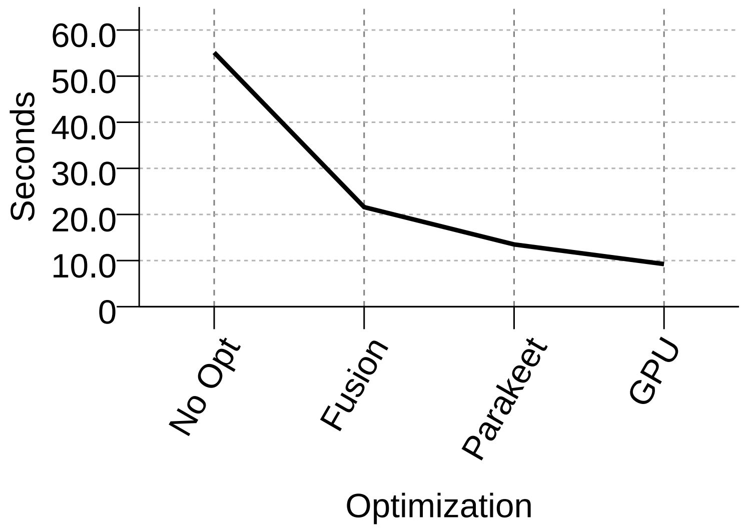

Figure 5.4: Performance of Black-Scholes with Optimizations

Matrix Multiplication Figure 5.3 shows the the speedup for matrix multiplication from one worker when multiplying 2 square 5000x5000 matrices. The speedup is close to ideal; for larger numbers of workers, system overhead begins to play a role, as the amount of local work per machine becomes very small.

5.3 Effect of Optimizations

The importance of the optimization passes is shown in Figure 5.4. This shows the time taken to calculate 100 million call options using the Black-Scholes formula on a single worker, after various optimizations have been applied. With all optimizations applied the application runs over 5 times faster than the naïve version. The fusion operators have the greatest effect; this is due to the dramatic reduction in temporary arrays needed by the computation.

These optimizations can actually have an even greater effect on shorter running or iterative applications, where the time required to create and initialize temporaries begins to dominate the actual processing time.

The remaining benchmarks in this section are evaluated with all optimizations enabled except for GPU code generation (as GPU code generation is yet not implemented for all high-level operators).

Application	Minimum input size	Maximum input size
Linear Regression	50M examples, 10 dimensions	4B examples
k-means	2.5M points, 1000 clusters	200M points
Black-Scholes Option Pricing	10M options	800M options
Price Change	10M prices	800M prices
PageRank	3M pages	200M pages

Table 5.1: Application input sizes

5.4 Applications

We implemented a number of applications using Spartan, including linear regression, k-means clustering, a convolution neural network and PageRank. When evaluating these applications, we scale the dataset size with the number of workers, keeping the amount of work performed by each worker the same. Ideal scaling would result in the same runtime regardless of the number of workers. The dataset sizes we used to test each application are shown in Table 5.1.

5.4.1 Linear Regression

This is the linear regression application shown in Listing 4.2. As this application uses dense tiles, Spartan can utilize highly optimized local libraries such as Atlas [30] for performing many of the operations. This is a relatively "easy" application: most of operations are element-wise and require no communication, with the exception of the final summation.

The scaling performance for linear regression is shown in Figure 5.5. For comparison, we show the performance of Spark [93], as derived from the Spark paper (we intend to reproduce the exact times on our cluster shortly). Spartan is approximately 5-6 times faster than Spark on this benchmark; this is due to the efficiency gains of performing large, tile-wise operations.

5.4.2 Black-Scholes

The Black-Scholes algorithm is used often in financial applications for pricing put and call options. For Spartan, we compute the put and call price given an array of stock information. This involves mapping a complex expression against a number of input arrays, and provides a dramatic illustration of the importance of the map-fusion operator: without optimization, evaluating an option price requires 34 separate kernel evaluations; map-fusion reduces this to

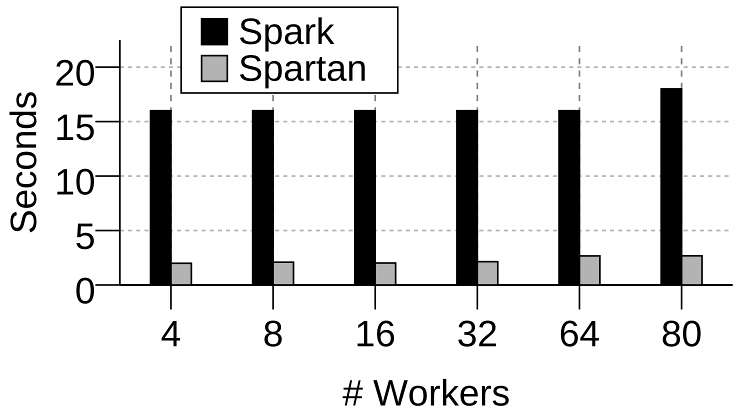

Figure 5.5: Scaling Performance for Linear Regression vs. Spark

1. The code for this application is shown in Listing 5.1. The scaling performance is shown in Figure 5.6. This application consists of only maps over the data; it thus does not require any cross-worker communication. The slight performance drop as the number of workers increases is due to cache and memory conflicts between cores as machines become more heavily loaded.

```
def black_scholes(current, strike, M, R, V):
  d1 = 1.0 / (V * sqrt(M)) *
    (log(current / strike) +
     (R + V ** 2 / 2) * M)

  d2 = d1 - V * M

  call = cdf(d1) * current -
         cdf(d2) * strike * exp(-R * M)

  put = cdf(-d2) * strike * exp(-R * M)
        - cdf(-d1) * current
  return put, call
```

Listing 5.1: Black-Scholes

5.4.3 k-means

k-means clustering is an example of *expectation-maximization*, a popular unsupervised machine learning technique. k-means attempts to find good clusters for a number of n-dimensional points. k-means can be implemented in an array language by combining the argmin and matrix multiplication operators [5]; our implementation does not take this approach, as it requires sig-

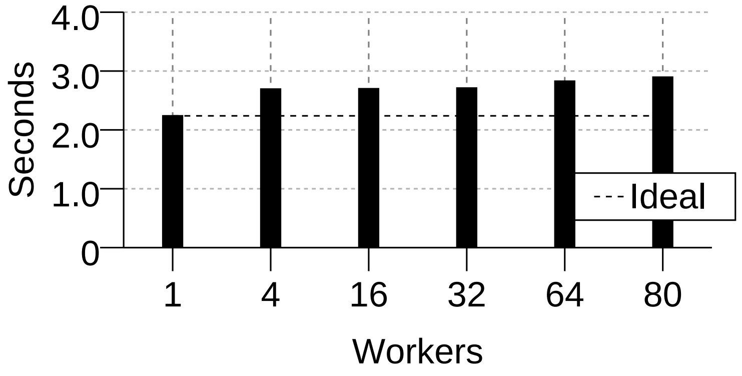

Figure 5.6: Scaling Performance for Black-Scholes

nificantly more memory to compute. Instead, we use a custom operator which maps over the points array and finds the nearest cluster for each point. It then aggregates a local set of updates for the cluster array. This is similar to the original Piccolo implementation. With the future addition of optimizations for combining matrix multiplication and reductions, this can be greatly simplified. Listing 5.2 shows the implementation.

The performance of this application is shown in Figure 5.7. As most of the work is performed locally, we see the same scaling as for our Black-Scholes application.

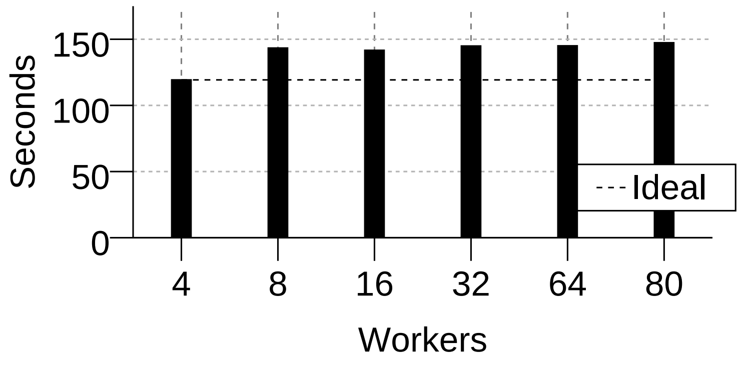

Figure 5.7: Scaling Performance for k-Means

```
def _find_clusters(inputs, ex, d_pts, old_centers, new_centers, new_counts):
  centers = old_centers.fetch_all()
  pts = d_pts.fetch(ex)

  # find the closest centers for our local points and
  # sum up their positions and counts
  closest = _find_closest(pts, centers)

  l_counts = zeros((centers.shape[0], 1))
  l_centers = zeros_like(centers)

  for i in range(centers.shape[0]):
    matching = closest == i
    l_counts[i,0] = matching.sum()
    l_centers[i] = pts[matching].sum(axis=0)

  # update centroid positions
  new_centers.update(extent.from_shape(new_centers.shape), l_centers)
  new_counts.update(extent.from_shape(new_counts.shape), l_counts)
  return []

def em_step(pts, centers):
  new_centers = ndarray((num_centers, num_dim))
  new_counts = ndarray((num_centers,1))
  shuffle(pts, _find_clusters, pts, centers, new_centers, new_counts)
  return new_centers / new_counts
```

Listing 5.2: Spartan k-Means Implementation

```
def find_change(prices, threshold=0.5):
  diff = abs(prices[1:] - prices[:-1])
  return prices[diff > threshold]
```

Listing 5.3: Price fluctuation

5.4.4 Price-change Calculation

Another example motivated by financial applications; this application returns positions in an input array where the array changes by more than a certain threshold, which might be used when trying to identify interesting points in a time series. This leverages the filter and scan operators. The code for this application is shown in Listing 5.3.

A naïve implementation of this benchmark would require a reduction over the *diff* array in order to compute how many prices changed, followed by a map over the prices array to copy out the changed prices. Spartan leverages the it's builtin support for masked tiles to avoid these expensive operations. The performance for this benchmark is shown in Figure 5.8.

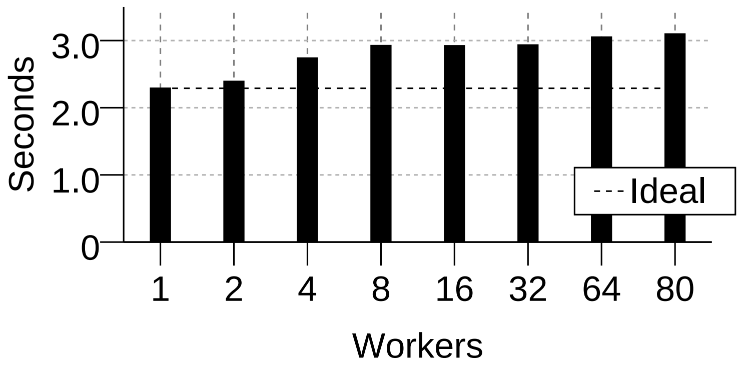

Figure 5.8: Scaling Performance for Price Change computation

```
def pagerank():
  graph = load('/dfs/graph.dat')
  num_pages = graph.shape[0]
  ranks = rand(num_pages)
  for i in range(50):
    ranks = dot(graph, ranks)
```

Listing 5.4: Spartan PageRank Implementation

5.4.5 PageRank

We implemented the PageRank algorithm in Spartan as shown in Listing 5.4. Rather than using the more familiar explicit form which loops over pages and outlinks, we simply use the dot (matrix multiplication) operator to multiply our graph matrix and rank vector. This is the *Power Method* for computing eigenvalues. It is dramatically simpler than the Piccolo implementation shown in Listing 2.4.

To evaluate this application, as with the Piccolo PageRank experiment, we generate a random web graph based on the expected distribution of sites. As Spartan does not yet have support for streaming sparse data from disk, we are not able to test it with as large of a graph. The performance is shown in in Figure 5.9. The Spartan implementation is slightly slower than the original Piccolo implementation. This is primarily due to inefficiencies in the local sparse multiplication routines, which we are addressing.

We also implemented 2 larger applications in Spartan; a convolutional neural network and an sparse matrix factorizer using stochastic gradient descent.

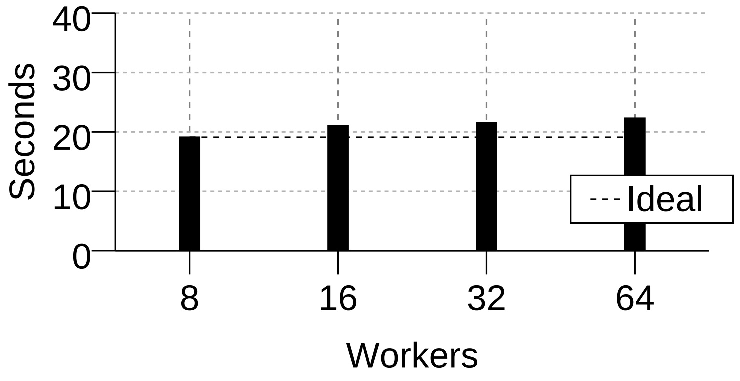

Figure 5.9: Scaling Performance for PageRank

5.4.6 Convolutional Neural Network

This example implements the forward propagation pass for a convolutional neural network (CNN) [52]. The builtin multiplication and stencil operators make this application reasonably straightforward to implement. Distributing CNNs has become a very popular area of interest recently [31], as they can achieve very good classification performance, but take a long time to train, even when using optimized GPU kernels [50]. The implementation of this application is shown in Listing 5.5. This application also makes use of the *maxpool* function; this function is substantially similar to the stencil operation, but instead of multiplying a window of an array with a set of weights, it applies a function (in this case *max*). While these operations are currently implemented separately in Spartan, we hope to combine their functionality soon.

Figure 5.10 shows the time to forward propagate a batch of images through a 3-layer CNN, using 512x512 images and filters of size 9x9. We scale the batch size with the number of workers. Our current implementation uses a CPU-based convolution routine, which limits the per-worker performance. We are working on extending the stencil operation to support efficient GPU based operations.

5.4.7 Matrix Factorization

For this experiment we implemented the stochastic gradient descent (SGD) based matrix factorization technique described in the Sparkler[53] paper. We used the sparse Netflix Prize [7]

```
# images is a 4-dimensional array (image #, width, height, color)
# filters are 4-dimensional arrays: (filter#, filter_size, filter_size, color)
def fprop(images, filter_0, filter_1, filter_2):
  # convolve images with filter_0 using a stride of 4.
  # scan across the width and height of each image
  conv1 = stencil(images, filter_0, 4, (1, 2))
  pool1 = maxpool(conv1, stride=2, axis=(1,2))

  conv2 = stencil(pool1, filter_1, 1, (1, 2))
  pool2 = maxpool(conv2, stride=2, axis=(1,2))

  conv3 = stencil(pool2, filter_2, 1, (1, 2))
  pool3 = maxpool(conv3, stride=2, axis=(1,2))

  # return a vector of the final output
  return pool3.ravel()
```

Listing 5.5: Convnet Forward Propagation

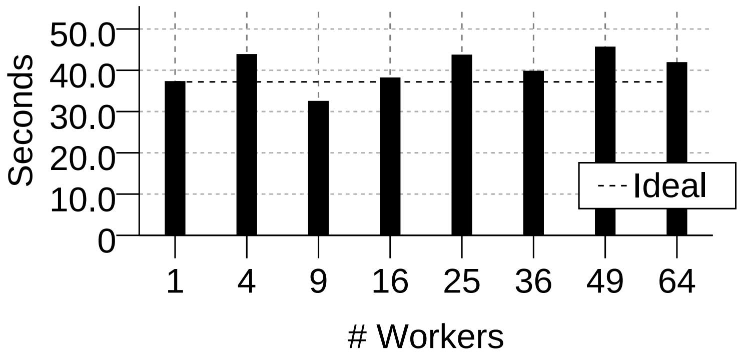

Figure 5.10: Scaling Performance for Convolutional Neural Network

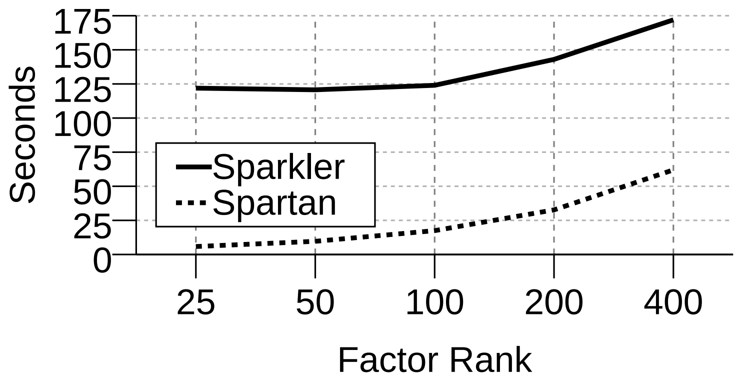

Figure 5.11: Netflix Matrix Factorization

dataset for evaluation: this consists of 100 million ratings across 500k users and 17k movies. Matrix factorization is a popular and effective mechanism for learning predictions on large datasets such as this one.

The core of this algorithm consists of mapping over independent tiles ("strata"), and updating the 2 *factor* matrices (U and M in the code listing). This operation does not directly correspond to any of the builtin Spartan operators, but it is relatively straightforward to implement via the generic shuffle operator and leveraging the updateable array interface provided by the backend. The implementation is shown in Figure 5.6. We refer readers to the Sparkler paper for details about the algorithm.

The results for this experiment (run on 10 machines to match the experiments from the Sparkler paper) are shown in Figure 5.11. Both systems scale reasonably well as the rank of the factor matrices increases, with Spartan generally running 50% faster. The native Spark implementation (from the Sparkler paper) is approximately 10 times slower. The Sparkler times are from the Sparkler paper (the Sparkler cluster machines are somewhat more powerful than our own).

```
# The input to this computation is three arrays: V, M and U.
# U and M are factor matrices with shape (#users, rank) and (#movies, rank).
# V is a sparse matrix of shape (#users, #movies) in COO format
# (it holds three arrays of the form (row, column, value))
#
# We iterate over V and update the U and M matrices based on prediction error.
# (Array mutation is allowed on backend arrays).
@parakeet.jit
def _sgd_inner(rows, cols, vals, u, m):
  for offset, mid, rating in zip(rows, cols, vals):
    u_idx = offset
    m_idx = mid
    guess = np.dot(u[u_idx], m[m_idx].T)
    diff = rating - guess
    u[u_idx] += u[u_idx] * diff * EPSILON
    m[m_idx] += u[u_idx] * diff * EPSILON

def sgd_netflix_mapper(inputs, ex, V, M, U, worklist):
  if not ex in worklist: return

  v = V.fetch(ex)
  u = U.fetch(ex[0]) # size: (ex.shape[0] * r)
  m = M.fetch(ex[1]) # size: (ex.shape[1] * r)

  _sgd_inner(v.row, v.col, v.data)

  U.update(ex[0], u)
  M.update(ex[1], m)
  return []

def strata_overlap(extents, v):
  for ex in extents:
    if v.ul[0] <= ex.ul[0] and v.lr[0] > ex.ul[0]: return True
    if v.ul[1] <= ex.ul[1] and v.lr[1] > ex.ul[1]: return True
  return False

def _compute_strata(V):
  strata = []
  extents = V.tiles.keys()
  random_shuffle(extents)

  while extents:
    stratum = []
    for ex in list(extents):
      if not strata_overlap(stratum, ex):
        stratum.append(ex)
    for ex in stratum: extents.remove(ex)

    strata.append(stratum)
  return strata

def sgd(V, M, U):
    strata = _compute_strata(V)
    for i, stratum in enumerate(strata):
      worklist = set(stratum)
      force(shuffle(V, sgd_netflix_mapper,
                    V, M, U, worklist))
```
Listing 5.6: Matrix Factorization Implementation

6 Related Work

6.1 Piccolo

Communication-oriented models: Communication-based primitives such as MPI [38] and Parallel Virtual Machine (PVM [82]) 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 [10], fault-tolerance via machine virtualization [60] and the use of hybrid checkpoint and logging for recovery [19]. 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 [88]. 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 [44]. Recently, the BSP model has been adopted in the Pregel framework for parallelizing work on large graphs [56].

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 [49, 48, 54, 15]. 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) [33, 62, 90] 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 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 languages such as Linda [23] and frameworks like JavaSpaces [39] and GigaSpaces [3], give users access to one or more global tuple-spaces accessible from all participating threads. Tuple spaces provide atomic primitives for executing tasks and reading and writing tuples, including a form of "compare-and-swap" to read a matching tuple and remove it from the space simultaneously. These features make them very useful for coordinating workers in a distributed environment. Enhancements to the original model have added support for multiple spaces [41], and bulk primitives for copying and moving matching tuples between spaces, enabling global synchronization [76, 22]. Modern implementations like GigaSpaces [3] have support for user specified partitioning of data and replication. Tuple-spaces do not have an efficient mechanism for coordinating multiple updates to a single entry; this limits their usefulness for update intensive tasks such as PageRank or web crawling.

MapReduce and Dataflow models: In recent years, MapReduce has emerged as a popular programming model for parallel data processing [36]. There are many recent efforts inspired by MapReduce ranging from generalizing MapReduce to support the join operation [46], improving MapReduce's pipelining performance [32], building high-level languages on top of MapReduce (e.g. DryadLINQ [92], Hive [85], Pig [65] and Sawzall [69]). FlumeJava [25] 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 [36] and Dryad [46] are instances of stream processing, or data-flow models. Because of MapReduce's popularity, programmers started using it to build in-memory iterative applications such as logistic regression and k-means [9], even though it is not natural fit for these applications. Spark [93] focuses on in-memory datasets and leverages an accumulator-like technique which allows it to provide reasonably high-performance for this type of iterative application. It retains a dataflow model with immutable collections, which prevents writing long-running applications such as a web crawler.

The sparse nature of these systems enables them to support a wide variety of datasets and programming tasks, but they suffer from poor performance for typically "dense" operations (matrix multiply, etc) [72] and lack efficient or convenient support for operations such as slicing.

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 [79, 84, 21], most parallel models for a single machine are based on shared-memory. For the GPU platform, there are CUDA [63] and OpenCL [43]. For multi-core CPUs, Cilk [17] and more recently, Intel's Thread Building Blocks [73] provide support for low-overhead thread creation and dispatching of tasks at a fine level. OpenMP [34] 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 [45, 13]. 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 [42], Memcached [68], the recently proposed RamCloud [67], and many key-value stores based on distributed hash tables [11, 37, 80, 75]. These systems do not seek to provide a computation model, but rather are targeted towards loosely-coupled distributed applications such as web serving.

6.2 Spartan

The above projects are all related to Spartan (as it uses a Piccolo-style backend). In addition, the following array oriented systems share similar feature sets.

Parallel vector languages such as ZPL [55], SISAL [58] and NESL [16] share many common characteristics with Spartan. These languages all expose a form of distributed array or vector primitive, with a small set of core operators to perform parallel operations. These languages each have certain limitations not shared by Spartan: ZPL does not allow for arbitrary indexing of parallel arrays, and does not allow parallelization of indexable arrays. NESL relies on a *PRAM* model which assumes that a shared, distributed region of memory can be accessed in a reasonable amount of time; this is not the case for modern computers and networks. The original SISAL design assumed a similar memory model, updated versions provide a more explicit tiled model for arrays [40]. Unlike Spartan's generic accumulator design, SISAL is limited to a small number of builtin reduction operators. Both SISAL and ZPL focus on static compilation of an array program, which makes them unsuitable for the interactive environment which modern array languages often used in.

SciDB [81] extends a traditional relational database (PostgreSQL) with support for distributed multi-dimensional arrays. SciDB is designed to enable processing of large dense or sparse array datasets within a familiar SQL environment. Users can express certain array computations using an extended form of SQL with support for some array slices and filtering operations. As SciDB is primarily targeted for data retrieval on very large (on-disk) datasets, it is does yet provide a very efficient backend for distributed in-memory computation, as we have found with our own testing.

Distributed array backends have been in development for many years: ScaLAPACK[29] provides distributed implementation of the Lapack [12] linear algebra API, and is still under development. More recent projects such as Elemental [70] and the Global Arrays Toolkit [61] export a similar interface but have better performance on modern hardware. These projects are focused on providing highly optimized implementations of specific operations: they are very fast for operations that have been built-in, but their low-level nature does not allow for higher-level optimizations, and their programming model (based on MPI) is difficult to extend.

Specialized array frameworks Dandelion [74] is an extension to the LINQ [59] programming framework which adds specific support for compiling operations to GPU kernels. Dandelion retains LINQ's sparse relational (key-value) focus, and does not support directly support such as slicing, tiling or filtering for arrays. MadLINQ [72] is an extension to the DryadLINQ [92] execution engine which adds specific support for dense distributed arrays. This is accomplished by creating a new *Matrix* class to represent distributed arrays, and uses a clever lazy evaluation strategy which expands execution graphs on-demand. MadLINQ shows good performance on matrix multiplication and factorization benchmarks (beating the highly tuned ScaLAPACK code). Operations in MadLINQ operate on whole tiles at a time; this makes it unsuitable for performing operations such as slicing or filtering or filtering.

Presto [89] is an extension to the R [83] statistical programming language which adds support for distributed array processing. Presto builds on top of distributed, updateable tables (similar to Piccolo), using a parallel *foreach* operator to exploit multiple machines. Presto has explicit support for iterative operations via the `onchange` and `update` operators. Like MadLINQ, Presto operates at the tile level and does not sub-tile operations such as slicing or filtering.

7

Conclusion

This thesis presented two complementary frameworks, Piccolo and Spartan, which are designed to help users write efficient distributed in-memory applications. The design of both systems allows them to efficiently execute a wide variety of in-memory applications.

Piccolo provides users with a distributed in-memory table interface for sharing state. A key feature of Piccolo is its usage of commutative operators to efficiently coordinate updates to shared data, which provides a general solution to the problem of write-write conflict issues found in existing distributed shared memory systems. We demonstrate the usefulness of Piccolo by developing a range of applications, including k-means, n-body simulation, PageRank and a distributed web crawler.

Spartan addresses the difficulty of writing raw Piccolo programs. Spartan implements a distributed version of the NumPy array language. Spartan combines a number of techniques, including lazy evaluation, high-level operators and expression graph optimizations to compile high-level array language code to an extended Piccolo backend. We evaluated Spartan on a number of common array language applications such as k-means, linear regression, PageRank and matrix factorization, and show that it obtains good performance while allowing users to write simple, high-level array code.

Bibliography

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

[2] GHC optimisations. `http://www.haskell.org/haskellwiki/GHC_optimisations`.

[3] Gigaspaces: Xap in-memory computing. `http://www.gigaspaces.com/`.

[4] Julia language. `http://julialang.org/`.

[5] Matlab K-means with Simple Patches.
`http://www.cc.gatech.edu/grads/d/dkuang3/software/kmeans3.html`.

[6] Matrix multiplication using mpi. `http://www.cs.umanitoba.ca/~comp4510/examples.html`.

[7] Netflix prize.

[8] numexpr: Fast numerical array expression evaluator for python and numpy.
`https://github.com/pydata/numexpr`.

[9] Mahout: Scalable machine learning and data mining, 2012. `http://mahout.apache.org/`.

[10] 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.

[11] 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.

[12] Anderson, E., Bai, Z., Dongarra, J., Greenbaum, A., McKenney, A., Du Croz, J., Hammerling, S., Demmel, J., Bischof, C., and Sorensen, D. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (1990), IEEE Computer Society Press, pp. 2–11.

[13] Basumallik, A., Min, S.-J., and Eigenmann, R. Programming distributed memory sytems using OpenMP. *Parallel and Distributed Processing Symposium, International 0* (2007), 207.

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

[15] 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).

[16] Blelloch, G. E. NESL: A nested data-parallel language.(version 3.1). Tech. rep., DTIC Document, 1995.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Butcher, P., Wood, A., and Atkins, M. Global synchronisation in linda. *Concurrency: Practice and Experience 6*, 6 (1994), 505–516.

[23] Carriero, N., and Gelernter, D. Linda in context. *Commun. ACM 32*, 4 (1989), 444–458.

[24] Chamberlain, B. L., Callahan, D., and Zima, H. P. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[25] 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).

[26] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3* (1985), 63–75.

[27] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS) 3*, 1 (1985), 63–75.

[28] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices 40*, 10 (2005), 519–538.

[29] Choi, J., Dongarra, J. J., Pozo, R., and Walker, D. W. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the* (1992), IEEE, pp. 120–127.

[30] Clint Whaley, R., Petitet, A., and Dongarra, J. J. Automated empirical optimizations of software and the atlas project. *Parallel Computing 27*, 1 (2001), 3–35.

[31] Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (2013), pp. 1337–1345.

[32] Condie, T., Conway, N., Alvaro, P., and Hellerstein, J. MapReduce online. In *NSDI* (2010).

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

[34] 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.

[35] Daily, J., and Lewis, R. R. Using the global arrays toolkit to reimplement numpy for distributed computation. In *Proceedings of the 10th Python in Science Conference* (2011).

[36] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (2004).

[37] 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.

[38] Forum, M. MPI 2.0 standard, 1997.

[39] Freeman, E., Arnold, K., and Hupfer, S. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[40] Gaudiot, J.-L., Bohm, W., Najjar, W., DeBoni, T., Feo, J., and Miller, P. The sisal model of functional programming and its implementation. In *Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium* (1997), IEEE, pp. 112–123.

[41] Gelernter, D. *Multiple tuple spaces in Linda.* Springer, 1989.

[42] 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.

[43] Group, K. O. W. The OpenCL specification. Tech. rep., 2009.

[44] 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).

[45] Hoeflinger, J. P. Extending OpenMP to clusters. Tech. rep., Intel, 2009.

[46] 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).

[47] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair scheduling for distributed computing clusters. In *SOSP* (2010).

[48] Johnson, K. L., Kaashoek, M. F., and Wallach, D. A. CRL: High-performance all-software distributed shared memory. In *SOSP* (1995).

[49] 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).

[50] Krizhevsky, A., Sutskever, I., and Hinton, G. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25* (2012), pp. 1106–1114.

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

[52] Le Cun, B. B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems* (1990), Citeseer.

[53] Li, B., Tata, S., and Sismanis, Y. Sparkler: Supporting large-scale matrix factorization. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), ACM, pp. 625–636.

[54] Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS) 7* (1989), 321–359.

[55] Lin, C., and Snyder, L. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing*. Springer, 1994, pp. 96–114.

[56] 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.

[57] MathWorks. MATLAB software.

[58] McGraw, J., Skedzielewski, S., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B., and Thomas, R. *SISAL: streams and iteration in a single assignment language. Language Reference Manual*. 1985.

[59] Meijer, E., Beckman, B., and Bierman, G. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), ACM, pp. 706–706.

[60] 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.

[61] Nieplocha, J., Harrison, R. J., and Littlefield, R. J. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing 10*, 2 (1996), 169–189.

[62] Numrich, R. W., and Reid, J. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum 17* (August 1998), 1–31.

[63] Nvidia. Cuda programming guide, 2007.

[64] Oliphant, T., et al. Numpy, a python library for numerical computations.

[65] 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).

[66] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 1099–1110.

[67] 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).

[68] Phillips, L., and Fitzpatrick, B. Livejournal's backend and memcached: Past, present, and future. In *LISA* (2004), USENIX.

[69] Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. Interpreting the data: Parallel analysis with Sawzall. In *Scientific Programming* (2005).

[70] Poulson, J., Marker, B., van de Geijn, R. A., Hammond, J. R., and Romero, N. A. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw. 39*, 2 (feb 2013), 13:1–13:24.

[71] Power, R., and Li, J. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI* (2010), pp. 293–306.

[72] Qian, Z., Chen, X., Kang, N., Chen, M., Yu, Y., Moscibroda, T., and Zhang, Z. MadLINQ: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 197–210.

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

[74] Rossbach, C. J., Yu, Y., Currey, J., Martin, J.-P., and Fetterly, D. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 49–68.

[75] 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).

[76] Rowstron, A. I., and Wood, A. Bonita: A set of tuple space primitives for distributed coordination. In *System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on* (1997), vol. 1, IEEE, pp. 379–388.

[77] Rubinsteyn, A. *Runtime Compilation of Array-Oriented Python Programs*. PhD thesis, New York University, 2013.

[78] Singh, J. P., Weber, W.-D., and Gupta, A. SPLASH: Stanford parallel applications for shared-memory. Tech. rep., Stanford University, 1991.

[79] Stephens, R. A survey of stream processing, 1995.

[80] 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.

[81] Stonebraker, M., Brown, P., Poliakov, A., and Raman, S. The architecture of SciDB. In *Scientific and Statistical Database Management* (2011), Springer, pp. 1–16.

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

[83] Team, R. D. R: A language and environment for statistical computing.

[84] Thies, W., Karczmarek, M., and Amarasinghe, S. StreamIt: A language for streaming applications. In *Compiler Construction* (2002), Springer, pp. 49–84.

[85] 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.

[86] tsang Lee, H., Leonard, D., Wang, X., and Loguinov, D. Irlbot: Scaling to 6 billion pages and beyond. In *WWW Conference* (2008).

[87] Twitter. Scalding: A scala api for cascading.

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

[89] Venkataraman, S., Bodzsar, E., Roy, I., AuYoung, A., and Schreiber, R. S. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 197–210.

[90] 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).

[91] 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).

[92] 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).

[93] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), pp. 10–10.