Piccolo

Fast, Distributed Programs with Partitioned Tables

Presenter: Wu, Weiyi
Yale University
Outline

• Background
• Intuition
• Design
• Evaluation
• Future Work
Outline

• Background
• Intuition
• Design
• Evaluation
• Future Work
MapReduce

Data

Humble Workers

Master
MapReduce

Humble Workers

Data

Assign Map / Reduce

Master

Saturday, October 15,
MapReduce

Data

Humble Workers

Master
MapReduce

Data

Humble Workers

Master

Map
MapReduce

Data

Humble Workers

Master

Reduce
MapReduce

Master

Data

Humble Workers
MPI / RPC

Humble Workers

Data
MPI / RPC

Messages All Around

Data

Humble Workers
MPI / RPC

Data

Humble Workers
Distributed Shared Memory

Humble Workers
Distributed Shared Memory

Humble Workers

Underlying Message
write / read
Distributed Shared Memory

Non-atomic
Hard to optimize
<table>
<thead>
<tr>
<th>Key-Value Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Humble Workers</td>
</tr>
</tbody>
</table>

Saturday, October 15,
Key-Value Table

Humble Workers

Underlying Message

cut / get
Key-Value Table

Atomic
Easy to optimize
Outline

• Background
• Intuition
• Design
• Evaluation
• Future Work
What do we need?

- MapReduce
- MPI / RPC
- DSM
- k/v Table
What do we need?

- In-memory
- MapReduce
- MPI / RPC
- DSM
- k/v Table
What do we need?

- In-memory
- Data-centric

MapReduce
MPI / RPC
DSM
k/v Table
What do we need?

- In-memory
- Data-centric
- Exposing globally shared state

MapReduce
MPI / RPC
DSM
k/v Table
What do we need?

- In-memory
- Data-centric
- Exposing globally shared state

MPI / RPC

DSM

k/v Table
What do we need?

- In-memory
- Data-centric
- Exposing globally shared state
- No low-level messages
What do we need?

- In-memory
- Data-centric
- Exposing globally shared state
- No low-level messages
What do we need?

- In-memory
- Data-centric
- Exposing globally shared state
- No low-level messages
- Easy to use / optimize

DSM

k/v Table
What do we need?

• In-memory
• Data-centric
• Exposing globally shared state
• No low-level messages
• Easy to use / optimize

k/v Table
What do we need?

• In-memory
• Data-centric
• Exposing globally shared state
• No low-level messages
• Easy to use / optimize
Is k/vTable enough?

- Replace put-get pairs to atomic ops
- Improving locality
- Load Balancing
- Rapid and Reliable Checkpoint
Outline

• Background
• Intuition
• Design
• Evaluation
• Future Work
Overview

Humble Workers

Master
Overview

Humble Workers

Master

Assign Partition & Task
Overview

Humble Workers

Master
Overview

Humble Workers

Master

Execute Kernel
Overview

Humble Workers

Master

Saturday, October 15, 11
Overview

Humble Workers

Kernel Finished

Master

Saturday, October 15,
Overview

Humble Workers

Master
Expressing Locality

- Reduce remote read (get)
- Co-locate a kernel execution with some table partitions
- Co-locate partitions of different tables (with same partition id)
User-defined accumulators
User-defined accumulators

• \( a \leftarrow \text{get}(A) \)
• \( b \leftarrow \text{get}(B) \)
• \( \text{res} \leftarrow a + b \)
• \( \text{put}(B, \text{res}) \)
User-defined accumulators

• $a \leftarrow \text{get}(A)$
• $b \leftarrow \text{get}(B)$
• $\text{res} \leftarrow a + b$
• $\text{put}(B, \text{res})$

• $a \leftarrow \text{get}(A)$

Saturday, October 15,
User-defined accumulators

• `a <- get(A)`
• `b <- get(B)`
• `res <- a + b`
• `put(B, res)`

• `a <- get(A)`
• `update(B, a)`
Load Balance

Humble Workers

Master
Load Balance

Humble Workers

Assign Partition & Task

Master
Load Balance

Humble Workers

Master
Load Balance

Humble Workers

Master

Execute Kernel
Load Balance

Humble Workers

Execute Kernel
Migrate Partition

Master
Load Balance

Humble Workers

Master

Execute Kernel
Migrate Partition
Load Balance

Humble Workers

Execute Kernel

Master

Saturday, October 15,
Load Balance

Humble Workers

Execute Kernel

Steal Work
Load Balance

Humble Workers

Master

Execute Kernel

Steal Work

Saturday, October 15, 2023
Load Balance

Humble Workers

Execute Kernel

Master
Load Balance

Humble Workers

Master
Checkpoint

Humble Workers

Start Checkpoint

Master
Checkpoint

Humble Workers

Snapshot

Master
Checkpoint

Humble Workers

Master

Snapshot
Checkpoint

Humble Workers

Log Ops

Master
Checkpoint

Humble Workers

Master
Checkpoint

Humble Workers

Finish Checkpoint

Saturday, October 15,
Checkpoint

Humble Workers

Master
Outline

• Background
• Intuition
• Design
• Evaluation
• Future Work
Scaling - Speedup

Figure 6: Scaling performance (fixed default input size)
Scaling - Input

Figure 6: Scaling performance (fixed default input size)

6.2 Scaling Performance

Figure 7: Scaling input size.

6.3 EC2

Figure 8: Scaling input size on EC2.

6.4 Comparison with Other Frameworks

Comparison with Hadoop:

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

Figure 7 shows the scaling for all applications. The achieved scaling for all applications is within 20% of the ideal number.

Figure 8 shows the scaling of PageRank and k-means on EC2 as we increase their input size with N. We were somewhat surprised to see that the resulting scaling on EC2 is better than achieved on our small local testbed. Our local testbed's CPU performance exhibited quite some variability, impacting scaling. After further investigation, we believe the source for such variability is likely due to dynamic CPU frequency scaling.

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

6.4 Comparison with Other Frameworks

Comparison with Hadoop:

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

Figure 7 shows the scaling for all applications. The achieved scaling for all applications is within 20% of the ideal number.

Figure 8 shows the scaling of PageRank and k-means on EC2 as we increase their input size with N. We were somewhat surprised to see that the resulting scaling on EC2 is better than achieved on our small local testbed. Our local testbed's CPU performance exhibited quite some variability, impacting scaling. After further investigation, we believe the source for such variability is likely due to dynamic CPU frequency scaling.

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

Figure 6 shows application speedup as the number of workers ($N$) increases from 8 to 64 for the default input size. All applications are CPU-bound and exhibit good speedup with increasing $N$. Ideally, all applications (except for PageRank) have perfectly balanced table partitions and should achieve linear speedup. However, to have reasonable running time at $N=8$, we choose a relatively small default input size. Thus, as $N$ increases to 64, Piccolo’s overhead is no longer negligible relative to applications’ own computation (e.g. k-means finishes each iteration in 1.4 seconds at $N=64$), resulting in 20% less than ideal speedup. PageRank’s table partitions are not balanced and work stealing becomes important for its scaling (see § 6.5).

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

6.3 EC2

We investigated how Piccolo scales with a larger number of machines using 100 EC2 instances. Figure 8 shows the scaling of PageRank and k-means on EC2 as we increase their input size with $N$. We were somewhat surprised to see that the resulting scaling on EC2 is better than achieved on our small local testbed. Our local testbed’s CPU performance exhibited quite some variability, impacting scaling. After further investigation, we believe the source for such variability is likely due to dynamic CPU frequency scaling.

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

6.4 Comparison with Other Frameworks

Comparison with Hadoop: We implemented PageRank and k-means in Hadoop to compare their performance against that of Piccolo. The rest of our applications, including the distributed web crawler, n-body and matrix multiply, were run on a 16-worker Hadoop cluster.

Figure 8: Scaling input size on EC2.
Comparison with MapReduce on Hadoop

![Bar chart comparing performance of PageRank and k-means on Hadoop and Piccolo](image)

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

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

Figure 11: Effect of Work Stealing and Slow Workers

Having only 50% of the CPU time of the other workers.
The results of these tests are shown in Figure 11. Work stealing improves running time by 10% when all machines are operating normally. The improvement is due to the imbalance in the input partition sizes - when run without work stealing, the computation waits longer for the workers processing more data to catch up.
The effect of slow workers on the computation is more dramatic. With work-stealing disabled, the runtime is nearly double that of the normal computation, as each iteration must wait for the slowest worker to complete all assigned tasks. Enabling work stealing improves the situation dramatically - the computation time is reduced to less than 5% over that of the non-slow case.

6.6 Checkpointing

We evaluated the checkpointing overhead using the PageRank, k-means and n-body problems. Compared to the other problems, PageRank has a larger table that needs to be checkpointed, making it a more demanding test of checkpoint/restore performance. In our experiment, each worker wrote its checkpointed table partitions to the local disk. Figure 12 shows the runtime when checkpointing is enabled relative to when there is no checkpointing. For the naïve synchronous checkpointing strategy, the master starts checkpointing only after all workers have finished. For the optimized strategy, the master initiates the checkpoint as soon as one of the workers has finished. As the figure shows, overhead of the optimized checkpointing strategy is quite negligible (≈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 size 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 computation whose table state uses all available system memory. We estimate the fraction of time a Piccolo computation would spend working productively (not in a checkpoint or restore state), for varying numbers of machines and failure rates. In our model, we assume that machine failures arrive at a constant interval defined by the failure rate and the number of machines in a cluster. While this is a simplification of real-life failure behavior, it is a worst-case scenario for the restore mechanism, and as such provides a useful lower bound. The expected efficiency based on our model is shown in Figure 13. For well maintained data-centers that we are familiar with, the average machine uptime is typically around 1 year. For these data-centers, the global checkpointing mechanism can efficiently scale up to a few thousand machines.

6.7 Distributed Crawler

We evaluated our distributed crawler implementation using various numbers of workers. The URL table was initialized with a seed set of 1000 URLs. At the end of a 30 minutes run of the experiment, we measured the number of pages crawled and bytes downloaded. Figure 14 shows the crawler's web page download throughput in
Outline

• Background
• Intuition
• Design
• Evaluation
• Future Work
Future Work

• Log-based scalable failure handling
• More user-defined accumulator per table
• Distributed as Parallel
Thanks ~.~