Piccolo: Building fast distributed programs with partitioned tables

Russell Power
Jinyang Li
New York University
Motivating Example: PageRank

for each node $X$ in graph:
for each edge $X \rightarrow Z$:
$\text{next}[Z] += \text{curr}[X]$

Repeat until convergence

Input Graph

<table>
<thead>
<tr>
<th></th>
<th>Curr</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow B, C, D$</td>
<td>A: 0.25</td>
<td>A: 0.25</td>
</tr>
<tr>
<td></td>
<td>B: 0.17</td>
<td>B: 0.17</td>
</tr>
<tr>
<td></td>
<td>C: 0.22</td>
<td>C: 0.22</td>
</tr>
<tr>
<td>$B \rightarrow E$</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$C \rightarrow D$</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Fits in memory!
PageRank in MapReduce

- Data flow models do not expose global state.
PageRank in MapReduce

- Data flow models do not expose global state.
PageRank With MPI/RPC

Graph A->B,C
Ranks A: 0

Graph C->E,F

Distributed Storage

Ranks B: 0
Graph B->D

User explicitly programs communication
Piccolo’s Goal: Distributed Shared State

Distributed in-memory state

Ranks
A: 0
B: 0
...

Graph
A->B,C
B->D
...

Distributed Storage

read/write
Piccolo’s Goal: Distributed Shared State

Graph A->B,C
Ranks A: 0
... 

Graph B->D
Ranks B: 0
... 

Graph C->E,F
Ranks C: 0
... 

Piccolo runtime handles communication

1

2

3
Ease of use  Performance
Talk outline

- Motivation
- Piccolo's Programming Model
- Runtime Scheduling
- Evaluation
Implemented as library for C++ and Python

Graph
A → B, C
B → D
...

Ranks
A: 0
B: 0
...

1

read/write

get/put
update/iterate

2

3

Programming Model
Naïve PageRank with Piccolo

curr = Table(key=PageID, value=double)
next = Table(key=PageID, value=double)

def pr_kernel(graph, curr, next):
    i = my_instance
    n = len(graph)/NUM_MACHINES
    for s in graph[(i-1)*n:i*n]
        for t in s.out:
            next[t] += curr[s.id] / len(s.out)

def main():
    for i in range(50):
        launch_jobs(NUM_MACHINES, pr_kernel, graph, curr, next)
        swap(curr, next)
        next.clear()
Naïve PageRank is Slow

Graph
B -> D
... 

Graph
C -> E, F
...

Ranks
A: 0
...

Ranks
C: 0
...

Ranks
B: 0
...

Graph
A -> B, C
...
PageRank: Exploiting Locality

curr = Table(..., partitions=100, partition_by=site)
next = Table(..., partitions=100, partition_by=site)

group_tables(curr, next, graph)

def pr_kernel(graph, curr, next):
    for s in graph.get_iterator(my_instance)
        for t in s.out:
            next[t] += curr[s.id] / len(s.out)

def main():
    for i in range(50):
        launch_jobs(curr.num_partitions,
                     pr_kernel,
                     graph, curr, next,
                     locality=curr)
    swap(curr, next)
    next.clear()
Exploiting Locality

**Graph**
- B->D
  ...

**Ranks**
- A: 0
  ...

**Graph**
- C->E,F
  ...

**Ranks**
- C: 0
  ...

**Graph**
- A->B,C
  ...

**Ranks**
- B: 0
  ...

2

1

3
Exploiting Locality

Graph B -> D

Ranks A: 0

1

get
put

get
put

get
put

Graph C -> E, F

Ranks C: 0

2

3

Ranks B: 0

Graph A -> B, C
Synchronization

1

How to handle synchronization?

put (α=0.3)

put (α=0.2)

Graph B→D
...

Graph C→E,F
...

Graph A→B,C
...

Ranks A: 0
...

Ranks C: 0
...

Ranks B: 0
...
Synchronization Primitives

- Avoid write conflicts with accumulation functions
  - NewValue = Accum(OldValue, Update)
    - *sum*, *product*, *min*, *max*

  ➔ Global barriers are sufficient

- Tables provide release consistency
PageRank: Efficient Synchronization

curr = Table(..., partition_by=site, accumulate=sum)
next = Table(..., partition_by=site, accumulate=sum)
group_tables(curr, next, graph)

def pr_kernel(graph, curr, next):
    for s in graph.get_iterator(my_instance):
        for t in s.out:
            next.update(t, curr.get(s.id)/len(s.out))

def main():
    for i in range(50):
        handle = launch_jobs(curr.num_partitions,
                              pr_kernel,
                              graph, curr, next,
                              locality=curr)
        barrier(handle)
        swap(curr, next)
        next.clear()
Efficient Synchronization

Runtime
Workers buffer updates locally
→ Release consistency

Graph
B→D
...

Graph
C→E,F
...

Graph
A→B,C
...

Ranks
A: 0
...

Ranks
C: 0
...

Ranks
B: 0
...

update (α, 0.2)

update (α, 0.3)

1

2

3
Table Consistency

Graph
A -> B, C
Ranks
A: 0

Graph
B -> D
Ranks
B: 0

Graph
C -> E, F
Ranks
C: 0

update (a, 0.3)
put (a=0.2)
put (a=0.3)
PageRank with Checkpointing

curr = Table(..., partition_by=site, accumulate=sum)
next = Table(..., partition_by=site, accumulate=sum)
group_tables(curr, next)
def pr_kernel(graph, curr, next):
    for node in graph.get_iterator(my_instance)
        for t in s.out:
            next.update(t, curr.get(s.id)/len(s.out))

def main():
    curr, userdata = restore()
    last = userdata.get('iter', 0)
    for i in range(last, 50):
        handle = launch_jobs(curr.num_partitions, pr_kernel,
                             graph, curr, next, locality=curr)
        cp_barrier(handle, tables=(next), userdata={'iter':i})
    swap(curr, next)
    next.clear()
Recovery via Checkpointing

Graph B->D
Ranks A: 0

Ranks C: 0

Graph C->E,F

Distributed Storage

Runtime uses Chandy-Lamport protocol

Ranks B: 0

Graph A->B,C
Talk Outline

- Motivation
- Piccolo's Programming Model
- Runtime Scheduling
- Evaluation
Load Balancing

Other workers are updating P6!

Pause updates!

Coordinates work-stealing

master
Talk Outline

- Motivation
- Piccolo's Programming Model
- System Design
- Evaluation
Piccolo is Fast

- NYU cluster, 12 nodes, 64 cores
- 100M-page graph

Main Hadoop Overheads:
- Sorting
- HDFS
- Serialization

PageRank iteration time (seconds)

<table>
<thead>
<tr>
<th>Workers</th>
<th>Hadoop</th>
<th>Piccolo</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>400</td>
<td>50</td>
</tr>
<tr>
<td>16</td>
<td>200</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>64</td>
<td>50</td>
<td>5</td>
</tr>
</tbody>
</table>
Piccolo Scales Well

- EC2 Cluster - linearly scaled input graph

1 billion page graph

PageRank iteration time (seconds)

Workers

- 12
- 24
- 48
- 100
- 200

ideal
Other applications

- Iterative Applications
  - N-Body Simulation
  - Matrix Multiply
- Asynchronous Applications
  - Distributed web crawler

No straightforward Hadoop implementation
Related Work

- **Data flow**
  - MapReduce, Dryad

- **Tuple Spaces**
  - Linda, JavaSpaces

- **Distributed Shared Memory**
  - CRL, TreadMarks, Munin, Ivy
  - UPC, Titanium
Conclusion

- Distributed shared table model
- User-specified policies provide for
  - Effective use of locality
  - Efficient synchronization
  - Robust failure recovery
Gratuitous Cat Picture

I can haz kwestions?

Try it out:
piccolo.news.cs.nyu.edu