Iterativeness-Aware Optimization for Big Data Analytics

Henggang Cui, Lianghong Xu

15799b Course Project Final Presentation
Big Data Analytics

Partitioned input data

Parallel iterative program

Model parameters
Big Data Analytics

Partitioned input data → Parallel iterative program → Model parameters

Parameter server
A Typical Parameter Server

Client -0

App. thread

Client library

Thread cache/oplog

Process cache/oplog

Tablet-server-0

Client -1

App. thread

Client library

Thread cache/oplog

Process cache/oplog

Tablet-server-1
Application is Iterative

• The application is often iterative
  – iterates over the input data
  – applies the same sequence of operations
  – reads/updates the same set of parameter data

• Knows everything about the access patterns
Example: PageRank

Illustration from http://www.seoxp.net/?p=40
Example: Topic Modeling

Topics

gene 0.04
dna 0.02
genetic 0.01
...

life 0.02
evolve 0.01
organism 0.01
...

brain 0.04
neuron 0.02
nerve 0.01
...

data 0.02
number 0.02
computer 0.01
...

Documents

Seeking Life's Bare (Genetic) Necessities

COLD SPRING HARBOR, NEW YORK—How many genes does an organism need to survive? Last week, at the genome meeting here, two genome researchers with radically different approaches presented complementary views of the basic genes needed for life.

One research team, using computer analyses to compare known genomes, concluded that today's organisms can be sustained with just 250 genes, and that the earliest life forms required a mere 128 genes. The other researcher mapped genes in a simple parasite and estimated that for this organism, 800 genes are plenty to do the job—but that anything short of 100 wouldn't be enough.

Although the numbers don't match precisely, those predictions are not all that far apart, especially in comparison to the 75,000 genes in the human genome, notes Svante Andersson of Uppsala University in Sweden, who arrived at the 800 number. But coming up with a consensus answer may be more than just a matter of numbers. "There are particularly more and more organisms that are completely sequenced," explains Arcady Mushegian, a computational molecular biologist at the National Center for Biotechnology Information (NCBI) in Bethesda, Maryland. Comparing...

Illustration from Blei, D. 2012
Project Idea

• Exploit the iterative nature
  – much optimization opportunity w/ known access patterns
  – this project exploits one of them: parameter data partitioning
Parameter Data Partitioning

Client -0
- App. thread
  - Client library
    - Thread cache/oplog
    - Process cache/oplog

Client -1
- App. thread
  - Client library
    - Thread cache/oplog
    - Process cache/oplog

Tablet-server-0
Tablet-server-1
Parameter Data Partitioning

• Intuition: minimize remote access
  – keeps the parameter data in the machine that accesses it with the highest frequency

• More considerations: load balancing
  – no server should have too many requests to service
  – no client should be delayed too much
Gathering Access Patterns

• A “Virtual Iteration”
  – has the application report the operation sequence
    • read, write, clock (sync point)

• Virtual iteration has almost no overhead
  – not any computation or data access involved!
  – worthwhile to do it before the real iterations
Example Info from a Virtual Iteration

Thread-0 on Machine-0

READ ROW 0
WRITE ROW 0
CLOCK
READ ROW 2
WRITE ROW 2
CLOCK

Thread-1 on Machine-1

READ ROW 1
WRITE ROW 1
CLOCK
READ ROW 3
WRITE ROW 3
CLOCK
Access Pattern to Tablet Servers

• The access pattern to servers is a little different
  – because of caching/batching
  – each client has only one read/write to a row per clock
Managing Parameter Data Location

• Manage location using a cluster of metadata servers
  – each manages the location of a subset of rows
  – client finds the right metadata server by hashing row ID

• Metadata servers also decides the mapping
  – clients send access info to them in the virtual iteration
Metadata Servers

Client -0

App. thread

Client library

Thread cache/oplog

Process cache/oplog

Client -1

App. thread

Client library

Thread cache/oplog

Process cache/oplog

Metadata-server-0  Tablet-server-0

Metadata-server-1  Tablet-server-1
Reasons of Design Choices

• Why distributed instead of centralized?
  – not much more complexity
  – don’t want one machine has more load than others
    • load balance issue
    • clients and servers are in the same set of machines
  – don’t have any centralized components before
    • had better not to break this
Reasons of Design Choices

• Why metadata server also decides the mapping?
  – the servers never talk to each other
  – communication only happens between clients and servers
Detailed Workflow

1. report access pattern
2. decide location
3. find row
4. reply location
5. access row

Client

Other clients

Metadata-server-X

Access patterns

Data location

Tablet-server-Y
Partition Policies

• Mapping: row_id -> server_id

• Policy 1: Random partitioning

• Policy 2: First accessing client

• Policy 3: Minimize remote access + load balancing
Random partitioning

• $server\_id = row\_id \% nr\_tablets$

• Pros: Load balancing

• Cons: Many non-local accesses
First Accessing Client

• server_id = first_accessing_client(row_id)

• Pros: At least 1 local access for any row

• Cons:
  – No explicit load balancing
  – Depending on arrival time of client requests
Min(non-local) + load balancing

• An row is placed on the server co-located with the most-frequently accessing client

• When access frequency is equal, choose the one with the least load

• Load: #rows
  – Future plan: use #(non-local) accesses
Experiment Setup

• 8 OpenCirrus machines
  – Each has 8 cores, 16GB memory

• Each machine runs a client, metadata server and tablet server

• Benchmarks:
  – PageRank (Berkstan dataset: 685k nodes and 7.6m edges)
  – Topic Modeling (ACL Anthology dataset: 15,032 docs)
  – Each benchmark is run for 3 times
PageRank Dataset

• 1 rank table (685229 rows)
Experimental Results: PageRank

![Bar Chart]
Experimental Results: PageRank

Local vs. Non-local Reads

#Read requests per server

Policy

<table>
<thead>
<tr>
<th>Policy</th>
<th>Non-local</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.5X local accesses than policy 1
Experimental Results: PageRank

Local vs. Non-local Updates

- **Policy 1**: Non-local = 12000000, Local = 2000000
- **Policy 2**: Non-local = 12000000, Local = 2000000

2X local accesses than policy 1
Experimental Results: Topic Modeling

![Row Assignment Diagram]

- **Policy 1:** Approximately 2,500 rows per server
- **Policy 2:** Approximately 2,500 rows per server
- **Policy 3:** Approximately 2,500 rows per server

The diagram illustrates the row assignment across different policies, showing consistent performance for each policy with around 2,500 rows per server.
Experimental Results: Topic Modeling

Local vs. Non-local Reads

#Read requests per server

Poor load balancing

Application-side optimization: most reads served from thread/process cache
Experimental Results: Topic Modeling

![Graph showing Local vs. Non-local Updates](image)

- **Policy 1**
  - Non-local: 200,000 updates per server
  - Local: 50,000 updates per server

- **Policy 2**
  - Non-local: 150,000 updates per server
  - Local: 25,000 updates per server

- **Policy 3**
  - Non-local: 100,000 updates per server
  - Local: 50,000 updates per server
Experimental Results: Runtime

Current implementation: Local access is as slow as remote access
Future Plans

• Look for more applications
• Optimize performance of local access
• Optimize implementation of policy 3
• Implement policy 4: using non-local requests as load metric
Topic Modeling Dataset

- 1 word table (101636 rows)
- 1 document table (299752 rows)