Tiled-MapReduce

Optimizing Resource Usages of Data-parallel Applications on Multicore

Rong Chen    Haibo Chen    Binyu Zang

Parallel Processing Institute
Fudan University
Data-Parallel Application

Data-parallel applications emerge and rapidly increase in past 10 years

- Google processes about 24 petabytes of data per day in 2008
- The movie AVATAR takes over 1 petabyte of local storage for 3D rendering *
- ...

Data-parallel Programming Model

MapReduce: a simple programming model for data-parallel applications from Google
Data-parallel Programming Model

**MapReduce**: a simple programming model for data-parallel applications from Google

- Functionality
- Parallelism
- Data Distribution
- Fault Tolerance
- Load Balance

**programmer**
Data-parallel Programming Model

**MapReduce**: a simple programming model for data-parallel applications from **Google**

Two Primitive:

- **Map** (*input*)
- **Reduce** (*key, values*)
Data-parallel Programming Model

**MapReduce**: a simple programming model for data-parallel applications from Google

### Two Primitive:

**Map** *(input)*
- for each *word* in *input*
- emit *(word, 1)*

**Reduce** *(key, values)*
- int *sum* = 0;
- for each *value* in *values*
-   *sum* += *value*;
- emit *(word, sum)*
Multicore is commercially prevalent recently

- Quad-cores and eight cores on a chip are common,
- Tens and hundreds of cores on a single chip will appear in near future
MapReduce on Multicore

Phoenix [HPCA’07 IISWC’09]

A MapReduce runtime for shared-memory
  > CMPs and SMPs
  > NUMA
MapReduce on Multicore

Phoenix [HPCA’07 IISWC’09]

A MapReduce runtime for shared-memory
> CMPs and SMPs
> NUMA

Features
> Parallelism: *threads*
> Communication: *shared address space*
MapReduce on Multicore

Phoenix [HPCA’07 IISWC’09]

A MapReduce runtime for shared-memory
  > CMPs and SMPs
  > NUMA

Features
  > Parallelism: threads
  > Communication: shared address space

Heavily optimized runtime
  > Runtime algorithm
    e.g. locality-aware task distribution
  > Scalable data structure
    e.g. hash table
  > OS Interaction
    e.g. memory allocator, thread pool
Implementation on Multicore
Implementation on Multicore

Start

Processors

Worker Threads

Disk

Input

Main Memory

Input Buffer

Load
Implementation on Multicore

Start

Processes

Worker Threads

Intermediate Buffer

Disk

Main Memory

Processors

Input
Implementation on Multicore

Start

Workers Threads

Processors

Main Memory

Disk

Input

value array

key array

but

boy

boy

but
Implementation on Multicore

Start

Main Memory

Disk

Input

Processors

Worker Threads

Final Buffer

M M M M M

R R R R R
Implementation on Multicore

Start

Work Worker Threads

Input

Main Memory

Disk

Processes

Start

M M M M

R R R R

Input

Main Memory

Disk

but...

boy
Implementation on Multicore

Start

M M M M M

R R R R R

Merge

Processes

Worker Threads

Main Memory

Disk

Input

Start

Worker Threads

Main Memory

Disk

Input

Result

Output Buffer

... but...

boy

... boy...
Implementation on Multicore

Start

Processors

Disk

Input

Output

Main Memory

Free

Merge

End

Write File
Deficiency of MapReduce on Multicore
High memory usage

- Keep the whole input data in main memory all the time
  e.g. WordCount with 4GB input requires more than 4.3GB memory on Phoenix (93% used by input data)
Deficiency of MapReduce on Multicore

High memory usage

• Keep the whole input data in main memory all the time
  e.g. WordCount with 4GB input requires more than 4.3GB memory on Phoenix (93% used by input data)

Poor data locality

• Process all input data at one time
  e.g. WordCount with 4GB input has about 25% L2 cache miss rate
Deficiency of MapReduce on Multicore

High memory usage

• Keep the whole input data in main memory all the time
  e.g. WordCount with 4GB input requires more than 4.3GB memory on Phoenix (93% used by input data)

Poor data locality

• Process all input data at one time
  e.g. WordCount with 4GB input has about 25% L2 cache miss rate

Strict dependency barriers

• CPU idle at the exchange of phases
Deficiency of MapReduce on Multicore

High memory usage
• Keep the whole input data in main memory all the time

Poor data locality

Strict dependency barriers
• CPU idle at the exchange of phases

Solution: Tiled-MapReduce
Contribution

Tiled-MapReduce programming model
- Tiling strategy
- Fault tolerance (*in paper*)

Three optimizations for Tiled-MapReduce runtime
- Input Data Buffer Reuse
- NUCA/NUMA-aware Scheduler
- Software Pipeline
Outline

1. Tiled MapReduce
2. Optimization on TMR
3. Evaluation
4. Conclusion
Outline

1. Tiled MapReduce
2. Optimization on TMR
3. Evaluation
4. Conclusion
Tiled-MapReduce

“Tiling Strategy”

• Divide a **large** MapReduce job into a number of **independent small** sub-jobs

• Iteratively process **one** sub-job at a time
Tiled-MapReduce

“Tiling Strategy”

- Divide a large MapReduce job into a number of independent small sub-jobs
- Iteratively process one sub-job at a time

Requirement

- Reduce function must be Commutative and Associative
  - all 26 applications in the test suit of Phoenix and Hadoop meet the requirement
Tiled-MapReduce

Extensions to MapReduce Model
Extensions to MapReduce Model

1. Replace the Map phase with a loop of Map and Reduce phases
Extensions to MapReduce Model

1. Replace the **Map** phase with a loop of **Map** and **Reduce** phases

2. Process one sub-job in each iteration
Extensions to MapReduce Model

1. Replace the Map phase with a loop of Map and Reduce phases
2. Process one sub-job in each iteration
3. Rename the Reduce phase within loop to the Combine phase
Extensions to MapReduce Model

1. Replace the Map phase with a loop of Map and Reduce phases

2. Process one sub-job in each iteration

3. Rename the Reduce phase within loop to the Combine phase

4. Modify the Reduce phase to process the partial results of all iterations
Prototype of Tiled-MapReduce

Ostrich: a prototype of Tiled-MapReduce programming model

- Demonstrate the effectiveness of TMR programming model
- Base on Phoenix runtime
- Follow the data structure and algorithms
Ostrich Implementation

Start

Worker Threads

Load

Intermediate Buffer

Main Memory

Processes

Disk

Input
Ostrich Implementation
Ostrich Implementation

Start

Worker Threads

Processes

Disk

Input

Main Memory

Iteration Buffer
Ostrich Implementation
Ostrich Implementation

Worker Threads

Processors

Start

Main Memory

Disk

Input
Ostrich Implementation

![Diagram of Ostrich Implementation]

- **Start**
- **Processors**
  - **Worker Threads**
  - **Main Memory**
  - **Disk**
  - **Input**
  - **Final Buffer**

Legend:
- M: Main Memory
- C: Processor
- R: Worker Thread

This diagram illustrates the flow of data from input through processors to the final buffer, with main memory and disk storage integrated into the process.
Ostrich Implementation

Worker Threads

Processors

Start

Disk

Input

Main Memory

Result

Merge

C C C C C

R R R R R

M M M M M
Ostrich Implementation

Start

MMMM

MM

CCCC

RRRR

Merge

End

Processors

Disk

Output

Input

Main Memory

Free
Outline

1. Tiled MapReduce
2. Optimization on TMR
3. Evaluation
4. Conclusion
OPT1: MEMORY REUSE
OPT1: Memory Reuse

High Memory Usage

• Keep the *whole* input data in memory during the *entire* lifecycle
OPT1: Memory Reuse

High Memory Usage

- Keep the whole input data in memory during the entire lifecycle

Observation

- Only few data in input data is necessary
e.g. WordCount: 1 copy for all duplicated words
OPT1: Memory Reuse

High Memory Usage

• Keep the **whole** input data in memory during the **entire** lifecycle

Observation

• Only **few** data in input data is necessary
  
  e.g. WordCount: 1 copy for all duplicated words

• The aggregation of **these** data improves data locality
OPT1: Memory Reuse

Input Data Memory Reuse

- Copy necessary data to a new buffer in each \textit{Combine} phase
- Only hold the input data of current sub-job in memory
- Reuse the Input Buffer among sub-jobs
**OPT1: Memory Reuse**

Extension of Interface

- Provide 2 optional interfaces
  - **Acquire**: load input data to memory
  - **Release**: free input data from memory

- The counterparts in other runtimes

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ostrich</td>
<td>acquire</td>
</tr>
<tr>
<td></td>
<td>release</td>
</tr>
<tr>
<td>Google MapReduce</td>
<td>reader</td>
</tr>
<tr>
<td></td>
<td>writer</td>
</tr>
<tr>
<td>Hadoop</td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td>close</td>
</tr>
</tbody>
</table>
Input Data Memory Reuse

Worker Threads

Start

Processors

Disk

Main Memory

Input

Load

acquire

Input Buffer

...
Input Data Reuse

Start

Processors

Worker Threads

Disk

Main Memory

Input

Disk

Baby...

But.

...
Input Data Reuse

Start

Disk

Input

Main Memory

Processes

Worker Threads

New Buffer

Baby...

But.
Input Data Reuse

Start

Disk

Main Memory

Processors

Worker Threads

..Baby...

..But.

New Buffer

Baby

But
Input Data Reuse

Start

M M M M M
C C C C C

Processes

Worker Threads

Input

Main Memory

Disk

Free

release

Baby
But

...
OPT2: Locality Optimization
OPT2: Locality Optimization

Poor Data Locality of MapReduce runtime on Multicore

• Process all input data in one time
OPT2: Locality Optimization

Poor Data Locality of MapReduce runtime on Multicore

• Process all input data in one time

Tiled-MapReduce improves data locality

• Make the working set of each sub-job fit into the last level cache

• Aggregate partial results in Combine phase (in OPT1)
OPT2: Locality Optimization

Memory Hierarchy

• Multicore hardware usually organizes caches in a non-uniform cache access (NUCA) way
• The cross-chip operations are expensive*
  e.g. Local/Remote L2 cache: 14/110 cycles*

* Intel 16-Core Machine with 4 Xeon 1.6GHz Quad-cores chips
Memory Hierarchy

- Multicore hardware usually organizes caches in a non-uniform cache access (NUCA) way
- The cross-chip operations are expensive*
  
  e.g. Local/Remote L2 cache: 14/110 cycles*

NUCA/NUMA-aware scheduler

- Eliminate remote cache and memory access
- Run each sub-job on a single chip

* Intel 16-Core Machine with 4 Xeon 1.6GHz Quad-cores chips
NuCA/NUMA-Aware Scheduler

Diagram showing the master/worker relationships and shared cache between cores.
NuCA/NUMA-Aware Scheduler

- master
- group
- repeater/worker
- worker
- worker
- worker
- worker
- worker
- worker
- worker
- worker
- shared cache
- main memory
- main memory
- main memory
- main memory
- core
- core
- core
- core
- $  
- $  
- $  
- $  
- shared cache
- core
- core
- core
- core
- $  
- $  
- $  
- $  
- main memory
- main memory
- main memory
- main memory
NuCA/NUMA-Aware Scheduler

- Master
- Repeater/Worker
- Worker

Job Queue

Core

Shared Cache

Main Memory
NuCA/NUMA-Aware Scheduler

The diagram illustrates the workflow of NuCA/NUMA-Aware Scheduler. It consists of a master node and several worker nodes. Each worker node has a job queue to manage incoming jobs. The scheduler is designed to efficiently distribute jobs across the worker nodes by utilizing a shared cache and main memory. Intermediate Buffer and Final Buffer are used to store intermediate and final results, respectively. The diagram also shows the iteration buffer, indicating the sequence of processing steps. The layout is optimized to ensure efficient data flow and reduce latency.
OPT3: CPU Optimization
Data Dependency

- Strict **barrier** after map and reduce phase
- The execution time of a job is determined by the **slowest** worker in each phase
OPT3: CPU Optimization

Data Dependency

- Strict barrier after map and reduce phase
- The execution time of a job is determined by the slowest worker in each phase

Observation

- No data dependency between one sub-job’s Combine phase and its successor’s Map phase
Software Pipeline

- Overlap the *Combine* phase of the current sub-job and the *Map* phase of its successor
Software Pipeline

Map
Combine
Idle
Software Pipeline

Map
Combine
Idle

Time

Barriers
Software Pipeline

- Red: Map
- Blue: Combine
- White: Idle

Time

- Software Pipeline

Speedup

Diagram showing the execution of tasks across cores.
Outline

1. Tiled MapReduce
2. Optimization on TMR
3. Evaluation
4. Conclusion
Configuration

Platform

Intel 16-Core machine (4 Quad-cores chips)
32GB Main Memory
Debian Linux with kernel v2.6.24

Systems:

Phoenix-2 with streamflow *
Ostrich with streamflow

* Scalable locality-conscious multithreaded memory allocation - ISMM’06
## Applications

<table>
<thead>
<tr>
<th>Applications</th>
<th>Key</th>
<th>Duplicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount (WC)</td>
<td>many</td>
<td>many</td>
</tr>
<tr>
<td>Distributed Sort (DS)</td>
<td>many</td>
<td>no</td>
</tr>
<tr>
<td>Log Statistics (LS)</td>
<td>few</td>
<td>many</td>
</tr>
<tr>
<td>Inverted Index (II)</td>
<td>one</td>
<td>few</td>
</tr>
</tbody>
</table>
# Burden of Programmer

## Code Modification

- Support input data memory reuse

<table>
<thead>
<tr>
<th>Applications</th>
<th>Acquire</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount (WC)</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Distributed Sort (DS)</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>Log Statistics (LS)</td>
<td>Default</td>
<td>Default</td>
</tr>
<tr>
<td>Inverted Index (II)</td>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>
Overall Performance

- WC: 3.3X
- DS: 1.2X
- LS: 1.2X
Related Work

Other extension to MapReduce model

- Database: Map-reduce-merge [SIGMOD’07]
- Online Aggregation: MapReduce Online [NSDI’10]
- ...

Other implementation of MapReduce runtime

- Cluster: Hadoop [Apache, OSDI’08]
- Shared Memory: Phoenix [HPCA’07, IISWC’09] and Metis [MIT-TR]
- GPGPU: Mars [PACT’07]
- Heterogeneous: MapCG [PACT’10]
- ...


Conclusion

- Environments differences between cluster and multicore open new design spaces and optimization opportunities

- Tiled-MapReduce and the three optimizations

- Ostrich outperforms Phoenix by up to 3.3X
Thanks

Ostrich
The top land speed and the largest of bird

Questions?

Parallel Processing Institute
http://ppi.fudan.edu.cn
Memory Consumption

![Memory Consumption Graph](image)

- **X-axis**: PHO-1, PHO-2, PHO-3, PHO-4, OST-1, OST-2, OST-3, OST-4
- **Y-axis**: Memory Consumption (GB)
- **Legend**:
  - Intermediate
  - Input

The graph shows the memory consumption in GB for different input types (PHO, OST) and different stages (WC, DS, LS, II). The memory consumption varies across different scenarios, with some stages and input types requiring significantly more memory than others.
Without NUCA/NUMA-Aware Scheduler

With NUCA/NUMA-Aware Scheduler

Speedup

WC

DS

LS

II
Exploit Locality

L2 Cache Miss Rate

<table>
<thead>
<tr>
<th>WC</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0%</td>
<td>5%</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DS</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0%</td>
<td>5%</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LS</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0%</td>
<td>5%</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>II</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0%</td>
<td>5%</td>
<td>10%</td>
</tr>
</tbody>
</table>
Software Pipeline

Execution Time (Sec)

- **WC**
  - Merge: 3
  - Combine (Idle): 1
  - Combine (Active): 1
  - Map: 3

- **DS**
  - Merge: 1
  - Combine (Idle): 1
  - Combine (Active): 1
  - Map: 3

- **LS**
  - Merge: 3
  - Combine (Idle): 1
  - Combine (Active): 1
  - Map: 3

- **II**
  - Merge: 3
  - Combine (Idle): 1
  - Combine (Active): 1
  - Map: 3

- **II/P**
  - Merge: 3
  - Combine (Idle): 1
  - Combine (Active): 1
  - Map: 3