CS 61C: Great Ideas in Computer Architecture

Warehouse-Scale Computers and Map Reduce

Guest Lecturer: Raphael Townshend
Review of Last Lecture

- **Disk Terminology**: spindle, platter, track, sector, actuator, arm, head, non-volatile
- Disk Latency = Seek Time + Rotation Time + Transfer Time + Controller Overhead
- Processor must synchronize with I/O devices before use due to difference in data rates:
  - Polling works, but expensive due to repeated queries
  - Exceptions are “unexpected” events in processor
  - Interrupts are asynchronous events that are often used for interacting with I/O devices
- In SW, need special handling code
New-School Machine Structures
(It’s a bit more complicated!)

Today’s Lecture

- Parallel Requests
  Assigned to computer
e.g., Search “Garcia”

- Parallel Threads
  Assigned to core
e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
e.g., Add of 4 pairs of words

- Hardware descriptions
  All gates @ one time
Agenda

- Warehouse-Scale Computers
- Administrivia
- Request Level Parallelism
- Map Reduce
  - Data Level Parallelism
Why Cloud Computing Now?

• "The Web Space Race": Build-out of extremely large datacenters (10,000’s of commodity PCs)
  – Build-out driven by growth in demand (more users)
  ⇒ Infrastructure software and Operational expertise

• Discovered economy of scale: 5-7x cheaper than provisioning a medium-sized (1000 servers) facility

• More pervasive broadband Internet so can access remote computers efficiently

• Commoditization of HW & SW
  – Standardized software stacks
Supercomputer for hire

• Top 500 supercomputer competition
• 290 Eight Extra Large (@ $2.40/hour) = 240 TeraFLOPS
• 42nd/500 supercomputer @ ~$700 per hour
• Credit card => can use 1000s computers
• FarmVille on AWS
  – Prior biggest online game 5M users
  – What if startup had to build datacenter? How big?
  – 4 days =1M; 2 months = 10M; 9 months = 75M
Warehouse Scale Computers

• Massive scale datacenters: 10,000 to 100,000 servers + networks to connect them together
  – Emphasize cost-efficiency
  – Attention to power: distribution and cooling

• (relatively) homogeneous hardware/software

• Offer very large applications (Internet services): search, social networking, video sharing

• Very highly available: <1 hour down/year
  – Must cope with failures common at scale

• “…WSCs are no less worthy of the expertise of computer systems architects than any other class of machines” Barroso and Hoelzle 2009
Design Goals of a WSC

• Unique to Warehouse-scale
  – *Ample parallelism*:
    • Batch apps: large number independent data sets with independent processing.
  – *Scale and its Opportunities/Problems*
    • Relatively small number of WSC make design cost expensive and difficult to amortize
    • But price breaks are possible from purchases of very large numbers of commodity servers
    • Must also prepare for high component failures
  – *Operational Costs Count*:
    • Cost of equipment purchases << cost of ownership
E.g., Google’s Oregon WSC
Containers in WSCs

Inside WSC

Inside Container
Equipment Inside a WSC

Server (in rack format):
1 ¾ inches high “1U”,
x 19 inches x 16-20 inches: 8 cores, 16 GB DRAM, 4x1 TB disk

7 foot Rack: 40-80 servers + Ethernet local area network (1-10 Gbps) switch in middle (“rack switch”)

Array (aka cluster):
16-32 server racks + larger local area network switch (“array switch”)
10X faster => cost 100X:
cost $f(N^2)$
Server, Rack, Array
Coping with Performance in Array

Lower latency to DRAM in another server than local disk
Higher bandwidth to local disk than to DRAM in another server

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Rack</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Racks</td>
<td>--</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Servers</td>
<td>1</td>
<td>80</td>
<td>2400</td>
</tr>
<tr>
<td>Cores (Processors)</td>
<td>8</td>
<td>640</td>
<td>19,200</td>
</tr>
<tr>
<td>DRAM Capacity (GB)</td>
<td>16</td>
<td>1,280</td>
<td>38,400</td>
</tr>
<tr>
<td>Disk Capacity (GB)</td>
<td>4,000</td>
<td>320,000</td>
<td>9,600,000</td>
</tr>
<tr>
<td>DRAM Latency (microseconds)</td>
<td>0.1</td>
<td>100</td>
<td>300</td>
</tr>
<tr>
<td>Disk Latency (microseconds)</td>
<td>10,000</td>
<td>11,000</td>
<td>12,000</td>
</tr>
<tr>
<td>DRAM Bandwidth (MB/sec)</td>
<td>20,000</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>Disk Bandwidth (MB/sec)</td>
<td>200</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>
Coping with Workload Variation

- Online service: Peak usage 2X off-peak
Impact of latency, bandwidth, failure, varying workload on WSC software?

• WSC Software must take care where it places data within an array to get good performance
• WSC Software must cope with failures gracefully
• WSC Software must scale up and down gracefully in response to varying demand
• More elaborate hierarchy of memories, failure tolerance, workload accommodation makes WSC software development more challenging than software for single computer
Power vs. Server Utilization

- Server power usage as load varies idle to 100%
- Uses $\frac{1}{2}$ peak power when idle!
- Uses $\frac{2}{3}$ peak power when 10% utilized! 90%@ 50%!
- Most servers in WSC utilized 10% to 50%
- Goal should be *Energy-Proportionality*: % peak load = % peak energy
Power Usage Effectiveness

• Overall WSC Energy Efficiency: amount of computational work performed divided by the total energy used in the process

• Power Usage Effectiveness (PUE):
  Total building power / IT equipment power
  – An power efficiency measure for WSC, not including efficiency of servers, networking gear
  – 1.0 = perfection
PUE in the Wild (2007)

FIGURE 5.1: LBNL survey of the power usage efficiency of 24 datacenters, 2007 (Greenberg et al.)
High PUE: Where Does Power Go?

- Uninterruptable Power Supply (battery)
- Power Distribution Unit
- Servers + Networking
- Chiller cools warm water from Air Conditioner
- Computer Room Air Conditioner
Google WSC A PUE: 1.24

1. Careful air flow handling
2. Elevated cold aisle temperatures
3. Use of free cooling
4. Per-server 12-V DC UPS
5. Measure vs. estimate PUE, publish PUE, and improve operation
Agenda

- Warehouse-Scale Computers
- Administrivia
- Request Level Parallelism
- Map Reduce
  - Data Level Parallelism
Administrivia

- Project 3 (individual) due Sunday 8/5
- Final Review – Friday 8/3, 3-6pm in 306 Soda
- Final – Thurs 8/9, 9am-12pm, 245 Li Ka Shing
  - Focus on 2nd half material, though midterm material still fair game
  - MIPS Green Sheet provided again
  - Two-sided handwritten cheat sheet
    - Can use the back side of your midterm cheat sheet!
- Lecture tomorrow by Paul
Agenda

• Warehouse-Scale Computers
• Administrivia
• Request Level Parallelism
• Map Reduce
  – Data Level Parallelism
Request-Level Parallelism (RLP)

• Hundreds or thousands of requests per second
  – Not your laptop or cell-phone, but popular Internet services like web search, social networking,…
  – Such requests are largely independent
    • Often involve read-mostly databases
    • Rarely involve strict read–write data sharing or synchronization across requests

• Computation easily partitioned within a request and across different requests
Google Query-Serving Architecture
Anatomy of a Web Search

- Google “Justin Hsia”
Anatomy of a Web Search (1 of 3)

• Google “Justin Hsia”
  – Direct request to “closest” Google Warehouse Scale Computer
  – Front-end load balancer directs request to one of many arrays (cluster of servers) within WSC
  – Within array, select one of many Google Web Servers (GWS) to handle the request and compose the response pages
  – GWS communicates with Index Servers to find documents that contain the search words, “Justin”, “Hsia”, uses location of search as well
  – Return document list with associated relevance score
Anatomy of a Web Search (2 of 3)

• In parallel,
  – Ad system: run ad auction for bidders on search terms
  – Get Images of various Justin Hsias

• Use docids (document IDs) to access indexed documents

• Compose the page
  – Result document extracts (with keyword in context) ordered by relevance score
  – Sponsored links (along the top) and advertisements (along the sides)
Anatomy of a Web Search (3 of 3)

• Implementation strategy
  – Randomly distribute the entries
  – Make many copies of data (aka “replicas”)
  – Load balance requests across replicas

• Redundant copies of indices and documents
  – Breaks up hot spots, e.g., “Justin Bieber”
  – Increases opportunities for request-level parallelism
  – Makes the system more tolerant of failures
Agenda

• Warehouse-Scale Computers
• Administrivia
• Request Level Parallelism
• Map Reduce
  – Data Level Parallelism
Data-Level Parallelism (DLP)

• 2 kinds
  1. Lots of data in memory that can be operated on in parallel (e.g., adding together 2 arrays)
  2. Lots of data on many disks that can be operated on in parallel (e.g., searching for documents)

• SIMD does Data Level Parallelism (DLP) in memory

• Today’s lecture and lab 12 does DLP across many servers and disks using MapReduce
What is MapReduce?

• Simple data-parallel programming model designed for scalability and fault-tolerance

• Pioneered by Google
  – Processes >25 petabytes of data per day

• Popularized by open-source Hadoop project
  – Used at Yahoo!, Facebook, Amazon, ...
What is MapReduce used for?

• At Google:
  – Index construction for Google Search
  – Article clustering for Google News
  – Statistical machine translation
  – For computing multi-layer street maps

• At Yahoo!:
  – “Web map” powering Yahoo! Search
  – Spam detection for Yahoo! Mail

• At Facebook:
  – Data mining
  – Ad optimization
  – Spam detection
Example: Facebook Lexicon

www.facebook.com/lexicon (no longer available)
MapReduce Design Goals

1. Scalability to large data volumes:
   - 1000’s of machines, 10,000’s of disks

2. Cost-efficiency:
   - Commodity machines (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer administrators)
   - Easy to use (fewer programmers)

MapReduce Solution

• Apply Map function to user supplied record of key/value pairs
  – Compute set of intermediate key/value pairs
• Apply Reduce operation to all values that share same key in order to combine derived data properly
• User supplies Map and Reduce operations in functional model
  – so can parallelize,
  – can use re-execution for fault tolerance
Data-Parallel “Divide and Conquer” (MapReduce Processing)

• Map:
  – Slice data into “shards” or “splits”, distribute these to workers, compute sub-problem solutions
    \[ \text{map(in key, in value)} \rightarrow \text{list(out key, intermediate value)} \]
    • Processes input key/value pair
    • Produces set of intermediate pairs

• Reduce:
  – Collect and combine sub-problem solutions
    \[ \text{reduce(out_key, list(intermediate_value))} \rightarrow \text{list(out_value)} \]
    • Combines all intermediate values for a particular key
    • Produces a set of merged output values (usually just one)

• Fun to use: focus on problem, let MapReduce library deal with messy details
Typical Hadoop Cluster

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort): 8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)
MapReduce Execution

Fine granularity tasks: many more map tasks than machines

2000 servers =>
≈ 200,000 Map Tasks,
≈ 5,000 Reduce tasks
MapReduce Processing
Example: Count Word Occurrences

map(String input key, String input value):
   // input_key: document name
   // input_value: document contents
   for each word w in input_value:
      EmitIntermediate(w, "1"); // Produce count of words

reduce(String output key, Iterator intermediate values):
   // output_key: a word
   // output_values: a list of counts
   int result = 0;
   for each v in intermediate_values:
      result += ParseInt(v); // get integer from key-value
   Emit(AsString(result));
MapReduce Processing

Shuffle phase
MapReduce Processing

1. MR 1st splits the input files into $M$ “splits” then starts many copies of program on servers
2. One copy—the master—is special. The rest are workers. The master picks idle workers and assigns each 1 of M map tasks or 1 of R reduce tasks.
MapReduce Processing

3. A map worker reads the input split. It parses key/value pairs of the input data and passes each pair to the user-defined map function. (The intermediate key/value pairs produced by the map function are buffered in memory.)
4. Periodically, the buffered pairs are written to local disk, partitioned into $R$ regions by the partitioning function.
5. When a reduce worker has read all intermediate data for its partition, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. (The sorting is needed because typically many different keys map to the same reduce task.)
MapReduce Processing

6. Reduce worker iterates over sorted intermediate data and for each unique intermediate key, it passes key and corresponding set of values to the user’s reduce function.

The output of the reduce function is appended to a final output file for this reduce partition.
MapReduce Processing

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. The MapReduce call in user program returns back to user code.

Output of MR is in $R$ output files (1 per reduce task, with file names specified by user); often passed into another MR job.
MapReduce Processing Time Line

- Master assigns map + reduce tasks to “worker” servers
- As soon as a map task finishes, worker server can be assigned a new map or reduce task
- Data shuffle begins as soon as a given Map finishes
- Reduce task begins as soon as all data shuffles finish
- To tolerate faults, reassign task if a worker server “dies”
Another Example: Word Index (How Often Does a Word Appear?)

Distribute

that that is is that that is not is not is that it it is
Map 1 Map 2 Map 3 Map 4
is 1, that 2 is 1, that 2 is 2, not 2 is 2, it 2, that 1

Shuffle

is 1,1,2,2 that 2,2,1, not 2,1 Reduce 1 Reduce 2
is 6; it 2 not 2; that 5

Collect

is 6; it 2; not 2; that 5
MapReduce Failure Handling

• On worker failure:
  – Detect failure via periodic heartbeats
  – Re-execute completed and in-progress map tasks
  – Re-execute in progress reduce tasks
  – Task completion committed through master

• Master failure:
  – Protocols exist to handle (master failure unlikely)

• Robust: lost 1600 of 1800 machines once, but finished fine (story from Google?)
MapReduce Redundant Execution

• Slow workers significantly lengthen completion time
  – Other jobs consuming resources on machine
  – Bad disks with soft errors transfer data very slowly
  – Weird things: processor caches disabled (!!)

• Solution: Near end of phase, spawn backup copies of tasks
  – Whichever one finishes first "wins"

• Effect: Dramatically shortens job completion time
  – 3% more resources, large tasks 30% faster
Summary (1/2)

• Parallelism applies at many levels, from instructions to data to within WSC

• WSC
  – SW must cope with failures, varying load, varying HW latency bandwidth
  – HW sensitive to cost, energy efficiency
  – Supports many of the applications we have come to depend on
Summary (2/2)

• Request Level Parallelism
  – High request volume, each largely independent
  – Replication for better throughput, availability

• Map Reduce Data Parallelism
  – Divide large data set into pieces for independent parallel processing
  – Combine and process intermediate results to obtain final result