New-School Machine Structures
(It’s a bit more complicated!)

- Parallel requests
  - Assigned to computer
  - e.g., Search “Katz”
- 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

Today’s Lecture

Review

- Cloud Computing
  - Benefits of WSC computing for all parties involved
  - “Elastic” pay as you go resource allocation
  - Amazon demonstrates that you can make money by selling cycles and storage
- Warehouse-Scale Computers (WSCs)
  - Power ultra large-scale Internet applications
  - Emphasis on cost, power efficiency
  - PUE - Ratio of total power consumed over power used for computing

Agenda

- Request Level Parallelism
  - MapReduce
  - Administrivia
  - MapReduce Processing
  - Break
  - Modern Microarchitectures

Request-Level Parallelism (RLP)

- Lots of independent tasks
  - In a WSC for a large internet app, hundreds to thousands per second.
    - Mostly involve read-only databases
    - Little read-write (aka “producer-consumer”) sharing
    - Rarely involve 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 “Randy H. Katz”
  – 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, “Randy”, “Katz”, uses location of search as well
  – Return document list with associated relevance score

In parallel,
  – Ad system: books by Katz at Amazon.com
  – Images of Randy Katz
  – 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)

Implementation strategy
  – Randomly distribute the entries
  – Make many copies of data (aka “replicas”) on many servers
  – Load balance requests across replicas
  – Redundant copies of indices and documents break up hot spots, e.g., “Justin Bieber”
  – Increases opportunities for request-level parallelism
  – Makes the system more tolerant of failures

Agenda

• Request Level Parallelism
• MapReduce
• Administrivia
• MapReduce Processing
• Break
• Modern Microarchitectures

Data-Level Parallelism (DLP)

• Lots of data that can be operated on in parallel
• We’ve already seen SIMD instructions as a way to exploit this
• What about larger scale DLP? Lots of data spread across many disks? WSC level quantities of data?
Problem Trying To Solve

• How process large amounts of raw data (crawled documents, request logs, ...) every
day to compute derived data (inverted
indices, page popularity, ...) when
computation conceptually simple but input
data large and distributed across 100s to
1000s of servers?
• Challenge: Distribute and load balance data,
tolerate faults.
  • Jeffrey Dean and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on

MapReduce Solution

• Reduce the complexity of an otherwise
complex parallel programming task by
providing a rigid structure.
• Create a robust implementation
  — Fault tolerant, handles all the pesky details
• To solve problem, simply formulate it in the
MapReduce format
  — Surprisingly versatile, can also chain together
multiple MapReduce tasks.

Inspiration: cs61a map/reduce

• (define (square x) (* x x))
  (reduce + (map square '(1 2 3 4)))
• Map - Applies the operation “square” to
every element in the list (result: ‘(1 4 9 16))
• Reduce - Combines the resulting data using
the provided operator + (1 + 4 + 9 + 16 => 30).

MapReduce Format

• 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

Data-Parallel “Divide and Conquer”
(MapReduce Processing)

• Map:
  – Map: (K,V) -> list((K1,V1), (K2,V2), ..., (Kn,Vn))
    • Processes input key/value pair
    • Produces set of intermediate key/value pairs
  – Slice data into “shards” or “splits”, distribute these to
workers, compute sub-problem solutions
• Reduce:
  – Reduce: (K, list(V1, V2, ..., Vn)) -> (K, combinedValue)
    • Combines all intermediate values for a particular key
    • Produces a set of merged output values (usually just one)
  – Collect and combine sub-problem solutions

MapReduce Execution

Fine granularity tasks: many
more map tasks
than machines

Input

Intermediate

Group by Key

2000 servers =>
+ 200,000 Map Tasks,
+ 5,000 Reduce tasks

Machine

Output

8/8/2011
Google Uses MapReduce For ...

- Extracting the set of outgoing links from a collection of HTML documents and aggregating by target document
- Stitching together overlapping satellite images to remove seams and to select high-quality imagery for Google Earth
- Generating a collection of inverted index files using a compression scheme tuned for efficient support of Google search queries
- Processing all road segments in the world and rendering map tile images that display these segments for Google Maps
- Fault-tolerant parallel execution of programs written in higher-level languages across collections of input data
- More than 10,000 MR programs at Google in 4 years

MapReduce Popularity at Google

<table>
<thead>
<tr>
<th></th>
<th>Aug-04</th>
<th>Mar-06</th>
<th>Sep-07</th>
<th>Sep-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jobs</td>
<td>29,000</td>
<td>171,000</td>
<td>2,217,000</td>
<td>3,467,000</td>
</tr>
<tr>
<td>Avg. completion time (secs)</td>
<td>634</td>
<td>874</td>
<td>395</td>
<td>475</td>
</tr>
<tr>
<td>Server years used</td>
<td>217</td>
<td>2,002</td>
<td>11,081</td>
<td>25,562</td>
</tr>
<tr>
<td>Input data read (TB)</td>
<td>3,288</td>
<td>52,254</td>
<td>403,152</td>
<td>544,130</td>
</tr>
<tr>
<td>Intermediate data (TB)</td>
<td>758</td>
<td>6,743</td>
<td>34,774</td>
<td>90,120</td>
</tr>
<tr>
<td>Output data written (TB)</td>
<td>193</td>
<td>2,970</td>
<td>14,018</td>
<td>57,520</td>
</tr>
<tr>
<td>Average number servers/job</td>
<td>157</td>
<td>268</td>
<td>394</td>
<td>488</td>
</tr>
</tbody>
</table>

What if Ran Google Workload on EC2?

<table>
<thead>
<tr>
<th></th>
<th>Aug-04</th>
<th>Mar-06</th>
<th>Sep-07</th>
<th>Sep-09</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jobs</td>
<td>29,000</td>
<td>171,000</td>
<td>2,217,000</td>
<td>3,467,000</td>
</tr>
<tr>
<td>Avg. completion time (secs)</td>
<td>634</td>
<td>874</td>
<td>395</td>
<td>475</td>
</tr>
<tr>
<td>Server years used</td>
<td>217</td>
<td>2,002</td>
<td>11,081</td>
<td>25,562</td>
</tr>
<tr>
<td>Input data read (TB)</td>
<td>3,288</td>
<td>52,254</td>
<td>403,152</td>
<td>544,130</td>
</tr>
<tr>
<td>Intermediate data (TB)</td>
<td>758</td>
<td>6,743</td>
<td>34,774</td>
<td>90,120</td>
</tr>
<tr>
<td>Output data written (TB)</td>
<td>193</td>
<td>2,970</td>
<td>14,018</td>
<td>57,520</td>
</tr>
<tr>
<td>Average number servers/job</td>
<td>157</td>
<td>268</td>
<td>394</td>
<td>488</td>
</tr>
<tr>
<td>Average Cost/job EC2</td>
<td>$0.5M</td>
<td>$6.7M</td>
<td>$57.4M</td>
<td>$133.1M</td>
</tr>
</tbody>
</table>

Agenda

- Request Level Parallelism
- MapReduce
- Administrivia
- MapReduce Processing
- Break
- Modern Microarchitectures
Agenda

- Request Level Parallelism
- MapReduce
- Administrivia
- MapReduce Processing
- Break
- Modern Microarchitectures

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));
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.)

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.

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”
Example MapReduce Job Running

- ~41 minutes total
  - ~29 minutes for Map tasks & Shuffle tasks
  - ~12 minutes for Reduce tasks
  - 1707 worker servers used
- Map (Green) tasks read 0.8 TB, write 0.5 TB
- Shuffle (Red) tasks read 0.5 TB, write 0.5 TB
- Reduce (Blue) tasks read 0.5 TB, write 0.5 TB
### MapReduce status: MR_INDEXER-large-2003_10_28_00_03

**Started Fri Nov 7 09:51:07 2003** — up 0 hr 31 min 34 sec

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13931</td>
<td>13931</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.2</td>
<td>523499.2</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.5</td>
<td>528937.8</td>
<td>530526.6</td>
</tr>
</tbody>
</table>

**Start Time:** 8/8/2011  
**Duration:** Summer 2011 – Lecture #28

---

### MapReduce status: MR_INDEXER-large-2003_10_28_00_03

**Started Fri Nov 7 09:51:07 2003** — up 0 hr 31 min 34 sec

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13931</td>
<td>13931</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.2</td>
<td>523499.2</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.5</td>
<td>528937.8</td>
<td>530526.6</td>
</tr>
</tbody>
</table>

**Start Time:** 8/8/2011  
**Duration:** Summer 2011 – Lecture #28

---

### MapReduce status: MR_INDEXER-large-2003_10_28_00_03

**Started Fri Nov 7 09:51:07 2003** — up 0 hr 31 min 34 sec

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13931</td>
<td>13931</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.2</td>
<td>523499.2</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.5</td>
<td>528937.8</td>
<td>530526.6</td>
</tr>
</tbody>
</table>

**Start Time:** 8/8/2011  
**Duration:** Summer 2011 – Lecture #28

---

### MapReduce status: MR_INDEXER-large-2003_10_28_00_03

**Started Fri Nov 7 09:51:07 2003** — up 0 hr 31 min 34 sec

<table>
<thead>
<tr>
<th>Type</th>
<th>Shards</th>
<th>Done</th>
<th>Active</th>
<th>Input(MB)</th>
<th>Done(MB)</th>
<th>Output(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>13931</td>
<td>13931</td>
<td>0</td>
<td>878934.6</td>
<td>878934.6</td>
<td>523499.2</td>
</tr>
<tr>
<td>Shuffle</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.2</td>
<td>523499.2</td>
<td>523499.2</td>
</tr>
<tr>
<td>Reduce</td>
<td>500</td>
<td>500</td>
<td>0</td>
<td>523499.5</td>
<td>528937.8</td>
<td>530526.6</td>
</tr>
</tbody>
</table>

**Start Time:** 8/8/2011  
**Duration:** Summer 2011 – Lecture #28
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:
  – Could handle, but don’t yet (master failure unlikely)
• Robust: lost 1600 of 1800 machines once, but finished fine

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

Impact of Redundant Execution, Failure for 10B record Sort using 1800 servers

Agenda

• Request Level Parallelism
• MapReduce
• Administrivia
• MapReduce Processing
• Break
• Modern Microarchitectures

Intel Nehalem

• Look at microprocessor from Intel in servers in your 61C labs and your laptops
• Nehalem is code name for microarchitecture at heart of Core i7 and Xeon 5500 series server chips
  – Intel legal said had to pick names of rivers for code names (vs. “Destroyer”)
• First released at end of 2008
• Die size 263mm² at 45 nm
• 731M transistors

Nehalem River, Oregon
Nehalem System Example: 
Apple Mac Pro Desktop 2010

Two Nehalem Chips ("Sockets"), each containing four processors ("cores") running at up to 2.93GHz 
Can have up to two DIMMs on each channel (up to 4GB/DIMM) 
Disk drives attached with 30Gb/s serial ATA link 

PCI Express connections for Graphics cards and other extension boards. Up to 8 GB/s per slot. 

Slower peripherals (Ethernet, USB, Firewire, WiFi, Bluetooth, Audio)

Nehalem Die Photo

Shared L3 Cache

Core Area Breakdown

Memory Controller

Load Store Queue

Extending Performance and Energy Efficiency

Accelerated String and Test Processing

Accelerated Searching & Pattern Recognition of Large Data Sets

New Communications Capabilities

What should the applications, OS and VMM vendors do? Understand the benefits & take advantage of new instructions in 2008. Provide us feedback on instructions ISV would like to see for next generation of applications
x86 Decoding

- Translate up to 4 x86 instructions into μOPS (=MIPS or RISC instructions) each cycle
- Only first x86 instruction in group can be complex (maps to 1-4 μOPS), rest must be simple (map to one μOP)
- Even more complex instructions, jump into microcode engine which spits out stream of μOPS

Branch Prediction

- Part of instruction fetch unit
- Several different types of branch predictor
  - Details not public
- Two-level Branch Table Buffer
- Loop count predictor
  - How many backwards taken branches before loop exit
- Return Stack Buffer
  - Holds subroutine targets
  - Separate return stack buffer for each SMT thread

Loop Stream Detectors save Power

Out-of-Order Execution Engine

Renaming happens at uOP level (not original macro-x86 instructions)
Multithreading effects in Out-of-Order Execution Core

- Reorder buffer (remembers program order and exception status for in-order commit) has 128 entries divided statically and equally between both threads
- Reservation stations (instructions waiting for operands for execution) have 36 entries competitively shared by threads

Non-Uniform Memory Access (NUMA)

- FSB architecture - All memory in one location
- Starting with Intel® Core™ microarchitecture (Nehalem) - Memory located in multiple places
- Latency to memory dependent on location
- Local memory has highest BW, lowest latency
- Remote Memory still very fast

Ensure software is NUMA-optimized for best performance

Core’s Private Memory System

- Load queue 48 entries
- Store queue 32 entries
- Divided statically between 2 threads
- Up to 16 outstanding misses in flight per core

Nehalem Memory Hierarchy Overview

- 32KB L1 DS
- 256KB L2
- 3MB Shared L3
- 8MB Shared L3
- 3 DDR3 DRAM Memory Controllers
- QuickPath System Interconnect

Each DRAM Channel is 64/72b wide at up to 1.33Gb/s
Each direction is 20b@6.4Gb/s

Each L3 fully inclusive of higher levels, but L2 not inclusive of L1.

Other sockets’ caches kept coherent using QuickPath messages.

All Sockets can Access all Data

How ensure that get data allocated to local DRAM?
OS doesn’t allocate pages to physical memory after malloc until first access to page. Be sure to touch what each CPU wants nearby.

Such systems called “NUMA” for Non Uniform Memory Access: Some addresses are slower than others.

Local memory access latency ~60ns
Remote memory access latency ~100ns

8/8/2011  Fall 2010  -- Lecture #38
What to do with So Many Features?

- “Introduction to Performance Analysis on Nehalem Based Processors”, 72 pages

“Software optimization based on performance analysis of large existing applications, in most cases, reduces to optimizing the code generation by the compiler and optimizing the memory access. Optimizing the code generation by the compiler requires inspection of the assembler of the time consuming parts of the application and verifying that the compiler generated a reasonable code stream. Optimizing the memory access is a complex issue involving the bandwidth and latency capabilities of the platform, hardware and software prefetching efficiencies and the virtual address layout of the heavily accessed variables.”

Summary

- Request-Level Parallelism
  - High request volume, each largely independent of other
  - Use replication for better request throughput, availability
- MapReduce Data Parallelism
  - Divide large data set into pieces for independent parallel processing
  - Combine and process intermediate results to obtain final result
- Intel Nehalem
  - Speculative execution: branch prediction, out of order execution, data prefetching
  - Hardware translation and optimization of instruction sequences
  - Opportunistic acceleration (Turbo Mode)

Acknowledgements

- Figures/Info from Intel, David Kanter at Real World Technologies.
- These slides contain material developed and copyright by:
  - Krste Asanovic (UCB)
  - Beeman Strong (Intel)
- UCB material derived from course CS152.