Scalable Cloud Computing

Keijo Heljanko

Department of Information and Computer Science
School of Science
Aalto University
`keijo.heljanko@aalto.fi`

5.6-2012

Business Drivers of Cloud Computing

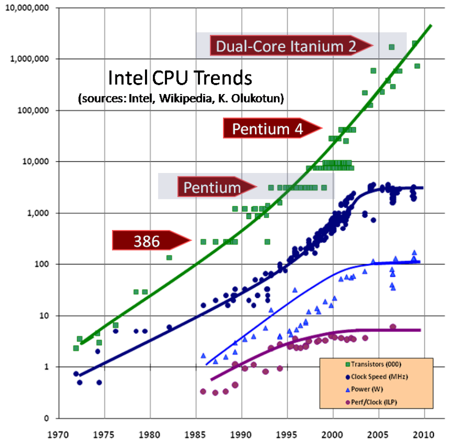
- Large data centers allow for economics of scale
  - Cheaper hardware purchases
  - Cheaper cooling of hardware
    - Example: Google paid 40 MEur for a Summa paper mill site in Hamina, Finland: Data center cooled with sea water from the Baltic Sea
  - Cheaper electricity
  - Cheaper network capacity
  - Smaller number of administrators / computer
- Unreliable commodity hardware is used
- Reliability obtained by replication of hardware components and a combined with a fault tolerant software stack

Cloud Computing Technologies

A collection of technologies aimed to provide elastic "pay as you go" computing

- Virtualization of computing resources: Amazon EC2, Eucalyptus, OpenNebula, Open Stack Compute, . . .
- Scalable file storage: Amazon S3, GFS, HDFS, . . .
- Scalable batch processing: Google MapReduce, Apache Hadoop, PACT, Microsoft Dryad, Google Pregel, Spark, . . .
- Scalable datastore:Amazon Dynamo, Apache Cassandra, Google Bigtable, HBase,. . .
- Distributed Consensus: Google Chubby, Apache Zookeeper, . . .
- Scalable Web applications hosting: Google App Engine, Microsoft Azure, Heroku, . . .

Clock Speed of Processors



- Herb Sutter: The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3), March 2005 (updated graph in August 2009).

Implications of the End of Free Lunch

- The clock speeds of microprocessors are not going to improve much in the foreseeable future
  - The efficiency gains in single threaded performance are going to be only moderate
- The number of transistors in a microprocessor is still growing at a high rate
  - One of the main uses of transistors has been to increase the number of computing cores the processor has
  - The number of cores in a low end workstation (as those employed in large scale datacenters) is going to keep on steadily growing
- Programming models need to change to efficiently exploit all the available concurrency - scalability to high number of cores/processors will need to be a major focus

Warehouse-scale Computing (WSC)

- The smallest unit of computation in Google scale is: Warehouse full of computers
- [WSC]: Luiz André Barroso, Urs Hölzle: *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* Morgan & Claypool Publishers 2009

  http://dx.doi.org/10.2200/S00193ED1V01Y200905CAC006
- The WSC book says:
  "…we must treat the datacenter itself as one massive warehouse-scale computer (WSC)."

Jeffrey Dean (Google): LADIS 2009 keynote failure numbers

- LADIS 2009 keynote: “Designs, Lessons and Advice from Building Large Distributed Systems” `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`
- At warehouse-scale, failures will be the norm and thus fault tolerant software will be inevitable
- Hypothetically assume that server mean time between failures (MTBF) would be 30 years (highly optimistic and not realistic number!). In a WSC with 10000 nodes roughly one node will fail per day.
- Typical yearly flakiness metrics from J. Dean (Google, 2009):
  - 1-5% of your disk drives will die
  - Servers will crash at least twice (2-4% failure rate)

Big Data

- As of May 2009, the amount of digital content in the world is estimated to be 500 Exabytes (500 million TB)
- EMC sponsored study by IDC in 2007 estimates the amount of information created in 2010 to be 988 EB
- Worldwide estimated hard disk sales in 2010: $\approx$ 675 million units
- Data comes from: Video, digital images, sensor data, biological data, Internet sites, social media, ...
- The problem of such large data massed, termed Big Data calls for new approaches to storage and processing of data

Example: Simple Word Search

- Example: Suppose you need to search for a word in a 2TB worth of text that is found only once in the text mass using a compute cluster
- Assuming 100MB/s read speed, in the worst case reading all data from a single 2TB disk takes $\approx$ 5.5 hours
- If 100 hard disks can be used in parallel, the same task takes less than four minutes
- Scaling using hard disk parallelism is one of the design goals of scalable batch processing in the cloud

Google MapReduce

- A scalable batch processing framework developed at Google for computing the Web index
- When dealing with Big Data (a substantial portion of the Internet in the case of Google!), the only viable option is to use hard disks in parallel to store and process it
- Some of the challenges for storage is coming from Web services to store and distribute pictures and videos
- We need a system that can effectively utilize hard disk parallelism and hide hard disk and other component failures from the programmer

Google MapReduce (cnt.)

- MapReduce is tailored for batch processing with hundreds to thousands of machines running in parallel, typical job runtimes are from minutes to hours
- As an added bonus we would like to get increased programmer productivity compared to each programmer developing their own tools for utilizing hard disk parallelism

Google MapReduce (cnt.)

- The MapReduce framework takes care of all issues related to parallelization, synchronization, load balancing, and fault tolerance. All these details are hidden from the programmer
- The system needs to be linearly scalable to thousands of nodes working in parallel. The only way to do this is to have a very restricted programming model where the communication between nodes happens in a carefully controlled fashion
- Apache Hadoop is an open source MapReduce implementation used by Yahoo!, Facebook, and Twitter

MapReduce and Functional Programming

- Based on the functional programming in the large:
  - User is only allowed to write side-effect free functions "**Map**" and "**Reduce**"
  - Re-execution is used for fault tolerance. If a node executing a Map or a Reduce task fails to produce a result due to hardware failure, the task will be re-executed on another node
  - Side effects in functions would make this impossible, as one could not re-create the environment in which the original task executed
  - One just needs a fault tolerant storage of task inputs
  - The functions themselves are usually written in a standard imperative programming language, usually Java

Why No Side-Effects?

- Side-effect free programs will produce the same output irregardless of the number of computing nodes used by MapReduce
- Running the code on one machine for debugging purposes produces the same results as running the same code in parallel
- It is easy to introduce side-effect to MapReduce programs as the framework does not enforce a strict programming methodology. However, the behavior of such programs is undefined by the framework, and should therefore be avoided.

Yahoo! MapReduce Tutorial

- We use a Figures from the excellent MapReduce tutorial of Yahoo! [YDN-MR], available from: `http://developer.yahoo.com/hadoop/tutorial/module4.html`
- In functional programming, two list processing concepts are used
  - Mapping a list with a function
  - Reducing a list with a function

Mapping a List

Mapping a list applies the mapping function to each list element (in parallel) and outputs the list of mapped list elements:

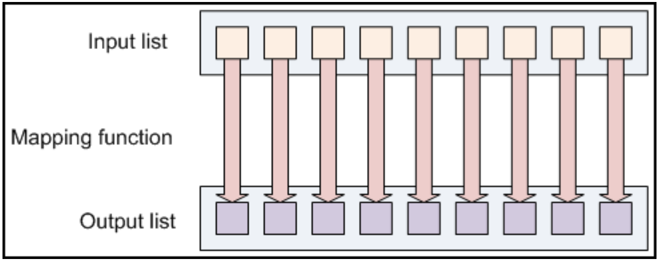


Figure: Mapping a List with a Map Function, Figure 4.1 from [YDN-MR]

Reducing a List

Reducing a list iterates over a list sequentially and produces an output created by the reduce function:

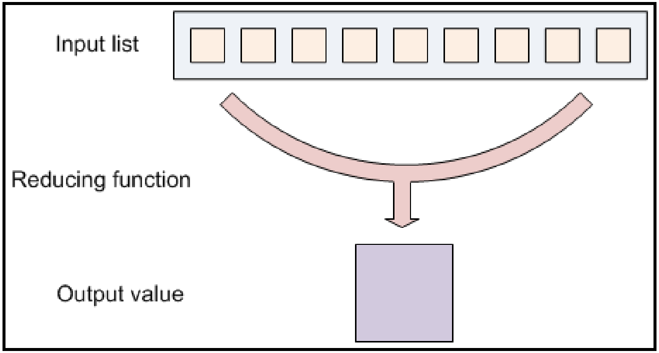


Figure: Reducing a List with a Reduce Function, Figure 4.2 from [YDN-MR]

Grouping Map Output by Key to Reducer

In MapReduce the map function outputs `(key, value)`-pairs. The MapReduce framework groups map outputs by key, and gives each reduce function instance `(key, (..., list of values, ...))` pair as input. Note: Each list of values having the same key will be independently processed:

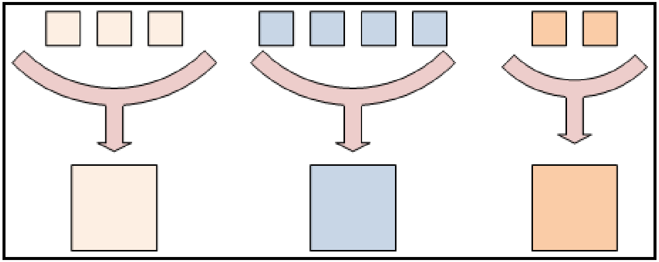


Figure: Keys Divide Map Output to Reducers, Figure 4.3 from [YDN-MR]

MapReduce Data Flow

Practical MapReduce systems split input data into large (64MB+) blocks fed to user defined map functions:

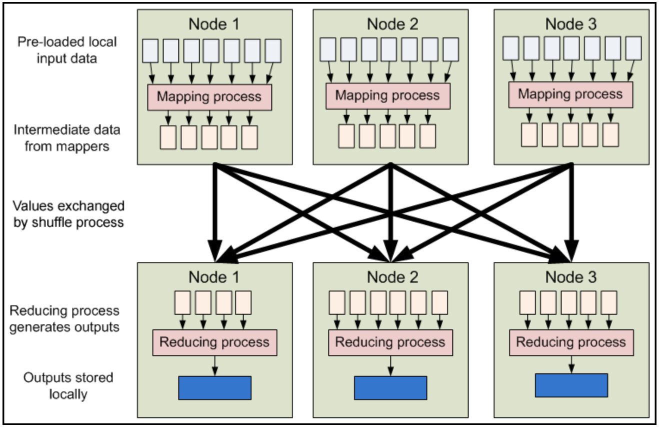


Figure: High Level MapReduce Dataflow, Figure 4.4 from [YDN-MR]

Recap: Map and Reduce Functions

- The framework only allows a user to write two functions: a “**Map**” function and a “**Reduce**” function
- The **Map**-function is fed blocks of data (block size 64-128MB), and it produces `(key, value)` -pairs
- The framework groups all values with the same key to a `(key, (..., list of values, ...))` format, and these are then fed to the **Reduce** function
- A special Master node takes care of the scheduling and fault tolerance by re-executing Mappers or Reducers

MapReduce Diagram

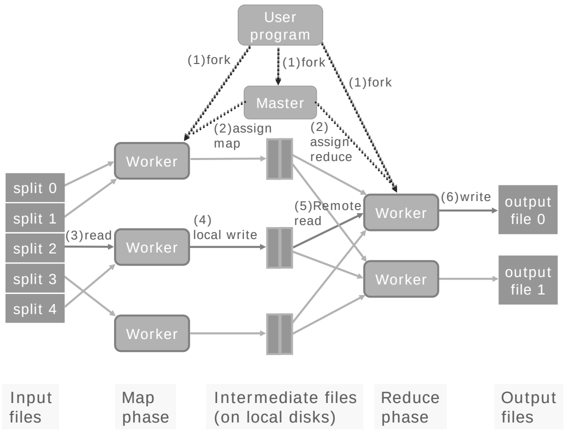


Figure: J. Dean and S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI 2004

Example: Word Count

- Classic word count example from the Hadoop MapReduce tutorial:
  `http://hadoop.apache.org/common/docs/current/mapred_tutorial.html`
- Consider doing a word count of the following file using MapReduce:

```
Hello World Bye World
Hello Hadoop Goodbye Hadoop
```

Example: Word Count (cnt.)

- Consider a Map function that reads in words one a time, and outputs `(word, 1)` for each parsed input word
- The Map function output is:

```
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
```

Example: Word Count (cnt.)

- The Shuffle phase between Map and Reduce phase creates a list of values associated with each key
- The Reduce function input is:

```
(Bye, (1))
(Goodbye, (1))
(Hadoop, (1, 1))
(Hello, (1, 1))
(World, (1, 1))
```

Example: Word Count (cnt.)

- Consider a reduce function that sums the numbers in the list for each key and outputs `(word, count)` pairs. The output of the Reduce function is the output of the MapReduce job:

```
(Bye, 1)
(Goodbye, 1)
(Hadoop, 2)
(Hello, 2)
(World, 2)
```

Phases of MapReduce

1. A Master (In Hadoop terminology: Job Tracker) is started that coordinates the execution of a MapReduce job. Note: Master is a single point of failure
2. The master creates a predefined number of *M* Map workers, and assigns each one an input split to work on. It also later starts a predefined number of *R* reduce workers
3. Input is assigned to a free Map worker 64-128MB split at a time, and each user defined Map function is fed `(key, value)` pairs as input and also produces `(key, value)` pairs

Phases of MapReduce(cnt.)

4. Periodically the Map workers flush their `(key, value)` pairs to the local hard disks, partitioning by their `key` to R partitions (default: use hashing), one per reduce worker
5. When all the input splits have been processed, a Shuffle phase starts where $M \times R$ file transfers are used to send all of the mapper outputs to the reducer handling each key partition. After reducer receives the input files, reducer sorts (and groups) the `(key, value)` pairs by the key
6. User defined Reduce functions iterate over the `(key, (..., list of values, ...))` lists, generating output `(key, value)` pairs files, one per reducer

Google MapReduce (cnt.)

- The user just supplies the Map and Reduce functions, nothing more
- The only means of communication between nodes is through the shuffle from a mapper to a reducer
- The framework can be used to implement a distributed sorting algorithm by using a custom partitioning function
- The framework does automatic parallelization and fault tolerance by using a centralized Job tracker (Master) and a distributed filesystem that stores all data redundantly on compute nodes
- Uses functional programming paradigm to guarantee correctness of parallelization and to implement fault-tolerance by re-execution

Apache Hadoop

- An Open Source implementation of the MapReduce framework, originally developed by Doug Cutting and heavily used by e.g., Yahoo! and Facebook
- "Moving Computation is Cheaper than Moving Data" - Ship code to data, not data to code.
- Map and Reduce workers are also storage nodes for the underlying distributed filesystem: Job allocation is first tried to a node having a copy of the data, and if that fails, then to a node in the same rack (to maximize network bandwidth)
- Project Web page: `http://hadoop.apache.org/`

Apache Hadoop (cnt.)

- When deciding whether MapReduce is the correct fit for an algorithm, one has to remember the fixed data-flow pattern of MapReduce. The algorithm has to be efficiently mapped to this data-flow pattern in order to efficiently use the underlying computing hardware
- Builds reliable systems out of unreliable commodity hardware by replicating most components (exceptions: Master/Job Tracker and NameNode in Hadoop Distributed File System)

Apache Hadoop (cnt.)

- Tuned for large (gigabytes of data) files
- Designed for very large 1 PB+ data sets
- Designed for streaming data accesses in batch processing, designed for high bandwidth instead of low latency
- For scalability: NOT a POSIX filesystem
- Written in Java, runs as a set of user-space daemons

Hadoop Distributed Filesystem (HDFS)

- A distributed replicated filesystem: All data is replicated by default on three different Data Nodes
- Inspired by the Google Filesystem
- Each node is usually a Linux compute node with a small number of hard disks (4-12)
- A single NameNode that maintains the file locations, many DataNodes (1000+)

Hadoop Distributed Filesystem (cnt.)

- Any piece of data is available if at least one datanode replica is up and running
- Rack optimized: by default one replica written locally, second in the same rack, and a third replica in another rack (to combat against rack failures, e.g., rack switch or rack power feed)
- Uses large block size, 128 MB is a common default - designed for batch processing
- For scalability: Write once, read many filesystem

Implications of Write Once

- All applications need to be re-engineered to only do sequential writes. Example systems working on top of HDFS:
    - HBase (Hadoop Database), a database system with only sequential writes, Google Bigtable clone
    - MapReduce batch processing system
    - Apache Pig and Hive data mining tools
    - Mahout machine learning libraries
    - Lucene and Solr full text search
    - Nutch web crawling

Two Large Hadoop Installations

- Yahoo! (2009): 4000 nodes, 16 PB raw disk, 64TB RAM, 32K cores
- Facebook (2010): 2000 nodes, 21 PB storage, 64TB RAM, 22.4K cores
  - 12 TB (compressed) data added per day, 800TB (compressed) data scanned per day
  - A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, H. Liu: *Data warehousing and analytics infrastructure at Facebook*. SIGMOD Conference 2010: 1013-1020.
    `http://doi.acm.org/10.1145/1807167.1807278`

Cloud Software Project

- ICT SHOK Program Cloud Software (2010-2013)
  - A large Finnish consortium, see: `http://www.cloudsoftwareprogram.org/`
  - Case study at Aalto: CSC Genome Browser Cloud Backend
  - Co-authors: Matti Niemenmaa, André Schumacher (Aalto University, Department of Information and Computer Science), Aleksi Kallio, Eija Korpelainen, Taavi Hupponen, and Petri Klemelä (CSC — IT Center for Science)

CSC Genome Browser

- CSC provides tools and infrastructure for bioinformatics
- Bioinformatics is the largest customer group of CSC (in user numbers)
- Next-Generation Sequencing (NGS) produces large data sets (TB+)
- Cloud computing can be harnessed for analyzing these data sets
- 1000 Genomes project (`http://www.1000genomes.org`): Freely available 50 TB data set of human genomes

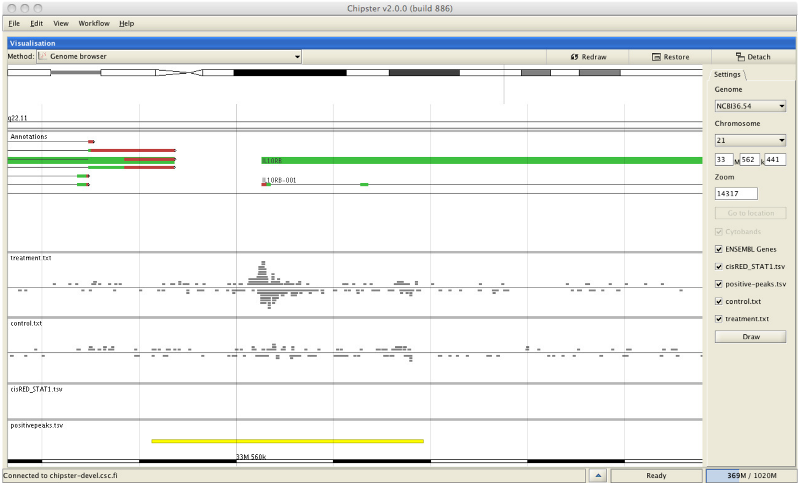
CSC Genome Browser

- Cloud computing technologies will enable scalable NGS data analysis
- There exists prior research into sequence alignment and assembly in the cloud
- Visualization is needed in order to understand large data masses
- Interactive visualization for 100GB+ datasets can only be achieved with preprocessing in the cloud

Genome Browser Requirements

- Interactive browsing with zooming in and out, “Google Earth”-style
- Single datasets 100GB-1TB+ with interactive visualization at different zoom levels
  - Preprocessing used to compute summary data for the higher zoom levels
  - Dataset too large to compute the summary data in real time using the real dataset
  - Scalable cloud data processing paradigm map-reduce implemented in Hadoop used to compute the summary data in preprocessing (currently upto 15 x 12 = 180 cores)

Genome Browser GUI



Method: Genome browser

Genome: NCBI36.54

Chromosome: 21

33 M 562 k 441

Zoom: 14317

[✓] Cytobands

[✓] ENSEMBL Genes

[✓] cisRED_STAT1.tsv

[✓] positive-peaks.tsv

[✓] control.txt

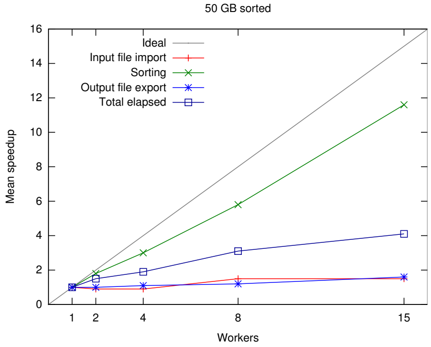
[✓] treatment.txt

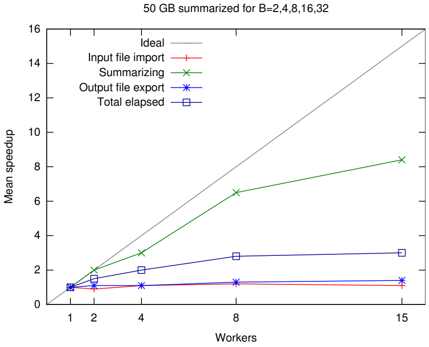
Connected to chipster-devel.csc.fi

Experiments

- One 50 GB (compressed) BAM input file from *1000 Genomes*
- Run on the Triton cluster 1-15 compute nodes with 12 cores each
- Four repetitions for increasing number of worker nodes
- Two types of runs: sorting according to starting position ("sorted") and read aggregation using five summary-block sizes at once ("summarized")

Mean speedup





Results

- An updated CSC Genome browser GUI
- Chipster: Tools and framework for producing visualizable data
- An implementation of preprocessing for visualization using Hadoop
- Scalability studies for running Hadoop on 50 GB+ datasets on the Triton cloud testbed
- Software released: `http://sourceforge.net/projects/hadoop-bam/`

Current Research Topics

- Aalto and CSC both have datacenters which can be used as testbeds for cloud computing technologies
- Focus on cloud based data analysis for “Big Data”
- MapReduce (Hadoop) scalable batch processing technologies in the cloud
- Scalable datastores: HBase (Hadoop Database) has been evaluated, also other cloud based datastores such as Cassandra are of interest