# MapReduce

Nguyen Quang Hung

# Objectives

- This slides is used to introduce students about MapReduce framework: programming model and implementation.

# Outline

- Challenges
- Motivation
- Ideas
- Programming model
- Implementation
- Related works
- References

# Introduction

- ❑ Challenges?
  - – Applications face with large-scale of data (e.g. multi-terabyte).
    - » High Energy Physics (HEP) and Astronomy.
    - » Earth climate weather forecasts.
    - » Gene databases.
    - » Index of all Internet web pages (in-house).
    - » etc
  - – Easy programming to non-CS scientists (e.g. biologists)

# MapReduce

- Motivation: Large scale data processing
  - Want to process huge of datasets (>1 TB).
  - Want to parallelize across hundreds/thousands of CPUs.
  - Want to make this easy.

# MapReduce: ideas

- Automatic parallel and data distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers

# MapReduce: programming model

- Borrows from functional programming
- Users implement interface of two functions: map and reduce:
  - map (k1,v1) → list(k2,v2)
  - reduce (k2,list(v2)) → list(v2)

# map() function

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

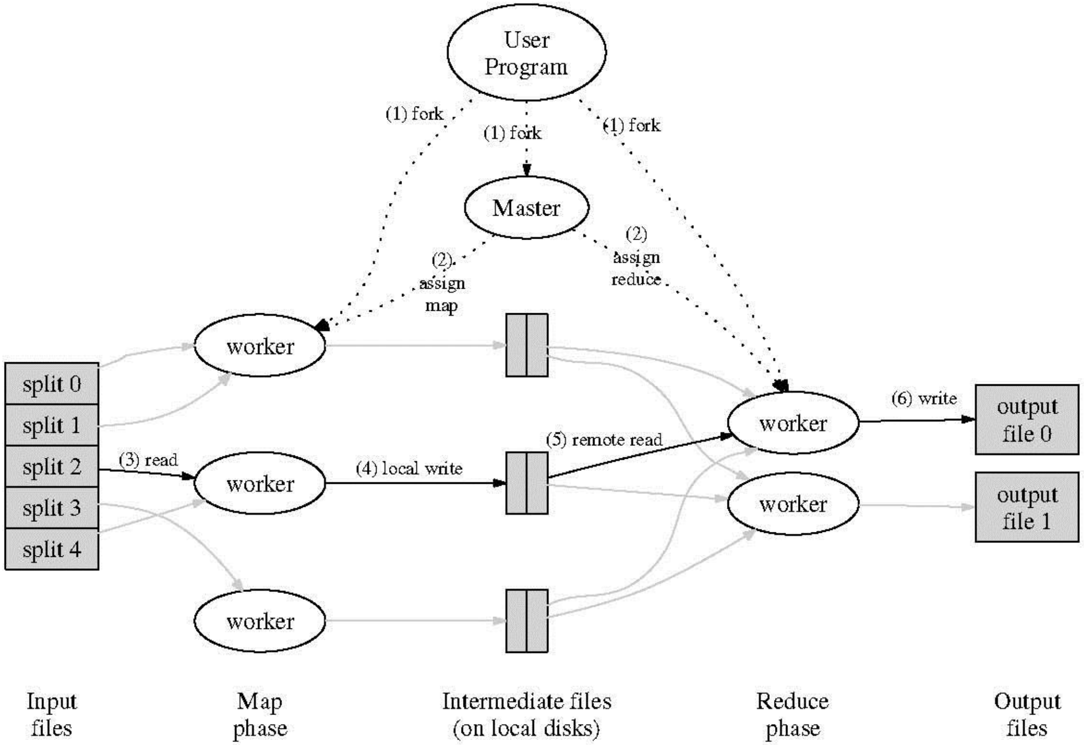
# reduce() function

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

# Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can’t start until map phase is completely finished.

# MapReduce: execution flows



# Example: word counting

```
map(String input_key, String input_doc):
    // input_key: document name
    // input_doc: document contents
    for each word w in input_doc:
      EmitIntermediate(w, "1"); // intermediate values
```

```
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);
     Emit(AsString(result));
```

- More examples: Distributed Grep, Count of URL access frequency, etc.

# Locality

- Master program allocates tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack (cluster rack)
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

# Fault tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.

# Optimizations (1)

- ❑ No reduce can start until map is complete:
  - – A single slow disk controller can rate-limit the whole process
- ❑ Master redundantly executes “slow-moving” map tasks; uses results of first copy to finish

*Why is it safe to redundantly execute map tasks? Wouldn’t this mess up the total computation?*

# Optimizations (2)

- “Combiner” functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

*Under what conditions is it sound to use a combiner?*

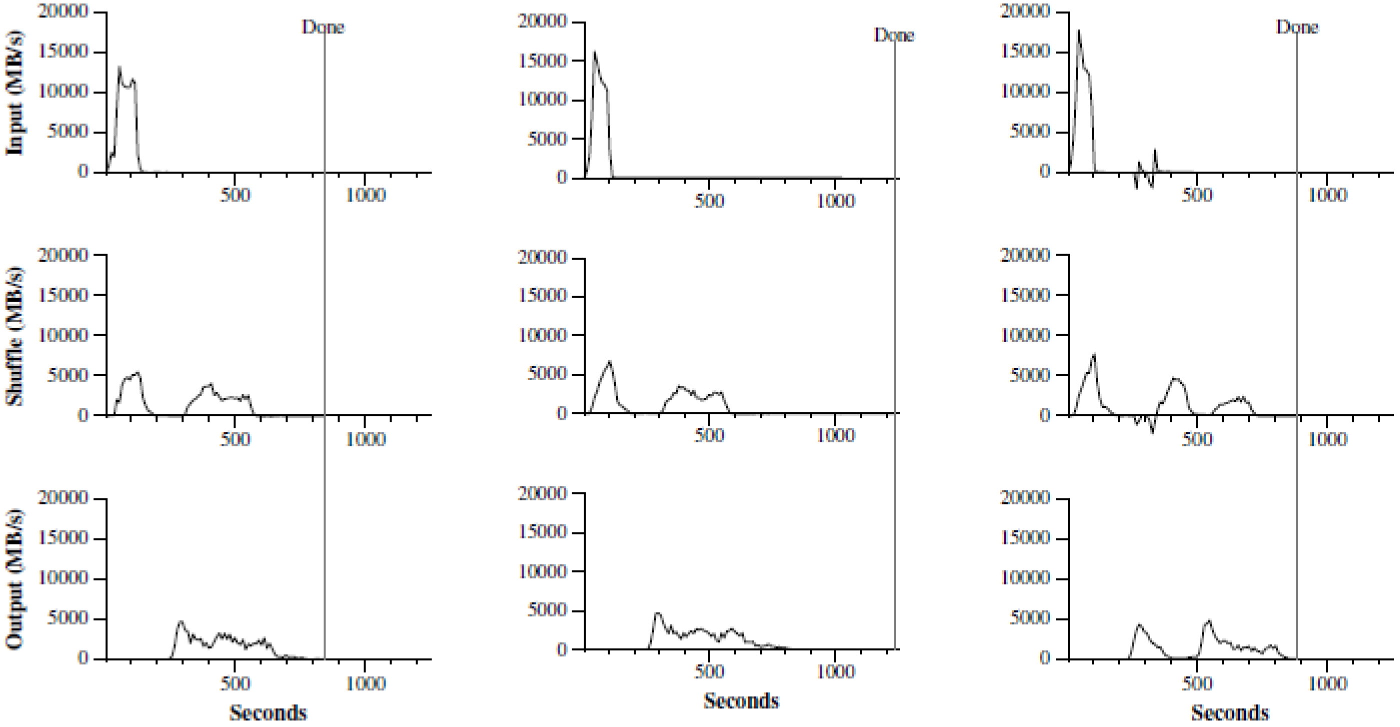
# MapReduce: implementations

- Google MapReduce: C/C++
- Hadoop: Java
- Phoenix: C/C++ multithread
- Etc.

# Google MapReduce evaluation (1)

- Cluster: approximately 1800 machines.
- Each machine: 2x2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks and a gigabit Ethernet link.
- Network of cluster:
  - Two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root.
  - Round-trip time any pair of machines: < 1 msec.

# Google MapReduce evaluation (2)



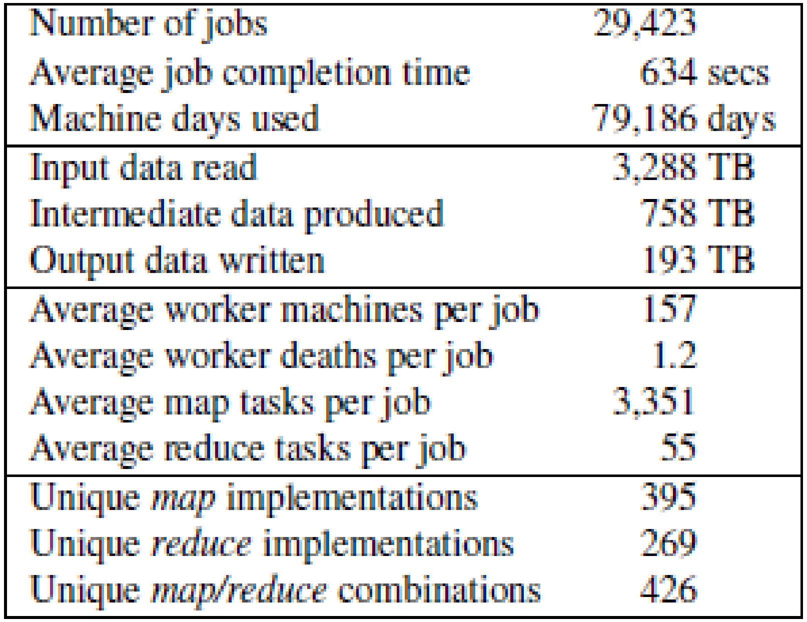
(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

*Data transfer rates over time for different executions of the sort program (*J.Dean and S.Ghemawat shows in *their paper [1, page 9])*

# Google MapReduce evaluation (3)



| | |
|---|---|
| Number of jobs | 29,423 |
| Average job completion time | 634 secs |
| Machine days used | 79,186 days |
| Input data read | 3,288 TB |
| Intermediate data produced | 758 TB |
| Output data written | 193 TB |
| Average worker machines per job | 157 |
| Average worker deaths per job | 1.2 |
| Average map tasks per job | 3,351 |
| Average reduce tasks per job | 55 |
| Unique *map* implementations | 395 |
| Unique *reduce* implementations | 269 |
| Unique *map/reduce* combinations | 426 |

Table 1: MapReduce jobs run in August 2004

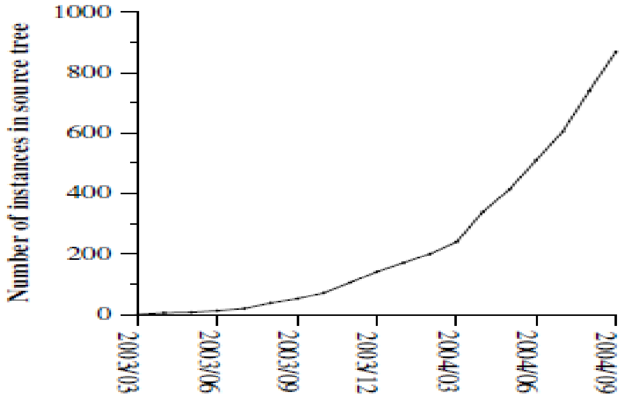
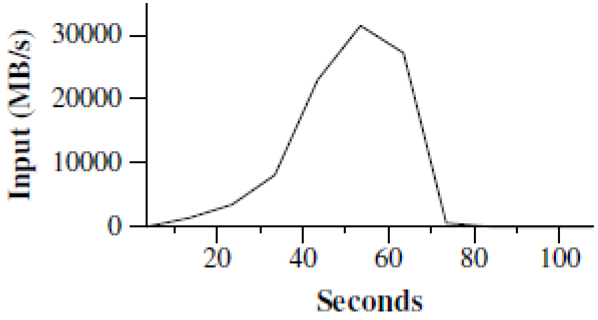


Figure 4: MapReduce instances over time

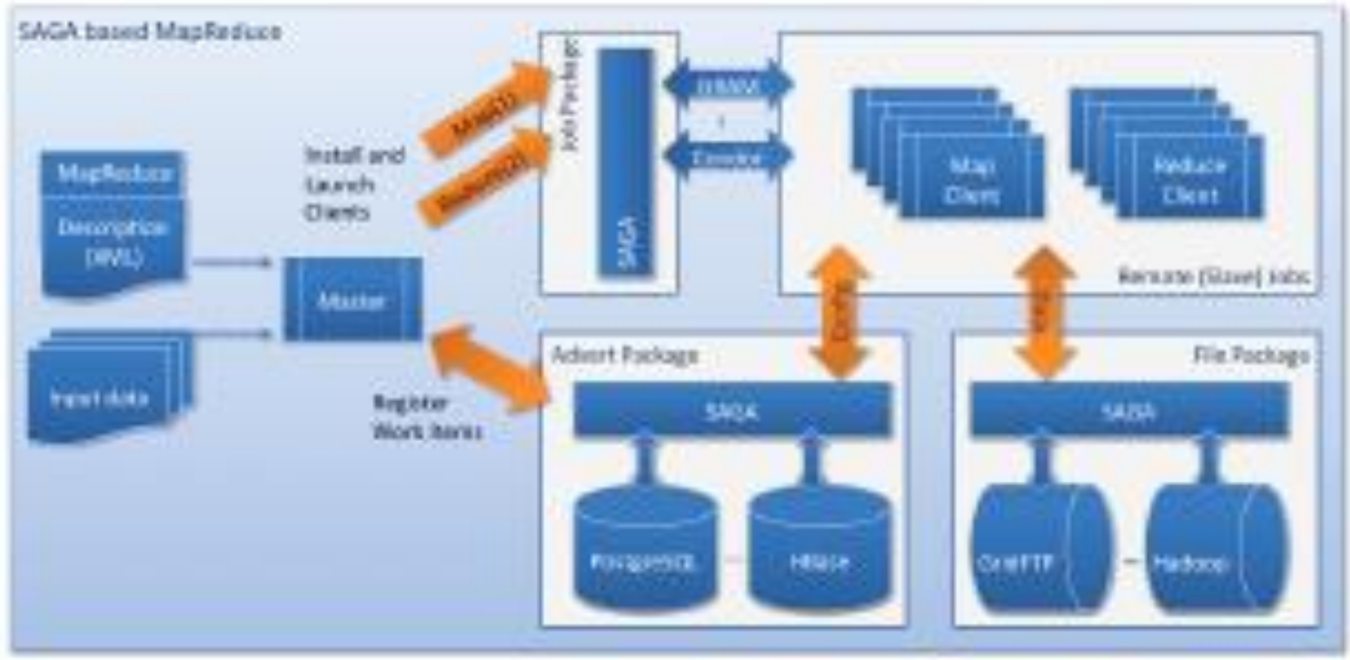


J.Dean and S.Ghemawat shows in theirs paper *[1]*

# Related works

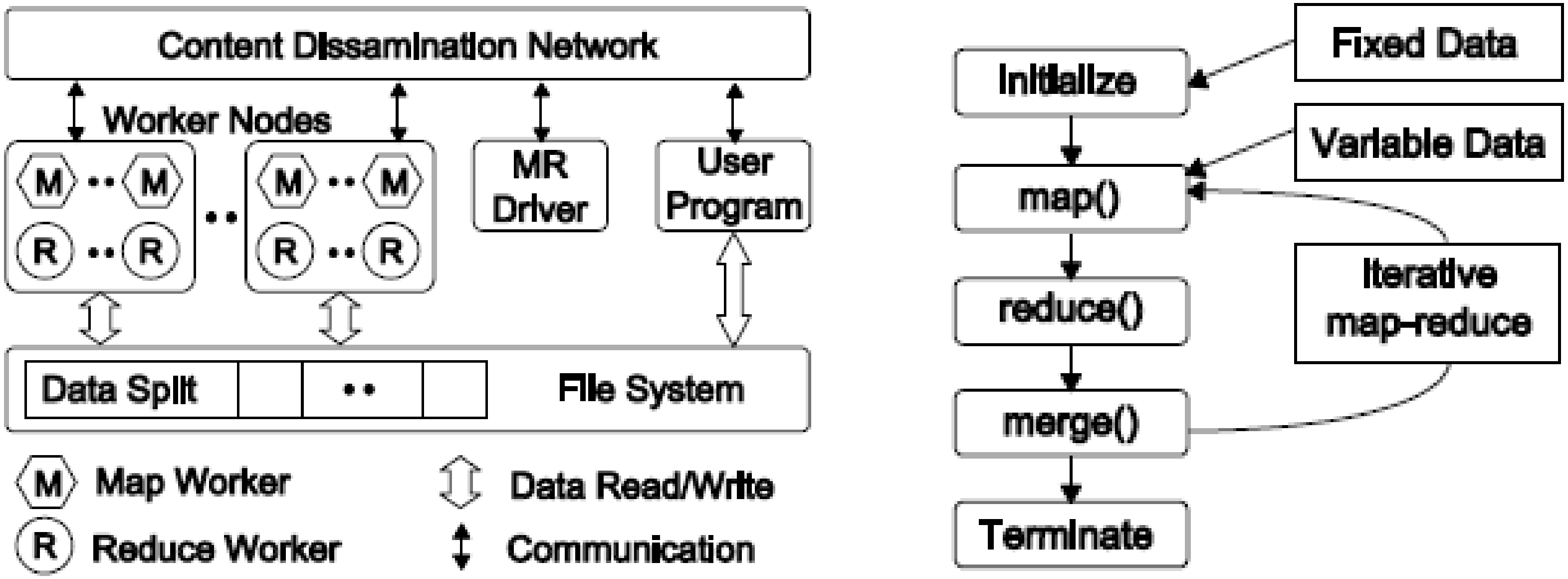
- Bulk Synchronous Programming [6]
- MPI primitives [4]
- Condor [5]
- SAGA-MapReduce [8]
- CGI-MapReduce [7]

# SAGA-MapReduce



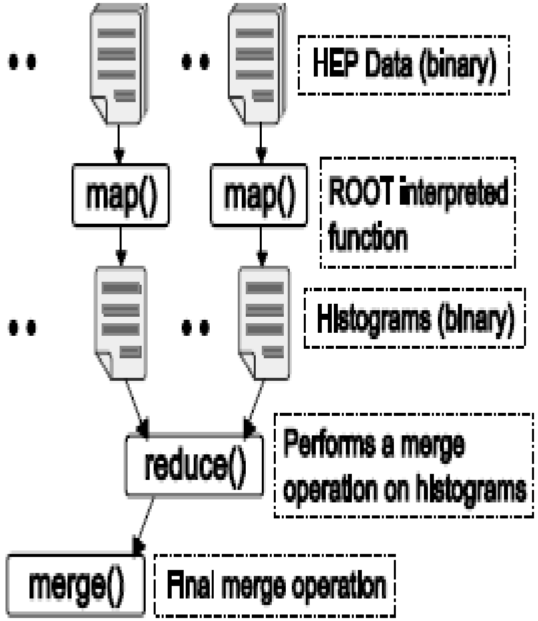
High-level control flow diagram for SAGA-MapReduce. SAGA uses a master-worker paradigm to implement the MapReduce pattern. The diagram shows that there are several different infrastructure options to a SAGA based application [8]

# CGL-MapReduce

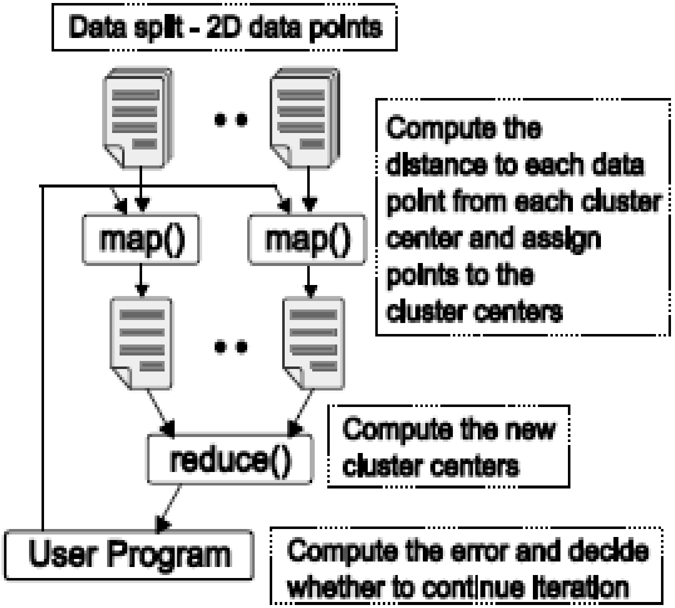


Components of the CGL-MapReduce , extracted from [8]

# CGL-MapReduce: sample applications

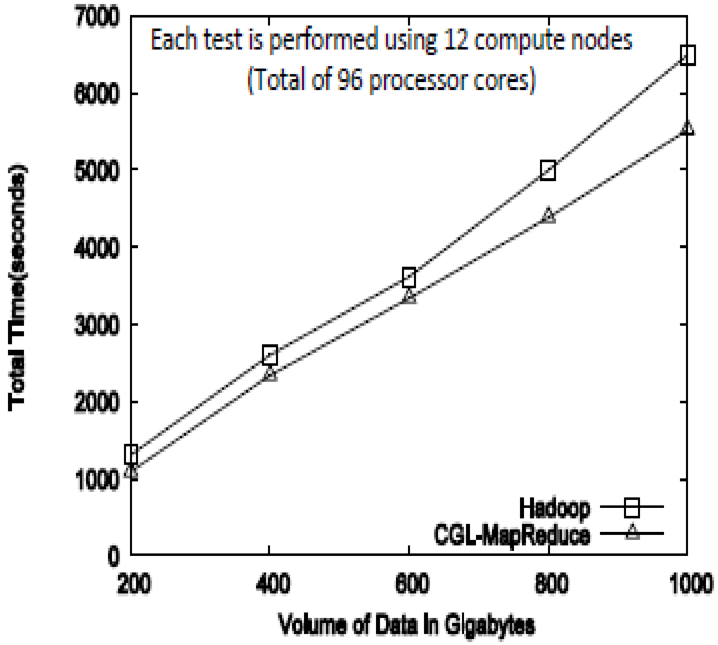


MapReduce for HEP

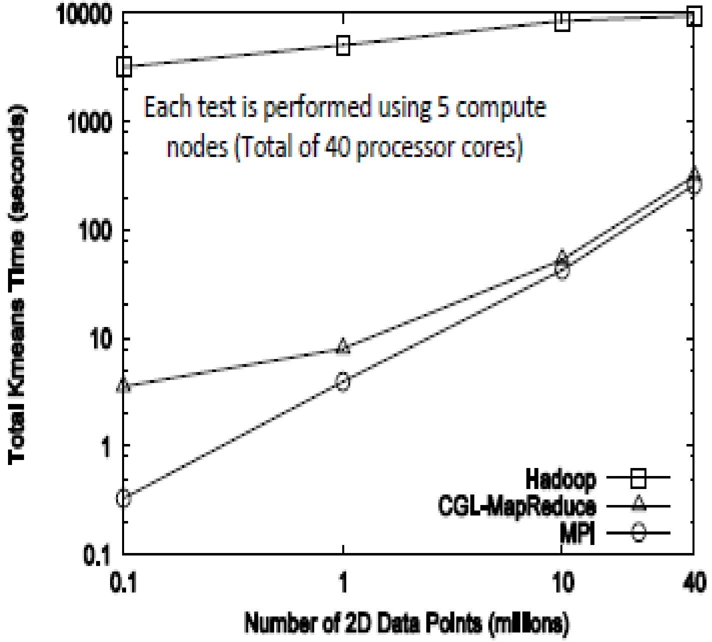

Data split - 2D data points
Compute the distance to each data point from each cluster center and assign points to the cluster centers
map()
map()
reduce()
Compute the new cluster centers
User Program
Compute the error and decide whether to continue iteration

MapReduce for Kmeans

# CGL-MapReduce: evaluation



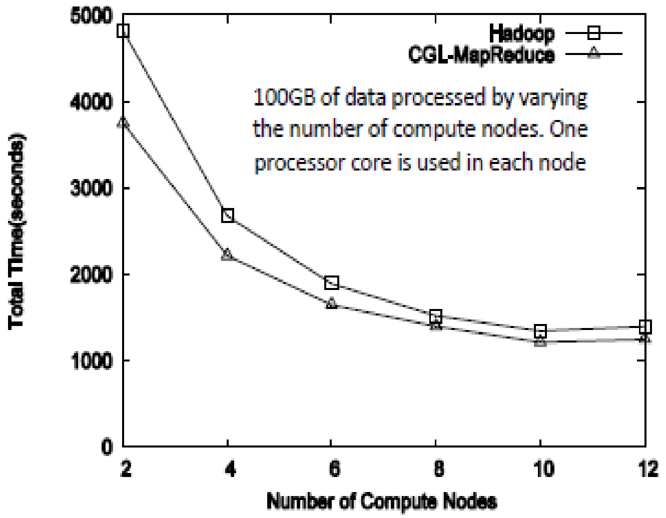
HEP data analysis, execution time vs. the volume of data (fixed compute resources)



Total Kmeans time against the number of data points (Both axes are in log scale)

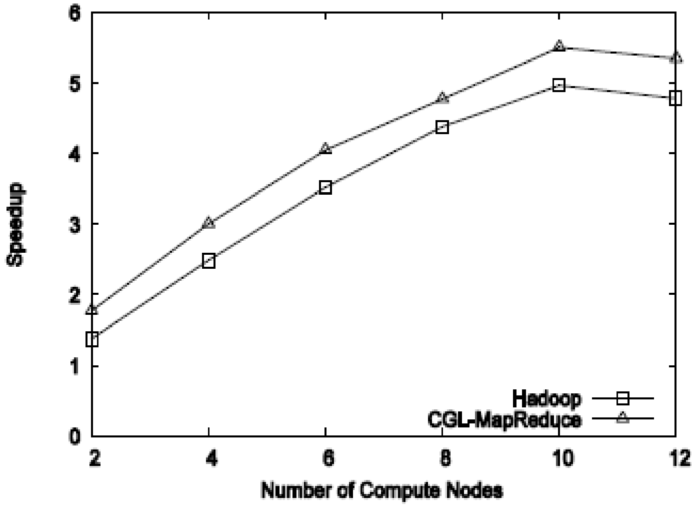
J.Ekanayake, S.Pallickara, and G.Fox show in their paper [7]

# Hadoop vs. CGL-MapReduce



100GB of data processed by varying the number of compute nodes. One processor core is used in each node

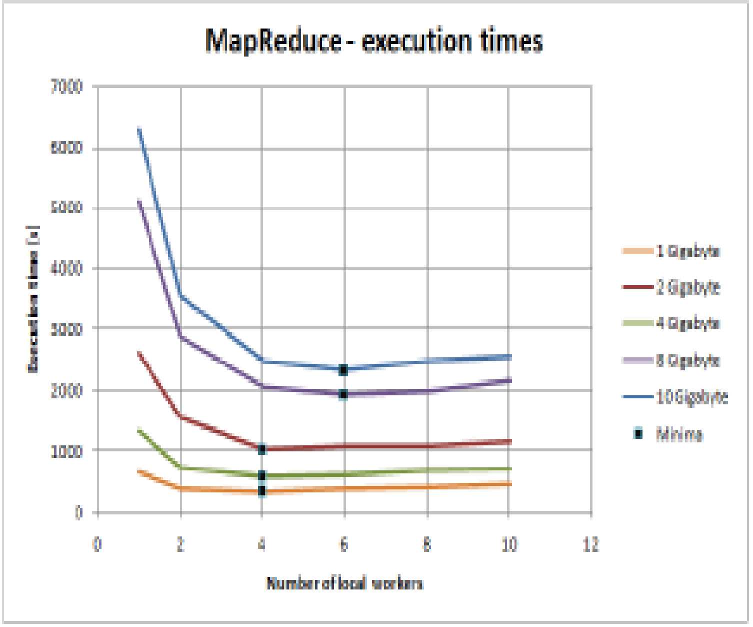
Total time vs. the number of compute nodes (fixed data)

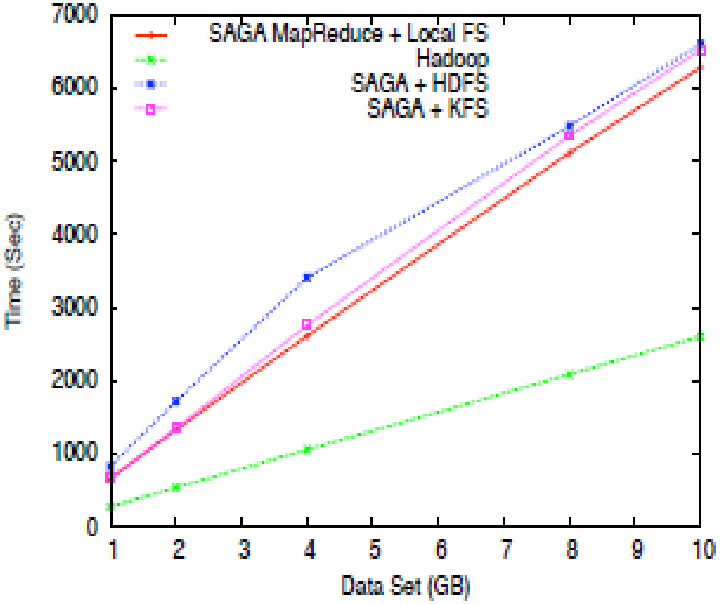


Speedup for 100GB of HEP data

J.Ekanayake, S.Pallickara, and G.Fox show in their paper [7]

# Hadoop vs. SAGA-MapReduce





C.Miceli, M.Miceli, S. Jha, H. Kaiser, A. Merzky show in [8]

# Exercise

- Write again “word counting” program by using Hadoop framework.
  - Input: text files
  - Result: show number of words in these inputs files

# Conclusions

- MapReduce has proven to be a useful abstraction
- Simplifies large-scale computations on cluster of commodity PCs
- Functional programming paradigm can be applied to large-scale applications
- Focus on problem, let library deal w/ messy details

# References

1. Jeffrey Dean and Sanjay Ghemawat, **MapReduce: Simplied Data Processing on Large Clusters, 2004**
2. Christophe Bisciglia, Aaron Kimball, & Sierra Michels-Slettvet, **Distributed Computing Seminar, Lecture 2: MapReduce Theory and Implementation**, Summer 2007, © Copyright 2007 University of Washington and licensed under the Creative Commons Attribution 2.5 License.
3. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. **The Google file system**. In *19th Symposium on Operating Systems Principles, pages 29.43, Lake George,* New York, 2003.
4. William Gropp, Ewing Lusk, and Anthony Skjellum. ***Using MPI: Portable Parallel Programming with the Message-Passing Interface****. MIT Press, Cambridge, MA,* 1999.
5. Douglas Thain, Todd Tannenbaum, and Miron Livny. **Distributed computing in practice: The Condor experience**. *Concurrency and Computation: Practice and Experience,* 2004.
6. L. G. Valiant. **A bridging model for parallel computation**. *Communications of the ACM, 33(8):103.111, 1997.*
7. Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox, **MapReduce for Data Intensive Scientific Analyses,**
8. Chris Miceli12, Michael Miceli12, Shantenu Jha123, Hartmut Kaiser1, Andre Merzky, **Programming Abstractions for Data Intensive Computing on Clouds and Grids**.

# Q/A