CPET 581 Cloud Computing: Technologies and Enterprise IT Strategies

Lecture 8

Cloud Programming & Software Environments: High Performance Computing & AWS Services

Part 2 of 2

Spring 2015

A Specialty Course for Purdue University's M.S. in Technology Graduate Program: IT/Advanced Computer App Track

Paul I-Hai Lin, Professor

Dept. of Computer, Electrical and Information Technology

Purdue University Fort Wayne Campus

References

1. Chapter 6. Cloud Programming and Software Environments, Book "Distributed and Cloud Computing," by Kai Hwang, Geoffrey C. Fox a,d Jack J. Dongarra, published by Mogan Kaufmman/ Elsevier Inc.

Topics

- High Performance Computation
  - Parallel Matrix Multiplication
  - Computational Complexity and Analysis
- Parallel Programming on Amazon Web Service (AWS)
  - Amazon Platforms and Service Offerings
  - AWS Elastic Compute Cloud (EC2)
  - AWS Simple Storage Services (S3)
  - AWS Elastics Bock Store (EBS)
  - AWS Simple DS

3

Parallel Matrix Multiplication

- **Matrix,** https://en.wikipedia.org/wiki/Matrix_(mathematics)

Given two n x n matrices : **A = (a_{ij})** and **B = (b_{ij}).**

Compute the product of **A** and **B** : **C = (c_{ij}) = A x B**

where $c_{ij} = \sum a_{ik} \times b_{kj}$ for all **k=1,2, ..., n**

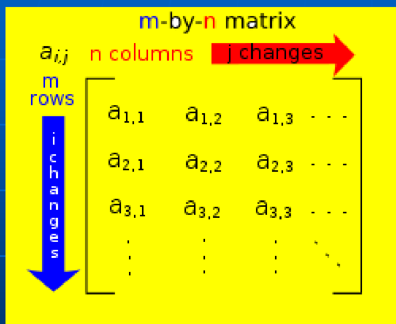
$= a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \ldots\ldots + a_{in} \times b_{nj}$

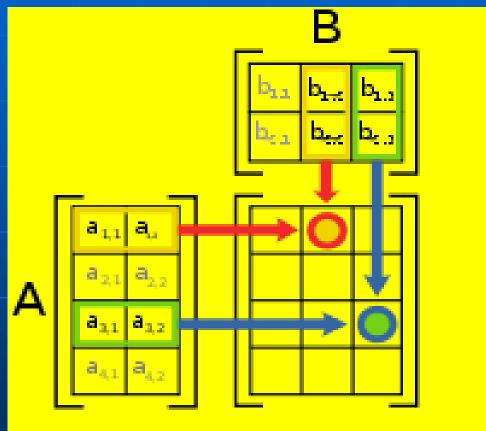
= Dot product of row vector $\mathbf{A}_i$ and column vector $\mathbf{B}_j$

= Dot product of row vector of $\mathbf{A}_i$ and row vector of $\mathbf{B}_j^T$

Parallel Matrix Multiplication

- Matrix, https://en.wikipedia.org/wiki/Matrix_(mathematics)





5

Parallel Matrix Multiplication

$$
\begin{bmatrix} c_{11} & c_{12} & \cdots & \cdots & c_{1n} \\ c_{21} & c_{22} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ c_{n1} & \cdots & \cdots & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & \cdots & b_{1n} \\ b_{21} & b_{22} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ b_{n1} & \cdots & \cdots & \cdots & b_{nn} \end{bmatrix}
$$

6

Computational Complexity Analysis

We need to perform n^2 dot products to produce all c_{ij}

The total complexity = $n^2 \times n = n^3$ "Multiply and Add " operations.

Thus, sequential execution time = $O(n^3)$.

In theory, all n^2 dot products can be done on n^2 processors in parallel (An embarrassingly parallel computation problem).

In reality, **n** is very large and n^2 is even greater,

It is impossible to exploit the full parallelism.

With **N** processors, where $N << n$, we can do it in $O(n^2 /N)$ time

Thus, the **Speedup** = $O(n^2) / O(n^2 /N) \sim O(N)$ is possible.

7

When n is very Large – Computational Cost

- Reading and storing large number of input and output matrix elements demand excessive I/O time and memory space
- Data reference locality demands many duplications of the row and column vectors to local processors
  - The Map functions in MapReduce model.
- Dot products can be done on the Reduce Nodes in parallel blocks identified by "keys"
- Demand large-scale shuffle and exchange sorting and grouping operations over all intermediate <key, value> pairs, even externally in and out of disks.
- The task fork out from the master server to all available Map and Reduce servers (workers) may result in scheduling overhead.

8

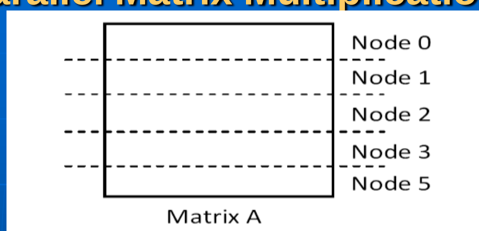
Ideas of Parallel Matrix Multiplication

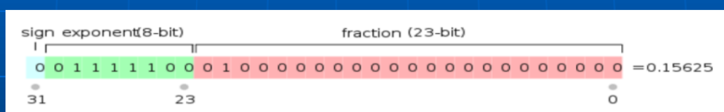
- Each time unit counts the time to carry out the dot product of two n-element vectors. (repeated multiply-and-add operations over a row vector of A and a column vector of B).
- In the sequential execution, it take n^2 time units to generate the n^2 output elements in the product matrix C. Here, the example matrix has an order n = 1,024.
- If you partition the matrix into 16 equal blocks (64 x 64 each). Then, only 256n output elements are generated in each block. Thus 16 blocks can be handled by 16 VM instances in parallel.
- In theory, the total execution time should be shortened to 1/16 of the total sequential execution time, if all communication and memory-access overheads are ignored.

9

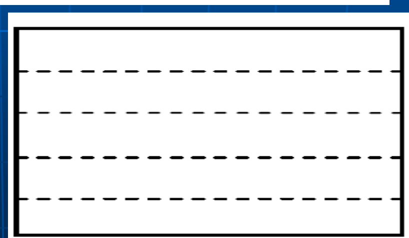
Ideas of Parallel Matrix Multiplication

Input Matrix partitioning
by row vectors of matrix A and by column vectors of matrix B or by row vector of the transposed matrix B^T





Dot Product Parallelization into Blocks affect the Reduce speed and efficiency in the computation section of the entire MapReduce process.



Matrix C

10

Parallel Matrix Multiplication (cont.)

- **Similarly, if you use 64 VM instances, you should expect a 1/64 execution time. Use up to the maximum number of 128 machine instances, if it is allowed in your assigned Amazon account.**
- **In the extreme case of using n^2 instances (1 M or 2^{20} instances), you may end up with only one time unit to complete the total execution. That is not allowed in the AWS platform, realistically speaking.**

11

Hadoop and Amazon Elastic MapReduce

- A software platform originally developed by Yahoo to enable user write and run applications over vast distributed data.
- Attractive Features in Hadoop:
  - Scalable
  - Economical: an open-source MapReduce
  - Efficient
  - Reliable

12

AWS Usage Growth

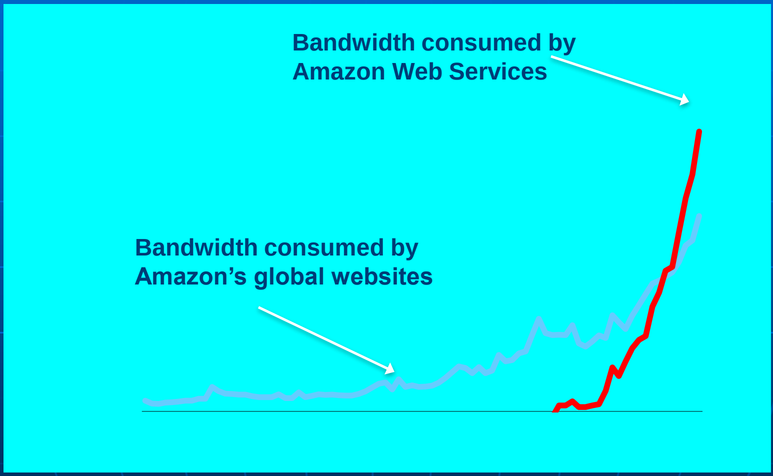
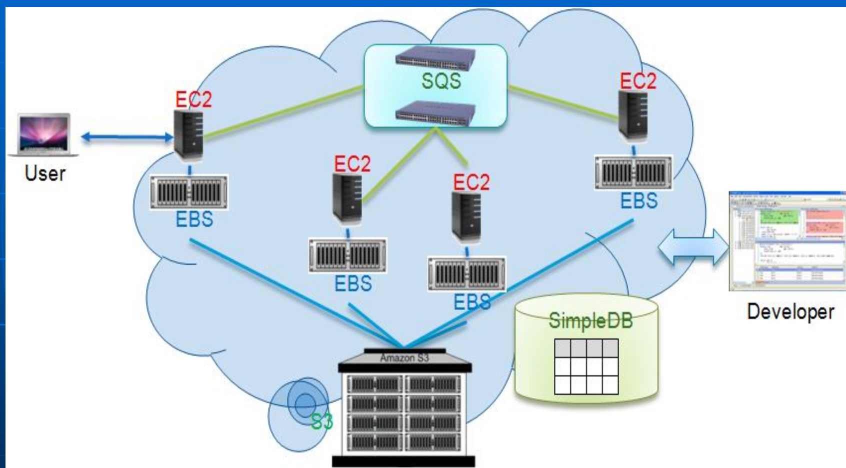


Table 4.6 Amazon Web Service (AWS) Offerings in 2011

Service Area	Service Modules and Abbreviated Names
Compute	Elastic Compute Clouyd (EC2), Elastic MapReduce, Auto Scaling
Messaging	Simple Queue Service (SQS), Simple Notification Service (SNS)
Storage	Simple Storage Service (S3), Elastic Block Storage (EBS), AWS Import/Export
Content Delivery	Amazon CloudFront
Monitory	Amazon CloudWatch
Support	AWS Premium Support
Datbase	Amazon SimpleDB, Realational Databasae Service (RDS)
Networking	Virtual Private Cloud (VPC) (Example 4.6), Elastic Load Balancing
Web Traffic	Alexa Web Information Service, Alexa Web Sites
E-Commerce	Fulfillment Web Service (FWS)
Payments and Billing	Flexible Payments Service (FPS), Amazon DevPay
Workforce	Amazon Mechanical Turk

The AWS Platform



15

Major Service Modules for IaaS on the AWS Platform

Amazon Application

Services

Simple DB	EC2	SQS	S3

Controller

Launch Controller	Monitor Controller	Shutdown Controller	Billing Controller

Amazon Physical Infrastructure

16

Amazon Web Services (AWS)

The AWS Infrastructure Platform

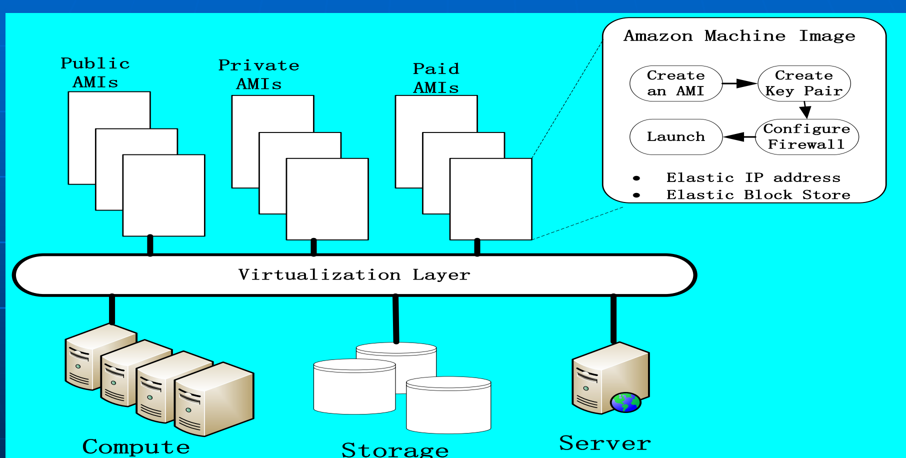
Monitoring Amazon CloudWatch	Management AWS Management Console	Tools AWS Toolkit for Eclipse	Isolated Networks Amazon Virtual Private Cloud

Parallel Processing Amazon Elastic MapReduce	Content Delivery Amazon CloudFront	Messaging Amazon Simple Queue Service (SQS)	Payments Amazon Flexible Payments Service (FPS)	On-Demand Workforce Amazon Mechanical Turk

Compute Amazon Elastic Compute Cloud (EC2) -Elastic Load Balancing -Auto Scaling	Storage Amazon Simple Storage Service (S3) -AWS Import/Export	Database Amazon RDS Amazon SimpleDB

(Courtesy of AWS, 2012)

Amazon EC2 Execution Environment



Amazon Machine Images (AMI)

Image Type	Definition
Private	Images created by you, which are private by default. You can grant access to other users to launch your private images.
Public	Images created by users and released to the Amazon Web Services community, so anyone can launch instances based on them and use them any way they like. The Amazon Web Services Developer Connection Web site lists all public images.
Paid	You can create images providing specific functions that can be launched by anyone willing to pay you per each hour of usage on top of Amazon charges.

- **AMI is a packaged server environment in EC2, based on Linux running any user software or application. AMIs are the templates for VM instances.**
- **Elastic IP address is specially reserved for EC2. Elastic Block Store offers persistent storage for EC2 instances.**

AWS Virtual Private Cloud (VPC)

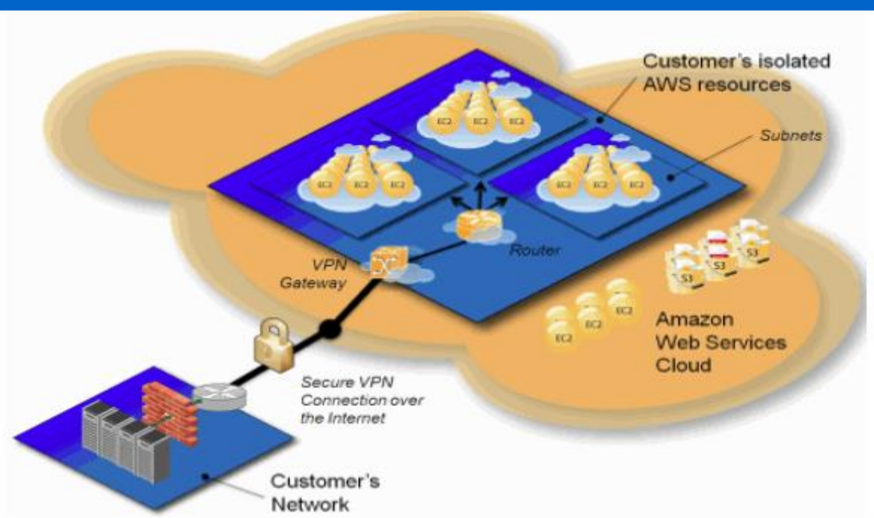
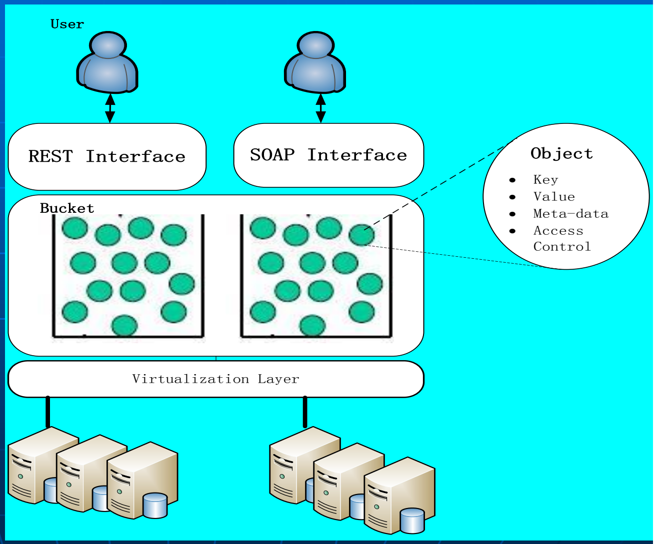


Figure 7.28 Amazon Virtual Private Cloud VPC (http://aws.amazon.com/vpc/)

Amazon S3 for Storage Provisioning

- Object is the basic unit of data
- Bucket for storing objects
- Key for data object retrieval
- Object is attributes to values, metadata, and access control



Hadoop and Amazon Elastic MapReduce

The Hadoop project is an open-source collection of projects all aimed at bringing distributed scalable data processing to the masses. Hadoop is a distributed computing platform written in Java. It incorporate features similar to those of the Google File System and of MapReduce to process vast amounts of data

Amazon Elastic MapReduce is a web service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process vast amounts of data. It utilizes a hosted Hadoop framework running on the web-scale infrastructure of Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3).

The MapReduce library in the user program first splits the input files into M pieces and then starts up many copies of the program on a cluster of machines. One of the copies of the program is the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign.

The master picks idle workers and assigns each one a map task or a reduce task. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out 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.

Python Code Solution by Risheng Wang, USC, 2011

Input Files for left Matrix A and right Matrix B

The original input files are two 1024 by 1024 matrix. Each file contains 1024 numbers and there are 1024 lines in total. However, in order to do the MapReduce efficiently, I preprocess these input files in following way:

1. The B matrix (i.e. right matrix) is transposed. In other words, each line in the file contains a column of matrix B.
2. Two more fields are inserted below each line for both matrix A and B.
    a. The first filed (L/R field) is used to distinguish lines from matrix A and those from matrix B. Its value is either "L" (Left Matrix A) or "R" (Right Matrix B).
    b. The second filed is line number (i.e. row/column number of matrix A/B)

(Courtesy of R. Wang, USC, 2011)

Input Files for left Matrix A and right Matrix B

Matrix A (AInput.txt)
L 0 4B37BC83 51EFDE9E 36AE5EE7 26687FD5 3335F2CC 5613B65E ... L 1 4291E86E 36035049 29400BFB 50E7A29A 3DCC6DC2 4311BA3D ... ... L 1023 2BA21DF8 33B5D026 2AB93D52 527ACB15 5A34AE24 ...
Matrix B (BInput.txt)
R 0 43309A27 4FB74074 4C926D41 3399E730 3F6D7ABD 4EAB174B ... R 1 495B3C1B 4596BDD8 53147CC6 2AB604AA 4BB006F5 28FBF6EC ... ... R 1023 4DE251DF 3C629BE8 434846E7 30D36D2A 25E578F0 2A888940 ...

The Output File for Matrix C

The final output matrix (Matrix C) is divided into blocks. Assume that the block size is BLOCKSIZE (=1024, 512, 256, 128 ...). The number of blocks in each row/column is 1024/BLOCKSIZE (=1, 4, 16, 64 ...). The map function is used to duplicate the input lines (rows and columns) for 1024/BLOCKSIZE times so that each block can have its required rows and columns. For example, if the number of blocks in each row is 4, each line in matrix A should be duplicated 4 times. If number of blocks in each column is 4, each line in transposed matrix B should also be duplicated 4 times. In my experiment, the number of blocks in each row and column are always same.

The Output File for Matrix C

Mapper

The map function reads the inputs lines of two matrices and dispatch/duplicate them for corresponding blocks. The intermediate key/value pair is like this:

Key	Value
{block number}	{L/R}:{Line Number}:{values of current line}

Block number is the key

The block number can be calculated as ib*NB+jb, where ib = row index of the block, jb = column index of the block, NB = the total number of blocks in each row.

The python code of map function is shown below

MatMulMapper.py

```
#!/usr/bin/env python

# Author: Risheng Wang (ruishenw@usc.edu)
# Date: 3/11/2011
# Note: This scipt is the mapper for Matrix Multiplication with Hadoop MapReduce.

import sys

# BLOCKSIZE must be the intergral power of two
BLOCKSIZE = 128
TOTALSIZE = 1024

# number of blocks for Matrix A/B
NB = TOTALSIZE/BLOCKSIZE

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input
    A_B, lineno, row_value = line.split(' ',2)
```

NB = No. of blocks in each row
(or in each transposed column)

```
if A_B == "L" :
    ib = (int)(lineno)/BLOCKSIZE;
    for jb in range(NB):
    # the key is the BLOCK Number.
        intermediate_key = '%05d'%(ib*NB+jb)
    # the value is the {L/R}:{LineNo}:{values of current line}
        intermediate_value = "L:%s:%s"%(lineno, row_value)
    # key and value are seperated by a tab
        print "%s\t%s"%(intermediate_key, intermediate_value)
if A_B == "R" :
    jb = (int)(lineno)/BLOCKSIZE;
    for ib in range(NB):
```

ib = row index of each block

jb = column index of the block

NB = No. of blocks in each row

ib * NB + jb = Block number

```
intermediate_key = "%05d"%(ib*NB+jb)
intermediate_value = "R:%s:%s"%(lineno, row_value)
print "%s\t%s"%(intermediate_key, intermediate_value)
```


Reducer

The intermediate key/value pairs will be sorted by key. And the lines for the same block will go to the same reducer. After the reducer collects all the lines (both rows and columns) for a block, it will perform matrix multiplication. The code of reducer is shown below

MatMulReducer.py

```
#!/usr/bin/env python

# Author: Risheng Wang (ruishenw@usc.edu)
# Date: 3/11/2011
# Note: This scipt is the reducer for Matrix Multiplication with Hadoop MapReduce.

import sys

import binascii
import struct

BLOCKSIZE = 128
TOTALSIZE = 1024
NB = TOTALSIZE/BLOCKSIZE

LeftMatrixBlock = []
RightTransposeMatrixBlock = []

# total number of lines (within a block),
nl = 0

# oldblockno = -1
blockno = -1
```

```
for line in sys.stdin:
    # for debug
    # nl = nl + 1

    nl = nl + 1
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input
    input_key, input_value = line.split('\t', 1)

    # for debug
    # print input_key

    blockno = int(input_key)

    A_B, index, row_value = input_value.split(':')

    if A_B == "L" :
        LeftMatrixBlock.append(row_value.split(' '))
    if A_B == "R" :
        RightTransposeMatrixBlock.append(row_value.split(' '))
```



```
# an block is finished
  if (nl == 2*BLOCKSIZE):
# reset nl
     nl = 0
# print block number to mark the output
     print blockno, BLOCKSIZE

# output & multiply and sum
res = [[0 for col in range(BLOCKSIZE)] for row in range(BLOCKSIZE)]
for i in range (BLOCKSIZE) :
   for j in range (BLOCKSIZE) :
      for k in range (TOTALSIZE) :
         left_val = struct.unpack("!f",binascii.a2b_hex(LeftMatrixBlock[i][k]))[0]
         right_val = struct.unpack("!f",binascii.a2b_hex(RightTransposeMatrixBlock[j][k]))[0]
         res[i][j] += left_val * right_val
      print res[i][j],
   # sys.stderr.write('reporter:counter:matmul,totalnum,%d\n'%(BLOCKSIZE))
   print
del LeftMatrixBlock[:]
del RightTransposeMatrixBlock[:]
```

Multiply-and-Add (dot

Output

The output of reducer is formatted like this:

Block number
The final results of this block (i.e. a BLOCKSIZE by BLOCKSIZE matrix)

An example is shown below

A submatrix (128x128) for each of 64 = 8x8 blocks, if the block size is 128 elements

Part-00000
0 2.4373216327e+32 3.88248143835e+31 5.19607198289e+32 7.53854952612e+30 ... . 17 9.21375889096e+31 2.54720909701e+30 2.44615706762e+32 3.8188708317e+32 ... ...

Note that one output file may contain the results of multiple blocks. The number of output files is depended on the number of reduce tasks (which is equal to the number of instances in my experiment) in the system. The output files are named like part-00000, part-00001 ... and so on.

Performance Results

The figure below shows the execution time (blue line with primary y axis) and efficiency (red line with secondary y axis) of matrix multiplication implementation with different number of instances (up to 20). The Python is a script language, and its performance is much lower than C/Java (more than 100 times slower [8]). To run a 1024 by 1024 matrix multiplication in a single instance (with one partition) needs more than two hours (9225 seconds). With the number of instances increase, the execution time is reduced rapidly. With four instances, it finishes in only half of time (4647 seconds) compared to single instance case. When number of instance reaches 20, the execution time is only about 15 minutes (938 seconds).

The efficiency (with n instance) can be calculated as this:

Speedup = execution time with one instance / execution time with n instances

Efficiency = Speedup / n

The red line (with secondary y axis) shows the efficiency of MapReduce matrix multiplication with different number of instances. As we can see from the figure, the efficiency is below one when the number of instance is larger than one. This is because not all the operations in MapReduce Matrix Multiplication can be paralleled. The serial operations in the MapReduce job flows includes all the operations done by master, like assign workload to mappers and reducers. Sorting intermediate key/value pairs are also part of serial operations. With the number of instance increases, the efficiency decreases. This is because serial operation takes larger and larger portion of execution time.

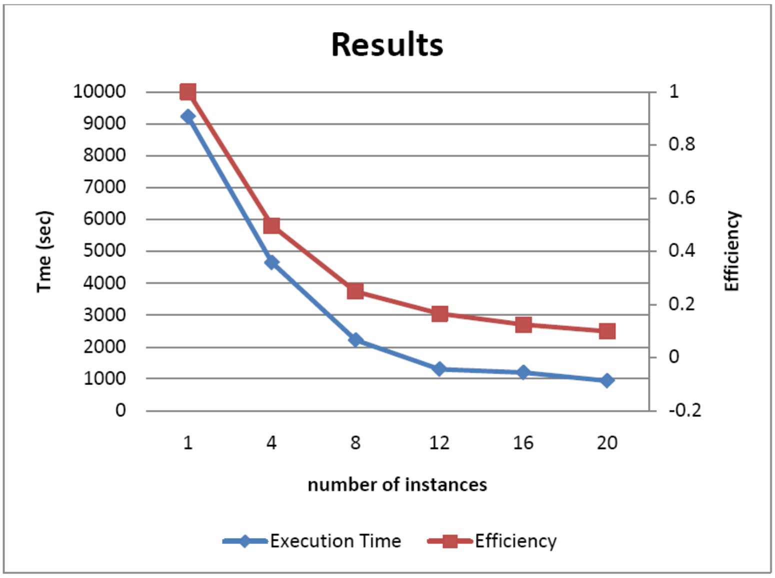
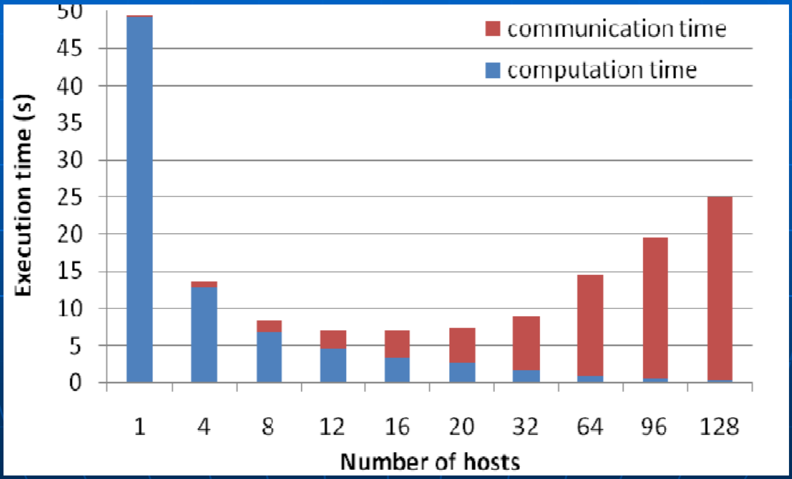


Figure 2 Performance and Efficiency of MapReduce matrix multiplication with different number of instances

Results on Computing Time and Communication Time



Some Observations :

- **Block size** is very sensitive to the speedup performance and implementation efficiency of the MapReduce process. The optimal choice should match with the cache size of the server nodes used.
- The **speedup** is slowed down by many overhead factors, such as data I/O and replication times, intermediate < key, value> matching, storing and retrieval, sorting and grouping, and the parallel task scheduling overheads, etc.
- The optimal **number of server or VM instances** is a direct function of the matrix order (n), effective dot product computing using GPU subcluster, and the reduction of all sorts of delays caused by parallelism handling, communication latency, memory and I/O overheads, etc.