1 Overview

MapReduce\cite{5} is an emerging programming model that utilizes distributed processing elements (PE) on large datasets. With this model, programmers can write highly parallelized code without explicitly dealing with task scheduling and code parallelism in distributed systems.

In this paper, we comparatively evaluate the performance of MapReduce model on Hadoop\cite{2} and on Mars\cite{3}. Hadoop is a software framework for MapReduce on large clusters, whereas Mars builds the framework on graphics processors using CUDA\cite{4}. We design and implement two practical programs, Document Similarity Score and Matrix Multiplication, to benchmark these two frameworks. Based on the experiment results, we conclude that Mars is up to two orders of magnitude faster than Hadoop, whereas Hadoop is more flexible in dependency to dataset size and shape.

The rest of the paper is organized as follows. Section 2 describes the parallel design of the two benchmarks, Document Similarity Score and Matrix Multiplication, as well as their target architecture. Section 3 presents our evaluation implementation and result. Finally, we conclude our work in section 4.

2 Parallel Design

In this section, we discuss the parallel design and implementation mechanisms for the benchmark problems, and the target architectures to which they apply.
2.1 Problem Definition

The benchmark problems we implemented are described as below.

2.1.1 Similarity Score

Computing Document Similarity Score is a common task in web search and web document clustering. This problem takes a set of documents as input. Each document is preprocessed and represented as a feature vector \( \vec{doc} \).

Given two document feature vectors \( \vec{d}_1 \) and \( \vec{d}_2 \), the similarity score (SS) of these two documents is defined as

\[
SS = \frac{\vec{d}_1 \cdot \vec{d}_2}{||\vec{d}_1|| \cdot ||\vec{d}_2||}
\]

(1)

The SS of each pair of document features in the input set will be calculated using equation (1). The output is produced as a sorted list of SS over all pairs.

2.1.2 Matrix Multiplication

Matrix multiplication is widely applicable to a lot of research areas such as computer vision, image processing and artificial intelligence. In this final project, what we are focusing on is to analyze the relationship between documents, which is part of the information retrieval area. As we mentioned in section 2.1.1, the feature of the document could be described as a N-dimensional vector. The matrix could be regarded as the set of vectors, which means the features of a group of documents could be described as a matrix.

So the matrix computing will be very useful when it is needed to analyze the relationship between one group of vectors to the others. For example, suppose we have two web documents A and B with n queries in documents A and m queries in B. So these two documents could be described as follows:

\[
A = \begin{pmatrix}
    a_{11} & a_{22} & \cdots & a_{n1} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_{1m} & a_{2m} & \cdots & a_{nm}
\end{pmatrix},
B = \begin{pmatrix}
    b_{11} & b_{21} & \cdots & b_{m1} \\
    \vdots & \vdots & \ddots & \vdots \\
    b_{n1} & b_{n2} & \cdots & b_{nm}
\end{pmatrix},
\]

where column \( i \) in \( A \) is the \( i \)th query of \( A \), and row \( i \) in \( B \) is the \( i \)th query of \( B \). The similarity of documents A and B should include the similarity of...
each query between A and B. Refer to equation 1

\[ SS = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|} \]

where \( \vec{a} \) and \( \vec{b} \) are the vectors that we want to compare. So for A and B, we need to compare each query of them, which could be achieved as:

\[
A \cdot B = \begin{pmatrix}
a_{11} & a_{22} & \cdots & a_{n1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{1m} & a_{2m} & \cdots & a_{nm}
\end{pmatrix} \cdot \begin{pmatrix}
b_{11} & b_{21} & \cdots & b_{m1} \\
\vdots & \vdots & \ddots & \vdots \\
b_{n1} & b_{n2} & \cdots & b_{nm}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
a_{11}b_{11} & a_{11}b_{21} & \cdots & a_{11}b_{m1} \\
a_{12}b_{11} & a_{12}b_{21} & \cdots & a_{12}b_{m1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{1m}b_{11} & a_{1m}b_{21} & \cdots & a_{1m}b_{m1}
\end{pmatrix},
\]

\((A \cdot B)_{ij}\) means the vector product between the \(i\)th query and the \(j\)th query. This approach will be better than using for loop to calculate \(n\) times.

2.2 Identifying Concurrency

The data of both Similarity Score and Matrix Multiplication can be decomposed into units on which can be processed relatively independently.

In the Similarity Score problem, the documents are preprocessed and represented as vectors. Each pair of them can be viewed as a data unit. Thus we can split the whole set of vector pairs into appropriate number of subsets and compute the similarity score within each subset in parallel.

For Matrix Multiplication, we split the two matrices into blocks. If the blocks are small enough (could be put into the cache of CPUs), each pair of matrices could be calculated on a single node in the cluster very fast. When the web document is too large (for example, a web document which contains millions queries and the feature of each queries is a 128-dimensional vector), decomposing the matrix will be necessary since the memory of the single node could not hold such large data. Each node in the cluster will compute the matrix block pair in parallel, after that the results will be combined as a whole large matrix.

2.3 Algorithmic Structure

The algorithmic structure we used here is Geometric Decomposition pattern.
2.3.1 Similarity Score

The input dataset, which is physically represented as a big text file, is slicing horizontally into chunks. Each chunk corresponds to a subset of vector pairs that we split in Identifying Concurrency section.

2.3.2 Matrix Multiplication

As we discussed in the last section, the matrices could be decomposed into blocks. So for each time, each sub-matrix pairs will be multiplied by the single node and the result will be added to the running matrix sum. The Geometric Decomposition pattern is based on decomposing the data structure into chunks (square blocks here) that can be operated on concurrently[7].

2.4 Supporting Structures

We are programming on MapReduce model, which makes the source code structure as Master/Worker by nature. In addition, Matrix Multiplication uses Distributed Array data structure. As we mentioned before, we partitioned the matrix between multiple UEs among the clusters, the thing is how could we do this making the resulting program both readable and efficient. We use distributed array in this task. The advantage of distributed array is as follows[7]:

- Load balance. Because a parallel computation is not finished until all UEs complete their work, the computational load among the UEs must be distributed so each UE takes nearly the same time to compute.

- Effective memory management. Modern microprocessors are much faster than the computer’s memory.

- Clarity of abstraction. Programs involving distributed arrays are easier to write, debug, and maintain if it is clear how the arrays are divided among UEs and mapped to local arrays.

Remember that our approach is to partition the matrix into blocks and then assign those blocks onto the nodes of the cluster. Suppose we want to
partition matrix $A$, the distribution of matrix $A$ could be as follows:

$$
\begin{pmatrix}
    a_{11} & \cdots & a_{1k} \\
    \vdots & \ddots & \vdots \\
    a_{jk} & \cdots & a_{jk} \\
    a_{j+1k} & \cdots & a_{j+1m} \\
    \vdots & \ddots & \vdots \\
    a_{nk} & \cdots & a_{nk} \\
\end{pmatrix}
\begin{pmatrix}
    a_{1k+1} & \cdots & a_{1m} \\
    \vdots & \ddots & \vdots \\
    a_{jK+1} & \cdots & a_{jm} \\
    a_{j+1m} & \cdots & a_{j+1m} \\
    \vdots & \ddots & \vdots \\
    a_{nm} & \cdots & a_{nm} \\
\end{pmatrix}
\begin{pmatrix}
    j \\
    n-j
\end{pmatrix}
$$

From the above distributed matrix, we could see that matrix $A$ are partitioned into four blocks, the dimensions of the four sub-matrix is $(k, j)$, $(m-k, j)$, $(k, n-j)$, $(m-k, n-j)$. For the matrix multiplication, suppose we have matrix $A$ and matrix $B$ which are partitioned as

$$
A = \begin{pmatrix}
    A_{11} & A_{21} \\
    A_{21} & A_{22}
\end{pmatrix},
B = \begin{pmatrix}
    B_{11} & B_{12} \\
    B_{21} & B_{22}
\end{pmatrix}
$$

The matrix multiplication will be equal to the combination of the sub-matrix multiplication that is as follows:

$$
A \cdot B = \begin{pmatrix}
    A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
    A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{pmatrix}
$$

### 2.5 Implementation Mechanisms

In MapReduce programming model, our algorithms are implemented using two functions, Map and Reduce. The Map function takes an input key/value pair and outputs a set of intermediate key/value pairs. All intermediate values are partitioned into groups according to the keys. The Reduce function accepts the intermediate key/value pairs and produces a sorted list of key/value pairs:

$$
Map : (k_1, v_1) \rightarrow list(k_2, v_2)
$$

$$
Reduce : (k_2, list(v_2)) \rightarrow list(k_3, v_3)
$$

#### 2.5.1 Similarity Score

The scheme of Similarity Score MapReduce program is define as follows:

Map function:

$$
InputSchema : K = \{\}, V = \{docId_i, vec_i, docId_j, vec_j\}
$$
Outputschema : \( K = \{ \text{SimilarityScore} \}, V = \{ \text{docId}_i, \text{docId}_j \} \)

In this program, no reduce stage is required. The number of reducer is set to 1 because we need a sorted list of the overall results.

2.5.2 Matrix Multiplication

The input to Matrix Multiplication program consists of a set of pairs in the form of \( \text{MatrixIndex} \) and \( \text{MatrixValue} \) of the matrix, where \( \text{MatrixIndex} \) means the index of the input matrix, and the \( \text{MatrixValues} \) means the corresponding value to that \( \text{MatrixIndex} \). In our problem, if we want to calculate the multiplication of matrix \( A \) and \( B \), the \( \text{MatrixIndex} \) would be the element index from \( A \) or \( B \), and the \( \text{MatrixValues} \) would be the element value corresponding to that index. For example, suppose \( a_{i,j} \) is one of the elements in matrix \( A \), so the \( \text{MatrixIndex} \) would be \( (i, j) \) and the \( \text{MatrixValues} \) would be \( a_{i,j} \).

For the Map function, the schema is defined as:

\[
\text{InputSchema} : K = \{ \}, V = \{ \text{MatrixIndex}, \text{MatrixValue} \}
\]

\[
\text{Outputschema} : K = \{ \text{MapperOutputKey} \}, V = \{ \text{MapperOutputValue} \}
\]

The key/value pair \( \text{MapperOutputKey} \) and \( \text{MapperOutputValue} \) are two kinds of data type that we extends from \( \text{WritableComparble} \) and \( \text{Writable} \) class in Hadoop library. The format of \( \text{MapperOutputKey} \) is as follows:

\[
\text{MapperOutputKey} = (\text{BlockRowIndex}, \text{BlockColumnIndex}, \text{IsFromA})
\]

where

\[
\text{BlockRowIndex} = \frac{\text{MatrixRowIndex}}{\text{BlockRowSize}},
\]

\[
\text{BlockColumnInde} = \frac{\text{MatrixColumnIndex}}{\text{BlockColumnSize}},
\]

\[
\text{IsFromA} = \text{True} \{\text{if fromA}\} \text{or False} \{\text{otherwise}\}
\]

The format of \( \text{MapperOutputValue} \) is:

\[
\text{MapperOutputValue} = (\text{BlockRowNum}, \text{BlockColumnNum}, \text{value})
\]

where the \( \text{BlockRowNum} \) and \( \text{BlockColumnNum} \) indicate the position of the block in the matrix, and the \( \text{value} \) means the value of the element. So the mappers partition the matrices into blocks and emit the element’s block
index as the key and the block position in the matrix and the element’s value as the value to the reducer.

For the Reduce function, the schema is as follows:

\[ \text{Inputschema} : K = \{ \text{MapperOutputKey} \}, V = \{ \text{MapperOutputValue} \} \]

\[ \text{Outputschema} : K = \{ \text{MatrixIndex} \}, V = \{ \text{MatrixValue} \} \]

The reducer multiplies the A and B block pairs and sums up the results.

2.6 Target Architecture

For Hadoop, we choose Amazon Elastic MapReduce cloud computing service. 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)\[1\]. It has been widely used in businesses and researches, which is feasible to demonstrate the Hadoop framework. As in the cloud, the computing units are virtual machine (VM) based. Each VM serves as a PE in the cluster. Each PE runs one or more map/reduce tasks.

Mars is implemented using NVIDIA CUDA, thus we demonstrate Mars on a piece of graphic card that supports CUDA computing architecture. The data domain, including the input data, the intermediate result and the final result are stored in GPU and are mapped into a grid on CUDA execution model. The grid is decomposed into thread blocks. Each map/reduce task runs the kernel with its own data in a thread block.

3 Evaluation

In this section, we present the performance for Hadoop and Mars on the benchmarks we describe above.

3.1 Experimental Setup

Hadoop experiment was performed on two High-CPU Medium Instance, with each instance having 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), 350 GB of instance storage and 32-bit platform. One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

The Mars experiment was running on a Macintosh with a GeForce 9600M GT GPU and a 2.66 GHz Intel Core 2 Duo processor running Mac OS X
10.6.5. The GPU consists of 4 multi-processors, each of which has 8 cores running at 1.25 GHz, and the device memory of the GPU is 256 MB. The version of the installed CUDA is 3.2.

3.2 Experiment Data Selection and Evaluation Methodology

To evaluate the performance, we select different size of dataset size for each program: small size, medium size and large size, as shown in Table 1. The document feature dimension for Similarity Score is 128.

<table>
<thead>
<tr>
<th>Application</th>
<th>Small</th>
<th>Media</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similarity Score</td>
<td>256 × 128</td>
<td>512 × 128</td>
<td>1024 × 128</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>256 × 256</td>
<td>512 × 512</td>
<td>1024 × 1024</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Small</th>
<th>Media</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similarity Score</td>
<td>89 M</td>
<td>357 M</td>
<td>1.4 G</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>5 M</td>
<td>10 M</td>
<td>40 M</td>
</tr>
</tbody>
</table>

We generate random data using the program we developed. Before we started running the Mars and Hadoop benchmarks, we found that Mars has a limitation on the dataset size selection. When the data is too large to fit into the GPU’s memory (for example a 2048 × 2048 matrix multiply by itself, or a 2048 × 128 document set), it will run out of memory and return error. This is due to the current imperfection of the Mars framework. So we select these datasets to make sure each one could be loaded into the memory and the program can run successfully (Although the input data of Matrix Multiplication program is relatively small, but it generates many intermediate results while running the map function).

3.3 Results

We present the results and analyze the performance of the two benchmarks in three parts.
Figure 1: The performance speedup of Mars over Hadoop

3.3.1 Overall Performance Speedup of Mars Over Hadoop

Figure 1 shows the overall performance speedup of Mars over Hadoop with the data set size varies. Overall, Mars is around 100-200x faster than Hadoop when the input data is in medium size. This is predictable because GPU has a much higher I/O bandwidth and more UEs than the cluster. However, for computation-intensive applications such as Matrix Multiplication, the speedup of Mars is not as good as it was when data size is large.

We can see that from figure 1, the speedup of Mars is fairly high when the data size is small, then the speedup goes down when the data size become large. In contrast to Mars’ performance, Hadoop’s performance will become better when the data size changing from small to large. This is due to two reasons. First, GPU has limited memories while memory is not an issue in cluster with our data size. In fact, when we tried to raise the data size to two times bigger than the large one, we found that Mars will crash for running out of memory, but Hadoop still works. Moreover, startup cost is much higher in cluster compared to GPU, especially in the experiment of small dataset size.
3.3.2 Time Breakdown of Mars

Figure 2 shows the time breakdown of Similarity Score and Matrix Multiplication on Mars. The total execution time of the programs can be divided into four parts: the file I/O time, the mapper execution time, the sorting time and the reducer execution time [6]. Here the mapper execution time includes the time copying the input data to the device memory and the reducer execution time includes the time writing the result to the main memory.

From Figure 2, we can see that the sorting time is dominant in Similarity Score and mapping time is dominant in Matrix Multiplication. This is decided by the nature of their algorithm. Recall that the mapper of SS calculates the score by simply multiplying two vectors and then emits all the intermediate data to one partitioner where all the data are to be sorted. GPU is very good at doing multiplication but not sorting. In contrast, the map function of MM has to split the input data into chunks, which involves time consuming calculation such as division and mode.

We can also see that the I/O time in GPU is so fast that it is negligible.
in the overall execution time. But in cluster, files are read from and written
to hard disks, of which bandwidth is two orders of magnitude slower than
GPU’s.

3.3.3 Hadoop Speedup of Local Machine over Cluster

Next we present the performance speedup of Hadoop on Local Machine
over Cluster. Hadoop on Local Machine means the Hadoop MapReduce
program is running in a non-distributed mode. As we discussed in the
section 3.1, we used 10 EC2 Compute Units in total, for each is equivalent
to a 1.0-1.2GHz 2007 Opteron or 2007 Xeon processor. Our local machine
has a 2.66GHz Intel Core 2 Duo processor. It is no doubt that the hardware
of Amazon Instance are superior than the local machine. The reason we
do this experiment is to compare the performance between the distributed
mode and the non-distributed mode, where we found something interesting
here.

In figure 3, we can see that the local machine performs about 5x faster
than Amazon Instance when the dataset is small. With the large dataset,
the speedup of the local machine shrinks to about 1.7, however, the local
single node machine still out performs a cluster with 10 node.

From the results we have, it is clear that the bottleneck of performance
in distributed cluster is I/O and startup cost. In clusters, computing nodes
not only heavily rely on disk I/O but also on the network to exchange the
intermediate results. Both of them are far more slower than the memory
I/O in local machine, not to mention the GPU I/Os. Only when dataset
size becomes large enough to allow the mapper stage and reducer stage
execution time dominate the total execution time, the cluster can show its supercomputing capability.

4 Discussion

4.1 Summary

This paper evaluated the performance of Mars and Hadoop by providing two benchmarks: Similarity Score and Matrix Multiplication. And three different size of dataset has been generated in the experiment. We discussed each of the benchmark from identifying concurrency, algorithmic structure, supporting structure to implementation mechanisms. In the experiments, we compared the performance of two MapReduce program implemented in different frameworks: Hadoop and Mars. The benchmark result shows that Mars out performs Hadoop greatly in this overall experiment. However, Hadoop is much flexible in terms of dataset size selection. Moreover, the performance gap between Hadoop and Mars becomes smaller when dataset size increases. In the mean time, Mars’ performance could be improved significantly when the sorting performance can be optimized. At last, we compared Hadoop’s performance between single node machine and the distributed cluster, from which we learn that the bottleneck of cluster is its I/O bandwidth and startup overhead.

4.2 Future Work

We are interested in extending the experiments on Mars. First, in this paper, our benchmarks run on a graphic card which memory is very limited. We would like to run the benchmarks on Nvidia Tesla or Nvidia Quadro graphic cards to see the different performances when we add more multi-processors and much more device memories. Further more, we are interested in applying Mars framework to distributed system, where we can take advantage of the huge storage and improve the overall computing ability as well. A more interesting probe is to develop programs that can be run on a SLI-video card system, which consists of two or four graphic cards with all of them share the memory of the host.

4.3 Lessons Learned

From our results, we are not surprised that Mars has such a good performance since it enjoys the highly parallelized architecture and huge band-
width provided by GPUs and CUDA. What was surprising is that the performance of Hadoop on Amazon Web Service is so weak compared to the non-distributed version that runs on a local computer. While parallelize programming might bring us faster programs running on local machines as we saw in our experiments, its real power unveils in the supercomputing cloud where program demands highly data-intensive computing.

References


