Large Scale Machine Translation Architecture

Qin Gao
Language Technology Institution
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
qing@cs.cmu.edu

Abstract

Parallelization is widely considered to be the future of high performance computation, and is a natural choice when scaling up the machine translation systems. In this report, a programming model called MapReduce is investigated and two supporting components for MapReduce framework to work efficiently are analyzed, namely the distributed storage for streaming data and distributed storage for structural data.

With the analysis of existing softwares, solutions for parallelizing several important tasks in machine translation are proposed.

1 Overview

Most machine learning technologies require large amount of data, so does machine translation. However, as the amount of data increases, the algorithms and computational facilities to handle these data becomes a problem. A first issue to be tackled is navigating or processing the large amount of data, recently, many corpora contain several terabytes of data, and even a single navigation through the data is impossible. Moreover, the representation of knowledge extracted from the data may also be very large. For example, the lexicon models of GIZA++ may contain several million translation probability entries, and the phrase tables can be even larger. N-gram language model is another example, for a fixed vocabulary size $|\mathcal{V}|$, the number of entries in a n-gram language model can be a fraction of $|\mathcal{V}|^n$. When n grows large, training and handling the models can be prohibitively large than most state-of-art computers. In case that we want to scale up the machine translation systems, many tasks such as word alignment, decoding and parsing are facing severe problems as mentioned above.

The development of computing technology provides abundant computation resource. However, the majority of computation resource is not within one single super computer, but rather spread out to a large amount of commodity PCs or general purpose servers. Google(Barroso et al., 2003) has set an example for making use of large clusters of commodity computers to provide computation power for large jobs and developed several computation model for what can be done efficiently in this kind of infrastructure. The most famous one is MapReduce. As we will see later in this paper, MapReduce is suitable for algorithms that falls into "massive unordered distributed" (*mud*) algorithmic model. However, in order to make the MapReduce framework suitable for most of machine translation algorithms, we need to have a reasonable data accessing model for both streaming and structural data.

The report will first investigate some representative algorithms in machine translation, and show how MapReduce algorithm can deal with them in the second section. In the third section we will discuss supporting software for MapReduce to work efficiently, including distributed storage for streaming data and distributed storage and accessing mechanism for structural data. The conclusion is given in the fourth section.

1.1 Representative Problems in Machine Translation

1.1.1 Counting

First we start by investigating several common tasks in machine translation. The simplest one may be counting the occurrence of unigrams in a large

body of corpus.

The algorithm is straight-forward, first of all, we should have a "map" from token to its count. Then, for each word in the corpus, we add the count of the token in the table by 1.

The unigram count can be extend to many other "counting" tasks, such as counting the total number of n-grams or the total number of a particular phrase in the corpus.

1.1.2 Sorting

The sorting task is also common in machine translation. For example, when we need to score the extracted phrases, we need to sort the phrase table to make sure all English phrase translations for an foreign phrase are next to each other in the file.

A merge sorting algorithm can be expressed as follow:

1. Divide the unsorted list into two sublists of about half the size
2. Divide each of the two sublists recursively until we have list sizes of length 1, in which case the list itself is returned
3. Merge the two sublists back into one sorted list.

Counting and sorting seems to be different, however, they share a common characteristic. Both algorithm does not need information other than the data itself, in contrast with decoding/parsing and EM algorithm in next two sections.

1.1.3 Decoding/Parsing

The tasks here are a little more complicated. However, if we leave the detail of the algorithms along, this kind of tasks does the following:

1. Load a certain kind of model, which is generally a map function from token sequence to a score in the real field, namely
$$f : \mathcal{V}^n \to \mathcal{R}$$
2. For each input token sequence in the data $\{v_1, v_2, \cdots, v_m\}, v_i \in \mathcal{V}$, find the optimal combination of subsequence set $V = \{V_1, V_2, \cdots, V_k\} V_i \in \mathcal{V}^n$ that cover the whole input and gives the highest score. Namely
$$V^* = \arg\max_V \sum_{V_k \in V} f(V_k)$$

The most important operations in the algorithm are the computation of $f(V)$ and the optimization. In most case, the $f(V)$ involves looking up a table, sometimes very large. With difference characteristic of the $f(V)$, the optimal way to solve the problem varies.

1.1.4 Iterative Optimization

Another common task is iterative optimization, the most representative one may be EM algorithm. The most significant property of EM is that it needs to iterate the following steps:

1. Compute the likelihood of all the samples, using the model from previous step. (E Step)
2. Accumulate the likelihood and perform normalization/optimization to get new parameters for the model. (M Step)

Therefore, it needs to load the model and process all the data, just like decoding/parsing tasks. Also, it must go over all the data and collect the counts, that is similar to counting.

1.1.5 Language Model Training

When training language model, getting the n-gram counts from data is similar to the counting algorithm, however, language model also need to estimate the smoothing (back-off) factors, and perform "normalization" over the counts and backoffs. In this case it is similar to the normalization step in EM algorithm.

1.2 Comparison of Algorithms

We listed typical algorithms in machine translation. All these algorithms share a common property, when dealing with data, which can be arbitrarily large, every element in the data is independent of each other. For example, when parsing data, the result of one sentence would not affect the result of another sentence, and in EM algorithm, the likelihood of one sentence in one iteration is independent of another sentence. Stated in another way, the algorithm is "unordered", which means the final result will be the same if we process the data in different order. This property is very important for parallel processing of the data, because it ensures a small fraction of data can be processed independently by a node without waiting the result for other fractions.

However, the algorithms are not completely the same. For counting and sorting, all the information is contained in the data, the program does not require any additional model. Parsing/decoding require models to process the data, in many cases the model itself become the major problem, rather than the data. A good example is loading and looking up the language models in decoding. In contrast, Language model training does not require external information, however, it requires a complicated process over the result got from the data, and in most case, the result is not "unordered", and can hardly be parallelized. For EM, the E-step is similar to parsing/decoding and the M-step is similar to language model training.

We can see from the analysis that besides data, another commonly encountered object in machine translation is the models. In most case the model is a "map", how to iterate or randomly access the map entries is another key point to be solved.

Therefore we can conclude that in machine translation we are dealing with two important objects:

- Data: Linear, unordered, we can call it "streaming data"
- Knowledge: Non-linear, ordered, structural, we can call it "structural data'

Figure 1 shows a high-level representation of the model of process in machine translation.

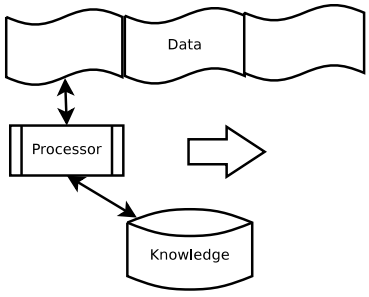

Figure 1: A general model of process in machine translation

1.3 Bottlenecks

Knowing the two objects we are dealing with, we can analyze the bottlenecks in large scale machine translation. For massive unordered data, the time complexity is the major problem. As the size of data grows, the processing time also grows linearly, to some point a single processor can not finish even one navigation over the data. However the problem is more easy to deal with, because the data is unordered, applying the "divide and conquer" strategy is natural. By partitioning the data and processing them in computer clusters, the problem can be solved. Figure 2 shows the model of parallelized machine translation algorithms. Google's MapReduce library(Dean and Ghemawat , 2008)and open source project Hadoop(Bialecki et al., 2005) together with distributed storage for streaming data such as Google FS(Ghemawat et al., 2003) and HDFS(Borthaku, 2007) are successful stories on solving the problem.

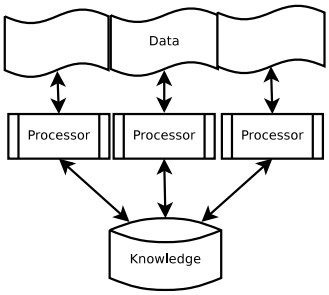

Figure 2: Parallelizing the machine translation algorithms

In the other side, the map object is ordered, and the algorithms on the maps can hardly be parallelized. Therefore, may "normalization" algorithms have to be performed on a single machine. In many cases, the maps are super-linear with respect to the size of data, and requires a huge amount of storage. When accessing the maps, hash table or other sophisticated data structures such as suffix array can provide fast access to the data, however, the size of the map often prevents the model from being loaded by a single machine. That is why caching and filtering technologies are useful for many tasks. To summary, the map object requires a distributed storage and fast access. Currently these kind of software is being actively developed. The "BigTable"(Chang et al. , 2006) software developed by Google and open source counterpart "HyperTable" [1] provides a common solution to most of these problems.

[1]Documentations are available at http://code.google.com /p/hypertable/ wiki/HowHypertableWorks

2 MapReduce

MapReduce is proposed by Jeffrey Dean and Sanjay Ghemawat in Google Inc. It can refer to the programming model and the Google implementation of supporting hardware and software infrastructure for the model. There is also an opensourced implementation of the model called Hadoop, written in Java and is now widely used in the community of "Cloud Computing". The MapReduce framework is considered a good solution for many machine learning problems.(Chu, 2007).

2.1 Programming Model

The MapReduce model abstracts the computation into two functions, *Map* and *Reduce*, and consider the input and the output to be key/value pairs. The user should supply the two functions, and the MapReduce library will take care of partitioning data, combining intermediate results and other common tasks.

The *Map* function takes *an* input key/value pair (k, v) and produces *a set of* intermediate key/value $\{K^1 : V^1, \cdots K^n : V^n\}$ pairs, and the MapReduce library groups all intermediate values associated with the same intermediate key K and passes them to the *Reduce* function. The map function can only access one key/value pair, and it is impossible for it to know any properties depend on other key/value pairs.

The *Reduce* function accepts an intermediate key K and a set of values for that key. It merges these values and outputs a smaller set of values. Notice that the *Reduce* function is called for one key K at a time together with all the values associated with the value, therefore inside the reduce function, it is impossible to know the computational results of other keys — it actually prohibited the implementation of algorithms that two keys are interdependent on this framework.

The purpose of defining two functions and restricting the parameters they can take is to ensure the algorithm can be divided and run in different nodes without impact the result. So, when designing MapReduce program the guideline must be followed:

- Carefully define the input key/value, and make sure the no interdependence exists. For example, it is suitable to define every word to be the key for word counting, while it is invalid to do so when counting bi-gram occurrence or sorting the sentences. Also, the intermediate key should be carefully defined to avoid dependency.

Before the *Reduce* function can be executed, the output must be grouped by the keys. The step is done by sorting all the records using the key. And in the implementation of Google, the order of final output is also guaranteed to be ordered according to the keys. The implicit sort procedure can make many things easy, as we can see later.

Below we will visit several examples of how to express algorithms with MapReduce. Most of these examples are taken from (Dean and Ghemawat , 2008).

2.2 Examples

2.2.1 Distributed WC

Here we want to count the total number of words in a corpus. We want only one value C for final output, and we just need one common intermediate key. We can have two different ways to define input key/value pair, either be one word of the corpus or one sentence/document of the corpus. For either case, the intermediate key takes only the common value. If we define the input value by one word the intermediate value will be 1, and if we define the input value by sentence/document, the intermediate value will be the word count of that sentence/document. It is easy to see that defining a reasonably larger gratuity for the input can be more efficient, namely the sentence/document.

The algorithm can be expressed as:

```
map(String key, String value):
 // Key: sentence offset or document name
 // Value: content of sentence/document
 count = 0;
 for each word w in value:
   count += 1;
   EmitIntermediate("c",count);


reduce(String key, Iterator values):
 // Key: common key, "c"
 // values: a list of counts
 int result = 0;
 for each v in values:
```

```
  result += v;
  Emit(result);
```

2.2.2 Distributed Unigram Count

We can also follow the previous example, however we replace the common intermediate key by every words, and for each word, simply emit an intermediate key/value pair $(w, 1)$, and the reduce function will sum over all the key/value pairs with the same w. Therefore we have the following definition:

```
map(String key, String value):
 // Key: sentence offset or document name
 // Value: content of sentence/document
 for each word w in value:
   EmitIntermediate(w,1);

reduce(String key, Iterator values):
 // Key: common key, "c"
 // values: a list of counts
 int result = 0;
 for each v in values:
   result += v;
 Emit(result);
```

The definition, however, is inefficient. Every occurrence of a word will emit one key/value pair, the intermediate key/value representation will be almost two times larger than original corpus. Therefore, an early combination is needed. You can either build a Map structure in the map function and count the occurrence yourself, or use the optional abstract function: *combiner*. The combiner is the same as the reduce function, but runs locally to combine the mapper's output before passing the result through slow network connections.

2.2.3 Distributed Sort

The map function extracts the key from each record and emits a $(key, record)$ pair. And the reduce function simply emits all pairs unchanged. Here we make use of the implicit sort procedure of the library. However, the library may not provide interface for defining custom comparison function, therefore the map function is responsible to extract proper keys.

2.3 Implementations

The MapReduce and Hadoop are primarily for large cluster of commodity PCs, therefore the machine failure is common in this case. Also, the storage is provided by normal hard drivers that may not be reliable, so they also developed file systems (Google FS, HDFS) that use replication to provide reliability.

The resource manager is used in both systems, which is responsible for allocating nodes for jobs submitted by users and balance the work load so as to get the maximum throughput from the cluster.

2.3.1 Execution overview

Below is a typical procedure of a MapReduce job(Dean and Ghemawat , 2008), and the flowchart of execution is shown in Figure 3.

1. *The MapReduce library in the user program first splits the input files into M pieces, whose size is controlled by user-supplied parameter. And then it starts up many copies of the program on a cluster of machines.*

2. *One of the copies of the program is the master and 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.*

3. *A worker who is assigned a map task reads the contents of the corresponding input split. It parses the 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.*

4. *Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.*

5. *When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, and external sort is used.*

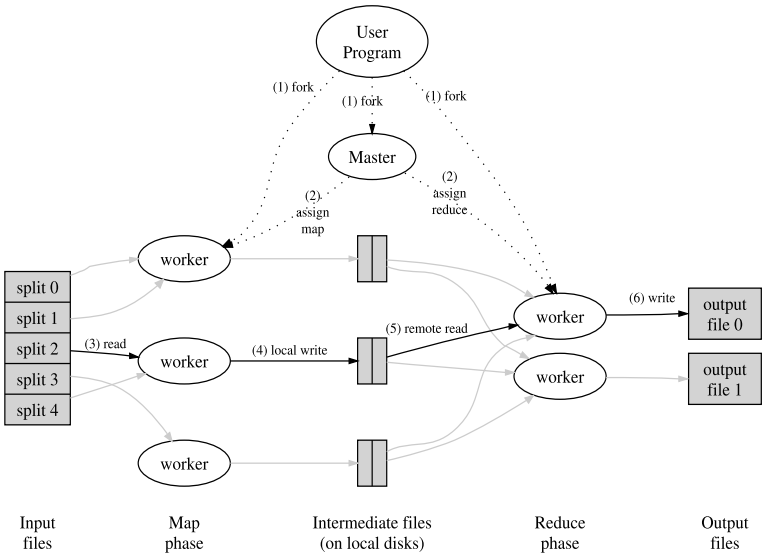

Figure 3: The execution flowchart of Google MapReduce library(Dean and Ghemawat , 2008)

6. *The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate 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. At this point, the MapReduce call in the user program returns back to the user code. Now the output of the MapReduce execution is in R output files, user may combine them to be an output, or use them directly for other MapReduce tasks.*

The execution procedure shows a first merit of MapReduce: the partitioning of data and combination of data is taken care of by the library, so users do not need to perform these tedious tasks again and again.

2.3.2 Fault Tolerance

In case that a task use one thousand commodity PCs, the failure becomes common, and the library must allow the fault to append and handle it gracefully. Google's MapReduce and Hadoop both implemented fault tolerance. Generally speaking, the master will keep on contacting the workers and if one worker does not responce or report a failure, the tasks in the worker will be reassigned to other workers. In contrast, if the master is failed, the job itself cannot be recovered. However, because only one machine out of one thousand is the master, the probability of master failure is rare.

3 Data IO and Memory

As we have seen in the previous section, the MapReduce provides a good programming model for many tasks. However, by investigating the examples shown above, all the tasks falls in the first category we discussed in the section 1, namely the tasks that require only the information from the data. Then, what about the other kind of processes like parsing, decoding and word alignment? These kind of processes requires reading external data, sometimes quite large. In this section, we will first visit the Google FS and HDFS, and then discuss the merit and deficiency of these file systems.

3.1 Distributed Streaming File System for MapReduce

3.1.1 Design

MapReduce is especially good for dealing with "mud" namely "massive, unordered, distributed" problems, therefore they focuses mainly on provide good support for stream reading/writing of

large files. Considering the hardware on which the file systems are built is unreliable commodity PCs, replication is also important. Below is the design assumptions of Google FS quoted from (Ghemawat et al., 2003):

- *The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly component failures on a routing basis.*
- *The system stores a modest number of large files. We expect a few million files, each typically 100MB or larger in size. Small files must be supported, but does not need to optimize for them.*
- *The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reading, successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.*
- *The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary position in a file are supported but do not have to be efficient.*
- *The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Atomicity with minimal synchronization overhead is essential. The file may be read later or be read through simultaneously.*
- *High sustained bandwidth is more important than low latency. Most target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.*

In the design of HDFS, the assumption is even strict, which states:

"HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future."

For simplicity, the random write is disabled. However, for many tasks, the append-writing may still be needed.

3.1.2 Implementation

For both file systems, the data are segmented into several *chunks*. The chunks are stored "chunkservers", which are common machines in the network. A master is responsible for supplying the chunk location to clients, and the client, after receive the location of the chunk, goes directly to the chunkserver for the data.

Figure 4 is the architecture of Google FS.

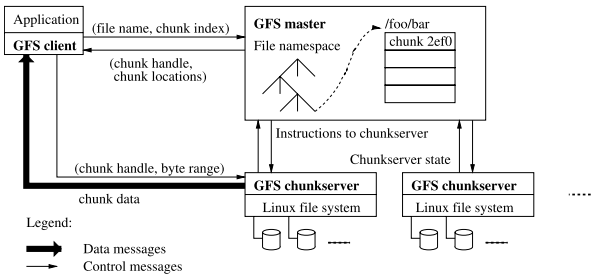

Figure 4: The architecture of Google FS(Ghemawat et al., 2003)

In order to ensure the integrity of data, and (possibly) for accelerating simulatenous reading, the chunks are replicated in several nodes.

After reading all the information above, we can observe a strong relationship between the two distributed file system and MapReduce. Here, the data are already chunked and in many cases, tasks can run locally to avoid the network traffic. Therefore, the design and implementation fits the streaming IO perfectly.

3.1.3 Performance

Several experiments are performed in (Ghemawat et al., 2003), and the results show that as the number of clients increases, the aggregate read rate is very closed to the network limitation, as shown in Figure 5[2]. The writing rate is also very high if take replication into account. However the appending operation is not so promising. Therefore, the decision of not supporting appending operation by HDFS is reasonable.

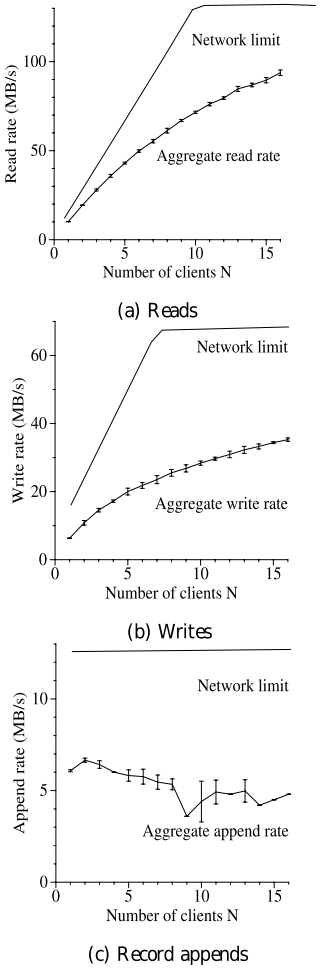

Figure 5: The aggregate throughputs of different operations on Google FS(Ghemawat et al., 2003)

[2]As you can see in the plot, the network limitation of different operation is different. That is because when write operation is performed, replication also require network bandwidth, and append operation requires modification on the metadata tables and also requires may seeking operations.

3.2 The IO of structured data

Given that Google FS and HDFS are good for streaming data, the efficient solution, it still does not solve the problem that external knowledge source is required. In case that the size of external knowledge is small – or modest, as long as it can be loaded into memory, it is possible to replicated them to local disk and read it into memory. However, this solution is not perfect for huge tables. For example, in decoding, the phrase tables are too large to be loaded into memory. The same problem exists in language models, for long-span n-gram language models, the number of entries may be tens of gigabytes.

A common solution for handling the huge language models or phrase tables are to filter the entries according to the sentence to be translated, however, in current framework, for each sentence or several sentences we need to perform a complete navigation on the whole model. Therefore, we do require a more efficient IO for these kind of models.

In this section we will discuss a distributed storage system for structural data implemented by Google, with the name "BigTable". And also the opensource counterpart "HyperTable" which based on Hadoop.

3.2.1 Data Model

The "BigTable" is designed to be a sparse, distributed, persistent multidimensional sorted map. The data model of BigTable can be treated as a four-dimension table, the four dimensions are:

1. Row: The primary key. The table is segmented into several chunks or so-called "tablets", and each tablet will contain a region of rows. Therefore, a certain row will not be stored separately.

2. Column family: Secondary key. The number of column family should be fixed[3], in that sense it can be treated as "columns" in databases. As we will see later, the column family is so called "access control" unit, and can be randomly accessed in high speed.

3. Column: One column family may contain any number of columns. We can also treat it as a the

[3]Although insertion and deletion is permitted, the overhead is very high and is discouraged.

row of a second table. The name of the column can be any arbitrary string. For example, if we defined a column family called "target_phrase" in a phrase table, the columns can contain the content of target phrase as the key. (However, that will not be efficient, according to the access method.)

4. Timestamp: By accessing from Row → Column Family → Column, we already reach the "cell". However, one cell can contain several version of values, and it is indexed by an integer number called "Timestamp", which may represent the time that the record is append.

5. Value: The value does not have any enforced data type or structure, instead it is treated as binary stream.

To summarize, the data model of big table is a four-dimension table, the first and third dimension are strings, the second dimension is (semi-)fixed, and the fourth dimension is of integer type.

Although represented as a 4-dimension structure, the randomly access to cells may not be performed directly. The publication contains several sample of BigTable APIs and shows that the randome access may only available for column family. For example, the following code shows the operation required for reading from BigTable:

```
Scanner scanner(T);
ScanStream *stream;
stream =
    scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
  printf("%s %s %lld %s\n",
         scanner.RowName(),
         stream->ColumnName(),
         stream->MicroTimestamp(),
         stream->Value());
}
```

A scanner is a abstraction to iterate over all anchors in a particular row, and the ScanStream has the functionality of fetching a column family and then use the *Lookup()* function to search for a particular column value within the column family. We can see from the code that the random access may only applicable on column family and the column value can only be accessed by performing search inside the column family.

3.2.2 Storage Structure

The BigTable is based on Google FS to store its data and log files. As stated before, the whole table is splitted into several "tablets", and each tablet contains a range of row values. The tablets may be stored in different locations. nodes which hosts the Google file system ("chunk servers"). Instead of referring to Google FS's chunk location query interface, the BigTable has its own hierarchy way of storing chunks. A file is stored in master which contains only the pointer to "Root Tablet". The "Root Tablet" contains the location of all the tablets in a special table named "METADATA" table. The "METADATA" table (which is also a "BigTable") provides location information for all user tables. The "Root Tablet" itself is the first tablet of METADATA table, and is treated specially. Therefore, a query for the location for a user table will only need to go through 3 levels. By applying the hierarchy query method, the client do not need to send request to the Google FS's master node frequently, and the METADATA table itself is distributed stored and distributed accessed. By incorporating caching mechanism in the clients, the tablet information can be fast accessed and the GFS access is minimized.

The storage structure of a tablet is called "Google SSTable", which is actually a local ordered immutable map from keys and values. The keys and values are both unstructured byte streams. There is a block index within the SSTable and it is loaded into memory, each block stores a range of key/value pairs. When looking up for records, the binary search is performed on the index to find the appropriate block and then the block is loaded into memory and then another binary search is carried out to find the records. Alternatively the whole table can be loaded into memory and allows for faster access.

3.2.3 Performance

The performance of BigTable is determined by the total number of tablet servers, and also whether the whole tablets are loaded in memory. As shown in Table 1 and Figure 6, the performance of random reading when all the tablets are in memory is almost 30 times as fast as loading the data from disk. Intuitively, when using 500 tablet servers, up to 3,000,000 records can be fetched in a second.

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Table 1: The number of 1000-byte values read/write per second per tablet server(Chang et al. , 2006)

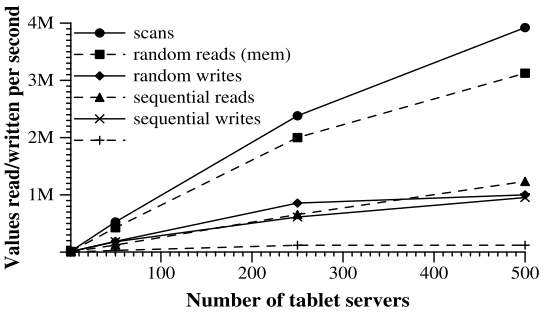

Figure 6: The number of 1000-byte values read/write per second of the whole system(Chang et al. , 2006)

3.2.4 Advantages and possible applications

The BigTable provides a good framework for solving the problems of accessing large structured data, and alleviates the problem of memory and disk IO for these tasks. We may take the decoding problem as an example.

In order to decode a sentence, we may require the following information:

1. All the possible phrase pairs given all the source phrases from the sentence.

2. All the possible reorder model entries given all the source phrases from the sentence.

3. The language model entries for all the possible translations of the source sentence.

All these information can be prohibitively large to be loaded into memory and therefore we need to undergo a filtering procedure. Current method of doing the filtering is simply navigate the whole model and choose those entries meet the requirement. It may be highly inefficient to perform the operation for each sentence and generally several sentences are grouped and the program will provide the filtered results for every sentence in a specific file. Alternatively, by incorporate the BigTable, we can build several tables for these information, for example, we can have a table of phrase pairs:

- Row: Source Phrases
- Column Family: One single constant value
- Column: Target Phrases
- TimeStamp: The id of features
- Cell Value: The value of the feature

Then when one sentence is inputted, all the source phrases are extracted, a large number of queries are formed and sent to several different tablet servers. These tablet servers look up local storage for the required entries and return them through the network.

In this case, the filtering task for one sentence is handled by several CPUs, and each CPU can process request from many decoding servers. Every CPU can play both role and therefore maximize the throughput of the CPU power. The most important advantage, however, is the memory usage. Consider a phrase table with a size of 100 GB, which can hardly be loaded into memory even for most up-to-date servers. If it is distributed stored in 200 tablet servers, only 512MB memory is required for each node to load the whole table into memory and the filtering can be done in a very high speed without accessing disk. The resulting phrase pairs for every sentence, may be much smaller, typically less than 1 GB - which means the network transferring load is also not a big issue.

We can also provide the similar structures for language models and reordering models, and distributed them on the same set of servers, in that case, the workload can be even more balanced.

3.3 An ideal infrastructure for machine translation

By summing up the technologies we described above, we may discuss an ideal infrastructure for machine translation and how it handle our tasks perfectly.

The infrastructure should contain the following three components:

1. MapReduce library and supporting software

2. Support software and library for fast access of streaming data

3. Support software and library for fast access of structural data

And also one important factor is the scale of the cluster, which should be large enough for actual "parallel processing" be carried out.

In the "Google side" the three components are *MapReduce*, Google FS and BigTable, and in the open source side, we have Hadoop, HDFS and HBase/HyperTable.

In case that we have all the three components, most problems in machine translation can be handled in a distributed way. For example, the training of word alignment models can be done by spawning a bunch of aligning tasks using the MapReduce library as the "Map" function, the library takes care of splitting the training data into fragments. The map function asks for the model parameters through the structural data accessing interface, and produces counts as intermediate keys. Finally the normalization process is called as "Reduce" function, where the counts are recombined and normalized. After that a new "BigTable" is produced for each model, which enables next round of optimization to be carried out.

As described above, the decoding can also be done in a efficient way, the models are stored in structural tables and the decoding routines are written into the form of "Map" functions. The decoder may query the phrase table entries and other knowledge directly and a filtering process is carried out implicitly. The library will take care of balancing the load of every "decoding server".

Other tasks such as parsing, sorting, counting can also be fit into the framework easily. And the most advantage of this infrastructure is that it can make full use of all the computational resource it have, one node can be a decoder server, a tablet server or a chunk server in the same time. This characteristic may be essential for real world applications.

4 Conclusion

In this report we investigated several representative algorithms in machine translation, and reviewed MapReduce programming framework which is proved to be efficient for many machine learning technologies, and introduced two important implementations of these technologies - Google's MapReduce library and Apache's Hadoop.

The report also visited the supporting facilities that make MapReduce framework work efficiently, including the distributed streaming data storage system and distributed structural data storage system. The implementations for both storage systems are also introduced, including Google FS and HDFS for streaming data, BigTable and HyperTable for structural data.

Considering the current clusters such as Yahoo M45 and Intel BigData, the Hadoop system and HDFS is already implemented and made good use of, whereas the structural data storage systems is not yet included in the systems. Therefore, a request for these functionality is reasonable and may enable may algorithms be port to Hadoop more easily.

References

Jeffrey Dean and Sanjay Ghemawat. 2008. *MapReduce: Simplified Data Processing on Large Clusters.* Communications of the ACM, vol. 51, no. 1 (2008), pp. 107-113

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. 2006. *Bigtable: A Distributed Storage System for Structured Data.* 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2006, pp. 205-218.

Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. 2003. *The Google File System.* Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 20-43.

Luiz Andre Barroso, Jeffrey Dean, Urs Hlzle. 2003. *Web Search for a Planet: The Google Cluster Architecture* IEEE Micro, vol. 23 (2003), pp. 22-28.

A Bialecki, M Cafarella, D Cutting, OOMalley. 2005. *Hadoop: a framework for running applications on large clusters built of commodity hardware.* Wiki at http://lucene.apache.org/hadoop.

Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, Yuanyuan Yu, Gary Bradski, Andrew Y. Ng, Kunle Olukotun 2007. *Map-Reduce for machine learning on multicore* Advances in Neural Information Processing Systems, 2007

Dhruba Borthaku 2007. *The hadoop distributed file system: Architecture and design* Wiki at http://lucene.apache.org/hadoop.