Optimizing the use of the Hard Disk in MapReduce Frameworks for Multi-core Architectures*

Tharso Ferreira, Antonio Espinosa, Juan Carlos Moure and Porfidio Hernández

Computer Architecture and Operating Systems Department, University Autonoma of Barcelona, Bellaterra, Barcelona, Spain

1 {tsouza, antonio.espinosa}@caos.uab.es
2 {juancarlos.moure, porfidio.hernandez}@uab.es

Abstract—MapReduce simplifies parallel programming, abstracting the responsibility of the programmer, such as synchronization and task management. The paradigm allows the programmer to write sequential code that is automatically parallelized. The MapReduce Frameworks developed for multi-core architectures provide large processing keys which consequently grow intermediate data structure, which in some environments causes the use of all the available main memory. Recently, with the development of MapReduce frameworks for multi-core architectures that distribute keys through the memory hierarchy, the problem of using the entire main memory, by the data generated was minimized. But in an environment where all threads access the same hard disk, certain situations may lead a competition between the threads, to take keys generated from main memory to the hard disk, thus creating a bottleneck.

Based on the behavior of threads and the growth of intermediate data structure in multi-core environments, we present an improvement of access to the hard disk in MapReduce frameworks for multi-core architectures. The main objective is to ensure that distinct threads do not compete to take the keys processed, from the memory until the hard disk.

Keywords: MapReduce, Multi-Core, Main Memory, Thread, Virtual memory.

1. Introduction

The massive use of multi-core processors in recent years has opened the possibility of creating new parallel applications. The idea of these new applications is to take advantage of the use of parallel processing offered by the existence of more than one processor or core on the same architecture. These applications force developers to manage a series of low-level details, such as thread creation, synchronization, concurrency, resource management and fault tolerance [1]. Within this context, creating a scalable, and correct parallel application has become a complex task.

MapReduce assumes that the programmer needs to express two functions to develop the application, Map and Reduce [2]. The Map function processes the input file and generates a set of intermediate pairs of key/values. The Reduce function joins these pairs through an intermediate sum or some kind of aggregation, using each key as an index, and the framework takes care of the parallelization details.

The MapReduce frameworks for multi-core architectures have been designed to work on the main memory, avoiding the need to use any type of secondary memory [3]. However, the increase in the number of keys processed, may consume the entire main memory, because of the growth of intermediate data structure. Previous studies [4] have shown the dependence that exists between the execution time and the distribution of keys in the input file. The same dependence is seen in the growth of intermediate data structures in the MapReduce framework, and therefore the use of main memory.

This situation has resulted in implementation of a MapReduce framework that distributes keys through the memory hierarchy, main memory and hard disk, preventing the entire main memory to be consumed [5]. However, we identified that in environments with an increasing number of threads, with a single hard disk, there may be competition between the threads to take the keys generated from main memory to the hard disk.

Our goal is through the administration of limits and priorities among different threads, avoid a situation where multiple threads need to take large amount of keys to the hard disk at the same time.

The objective is to evaluate the system dynamically, define a set of execution parameters, and manage intermediate data structures and access of the framework to hard disk by distinct threads. The focus in this paper is the use of applications involving a relatively large number of unique keys. A word count application represents this scenario, where there may be a large amount of unique keys.

To reach our goal, we developed and implemented an model, which evaluates the use of main memory dynamically. Based on the usage of main memory, determines the moment that a certain amount keys should be moved from main memory to the hard disk at different times by distinct threads.

The rest of this paper is organized as follows. In Section 2 we introduce the MapReduce paradigm, and present an ex-
tension to Metis that distributes keys between main memory and hard disk, the basis for this work. In Section 3 we present the main issue on which we dedicate ourselves to this work. In the section 4 we show our proposed model. In section 5 we present a summary of the environment, the benchmark application and the evaluation method implemented in this work. Finally in section 6 we present a brief conclusion of this work.

2. Related Work

There are different studies and implementations of the MapReduce model in different types of architectures. Of the model introduced by Google [2], there are implementations for clusters, such as Hadoop [6], or multi-core architectures as Phoenix [1] [3], Metis [7] and Phoenix++ [8], as well as the extension of Metis that distributes keys through the memory hierarchy [5]. This section summarizes the basic principles of the MapReduce model, and the basis for this work.

2.1 MapReduce Programming Model

MapReduce [2] is a paradigm created by Google, to support parallel processing of large data sets. The goal of this paradigm is to make easy the programmer creating parallel applications, where it is only necessary to provide a sequential implementation, expressed by two main functions, Map and Reduce.

MapReduce is able to achieve a high degree of data parallelism because it breaks workloads down into tasks that can be processed independently of each other [9]. In theory MapReduce splits the input file into M parts and sends each Map worker. Each Map worker using the functions provided by the programmer, processes their own part of the input file, generating a list of key/values pairs. When all parts end up to be processed by the Map tasks, the MapReduce framework invokes the function Reduce, which performs data reduction through a sequential implementation provided by the programmer, one task for each distinct key produced by the Map tasks. Each Reduce worker generates an output of key/value, which are usually aggregates to generate a final output.

One problem that MapReduce solves, is to take the output of the Map phase to Reduce phase through an intermediate structure, which receives the keys of Maps tasks [7].

In the multi-core architectures, the structures used are placed in main memory. The entire organization of Map output is critical to the performance of MapReduce applications, since the entire set of intermediate data need to be rearranged between the Map and Reduce phases. In short, the data produced by the Map phase are dumped in the same order they are read from the input file, while the Reduce phase the data are grouped by key. In multi-core architecture, the whole application performance is dominated by the operations performed on the intermediate structure, which is in main memory. To guarantee the performance of MapReduce applications on multi-core architectures, the system should provide enough main memory to run of the application.

2.2 Metis

Using Phoenix [3] as a base, Metis [7] has been developed as a library of MapReduce, an improvement to store intermediate data in a new data structure, to improve the performance with most types of workloads.

To reduce the amount of data stored in intermediate structure, Metis uses a combiner function, as proposed by Google the original idea of MapReduce paradigm. The combiner performs data reduction per Map thread before the Reduce phase starts, in a way to avoid all the main memory be consumed.

Metis uses as an intermediate data structure, a hash table, where each entry contains a B+tree, as shown in figure 1. The idea is to take advantage of the complexity O(1) in search of the hash table, to find the entrance of certain key, and then down in the tree using O(logN) complexity of the B+tree to find the key position. If the key already exists, the framework adds a new value to list of this key. Using the combiner function that list of occurrences is reduced to only one item, which represents the number of occurrences of a given key. If the key does not exist, then the new key is inserted into the tree. To avoid competition between different threads on the same memory region, each thread has its own Hash+tree, which is represented by each row of the matrix shown in figure 1.

All the Map workers share the same hash table size, and the same hash function, which attempts to ensure a good distribution of keys throughout the hash table.

3. The Problem

In Metis, there are two types of memory allocation. The first of these occurs in the prediction phase, when the size of the hash table is set, depending on the number of distinct
keys. This size is fixed and is not changed during use of the intermediate structure. The second memory allocation is observed in the creation of trees, which occur dynamically, depending on the number of levels and nodes.

Metis tries to keep an average of 10 distinct keys for each hash entry, however for environments that provide an execution of many threads in parallel, this number is usually lower. For example, for an input of 100 million distinct keys using the standard 10 distinct keys for hash entry, are necessary 10 million of hash entries. In an environment where there are 24 threads for example, this would create a hash table of 24 rows by 10 million of columns, or 240 million hash entries.

Assuming that the framework takes to hard disk only the key and value, where each key is a word of 20 characters and the value is an integer, each line would have a cost around 228 megabytes of data. In the worst case, depending on the memory usage, move all keys from the intermediate data structure to the hard disk in an environment of 24 threads, would cost about 5472 megabytes. Move large set of keys in an environment where all the threads can not access the hard disk at the same time, requires that the operating system serialize the access.

In short, while a thread takes their own keys to the hard disk, all other threads are waiting. With the waiting threads, no more key are processed, resulting in a cost-time.

4. Implementation

Using the extension of Metis that distributes the keys through the memory hierarchy as a base [5], we propose a solution using a model of soft and hard limit on all the threads, which move keys through the memory hierarchy, preventing threads to compete for access to the hard disk.

The main goal is to use a high limit, in this case hard limit to derive the soft limits, in this case an individual limit per thread, which must be distributed among the available threads, giving more priority to certain threads to access the hard disk.

In theory, the focus of this strategy is to allow the application decide the moment that the keys should be stored on the hard disk according to the thread, and the amount of keys that must be stored, also preventing the threads compete for access to the hard disk.

4.1 Map phase

Map phase is where the keys are are inserted in the intermediate data structure, it is essential to manage the growth of this structure, and the consumption of main memory.

As may be seen in figure 2 item 2, when a key is emitted by the Map task in the function emit_intermediate, the thread searches the entry in the hash table in which this key corresponds. When the hash entry is found the thread traverses down on the tree contained in the entry, until it finds the key position. Each new key, inserted into the hash entries, has a cost in memory usage. At the end of the Map task, an amount of keys have been generated and stored in main memory, generating main memory consumption. The memory consumed by Map Task is variable, depending on the amount and distribution of unique keys in the hash table. Knowing that the B+tree is a order 3 tree, and may be up to 7 keys per node, each time that a new key is inserted into a new node, is allocated space for seven keys. In short, if the key distribution occurs in different hash entries, the consumption of main memory will occur more rapidly, dominant situation when there is an large amount of different keys.

At the end of the Map task, the same thread executes the function verify_hard_limit and verify_soft_limit, as may be seen in figure 2 items 3 and 4. The verify_hard_limit function returns the maximum memory that can be used in the environment, i.e. the most critical memory usage. This value is the same for all threads, which serves as the basis for determination the soft limit, returned by the function verify_soft_limit. The soft limit is unique to each thread, and determines the time that each thread can start to take the keys to the hard disk, i.e., while a thread move keys between main memory and hard disk, another thread can continue processing new keys and inserting on the intermediate data structure.

If the soft limit of the running thread has been reached or exceeded, the same thread copies the keys stored in the last column of its own line of the hash table using copy_keys, as is shown in figure 2 item 5, to the buffer through buffer_keys on item 6. When the copying of keys is completed, the same thread back to check if the main memory was reduced to below the limit. If memory usage is not reduced sufficiently, the same thread performs the same operation by copying the keys from the last column, in direction to first column. The thread moves to the next task only if the memory is reduced enough or column zero is reached.

The buffer where the keys are copied has a fixed size, and just spills the keys on the hard disk through spill_keys, as is shown in figure 2 item 7, when the task is completed or the buffer is full. Each thread running generates a file on the hard disk, where it stores its keys. The goal is that at the end of Phase Map, if the memory is not enough, some of the keys are stored in the main memory through the intermediate data structure, while the other part of the key are stored on the hard disk. The idea is to take the keys that exceed the limit of main memory to the hard disk, providing space in main memory, so that the Map tasks can continue processing keys without the use of swap.

The use of a device such as the hard disk, means inserting overhead in the framework. The objective is hiding the latency of hard disk and minimize the overhead generated, through a set of proposed strategies to prevent the movement.
4.2 Evaluation of main memory available

To avoid that the amount of keys generated by the Map tasks use the entire main memory, monitor the environment and the amount of available main memory becomes essential. We created a set of functions that aims to evaluate the amount of memory available at the end of each Map task. Each thread at the end of its execution verifies that the limit set has been reached.

4.3 Hard Limit and Soft Limit

To avoid competition between different threads to take the keys to hard disk, we set two parameters to the limits, soft limits and hard limits. The goal is to treat each thread differently, so that disk access is done at different times.

The value of the hard limit, is calculated in percent on the total amount of main memory. To calculate the base of the soft limit, the running thread divides the hard limit by total number of threads. To find the corresponding soft limit, the running thread calculates \( N_t \times \text{SoftLimit} + \text{SoftLimit} \), where \( N_t \) corresponds to the number of the running thread, and \( \text{SoftLimit} \) corresponds to the base of the soft limit previously calculated. Using as an example an environment with 6 threads, where the total amount of available main memory is 24 gigabytes, and using a hard limit of 50% would result in the following limits soft shown in table 1.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Soft Limit (MB)</th>
<th>Hard Limit (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2,000</td>
<td>12,000</td>
</tr>
<tr>
<td>1</td>
<td>4,000</td>
<td>12,000</td>
</tr>
<tr>
<td>2</td>
<td>6,000</td>
<td>12,000</td>
</tr>
<tr>
<td>3</td>
<td>8,000</td>
<td>12,000</td>
</tr>
<tr>
<td>4</td>
<td>10,000</td>
<td>12,000</td>
</tr>
<tr>
<td>5</td>
<td>12,000</td>
<td>12,000</td>
</tr>
</tbody>
</table>

In small environments with few threads, the competence for access to the hard disk becomes small. In this situation the use of soft limit creates a situation of overhead. To avoid adding overhead when the number of threads is less than 4, in this case only the hard limit is checked.

4.4 Spill Buffer

Before spilling the keys on the hard disk, the thread responsible should copy the keys to a buffer. To manage these keys, we create a spill buffer, which has the same number of rows as the original hash table, i.e., the amount of threads that are running, but that does not have hash function, and simply performs sequential storage. Although the keys were removed from the intermediate data structure and taken to spill buffer are in memory, until the spill buffer is full or the task ends. Each key stored in spill buffer occupies less space as opposed to the tree. Since this is only an intermediate buffer, without search purpose, only sequential insertion, each line consists of an array that stores only keys and value. Each B+tree which contains 10 different keys, occupies about 726 bytes in the intermediate structure of Metis, while in our array the same 10 keys occupy 540 bytes. Thus freeing memory until the keys of the Spill buffer has been spilled on the hard disk.

4.5 Reduce phase

Knowing that part of keys that exceed the memory is on the hard disk, and should be brought back to the main memory to be processed by the Reduce workers, we turn to the problem of how to keep all keys in main memory. To avoid the risk of running out of main memory, the keys must be brought from the disk on demand. In this case we took advantage Reduce tasks are performed in sequence, i.e., column 0 to the N column or last column. Before the Reduce phase starts, a certain amount of keys is moved from the hard disk to fill the first group of columns which will be reduced. Until the \text{read_data_disk} \ finishes copying the keys to the first group of columns, all the threads stay blocked.

When the copy is complete, the threads are liberated to effect the reduction of keys and store them in the final buffer. To know what the limit is of columns to be reduced before brings the next sets of keys from the hard disk, we maintain a column counter. When the counter reaches the value of the next group, again all threads are blocked waiting for a copy of a new set of keys that are on the hard disk. Before starting to copy the keys from the hard disk into main memory, the thread that is running to make the copy, checks the memory usage using \text{verify_hard_limit}. If the amount of main memory used reached or exceeded the limit, the running thread makes a copy of the keys stored in the final buffer through the \text{copy_keys}, and stores it in the spill buffer with \text{buffer_keys} to be spilled back into the hard disk using \text{spill_keys}, freeing up space in the main memory.

When all columns of intermediate data structure are reduced by Reduce workers, if there are keys on the hard disk, they are all copied to the buffer for the merge phase.
through the `read_data_disk_merge`. To start the Merge phase, all the keys that were on the hard disk needs to be brought back to memory. The entire scheme of phase Reduce can be seen from figure 3.

![Fig. 3: Reduce Phase.](image)

### 4.6 Influence of key distribution

Performance optimization is directly linked to key distribution in the hash table entries, i.e., the number of keys stored in each entry when the workers begin to copy them to the hard disk. If workers begin to copy keys to the hard disk too early, it may be that find a few keys stored in the hash entries, occupying the time of the workers, however making little reduction of main memory. The decision to start taking the keys to the hard disk, depends directly on the consumption of main memory, and the memory usage limit set.

If other applications are consuming the main memory, the framework will choose to start store the keys on the hard disk in the execution of the first Map tasks. This scenario is not favorable to our optimization, since many hash entries may still be empty. Each thread executes only one copy of keys from its own line, in short, if a Map task verifies that a given entry in the hash table has no keys stored, this task proceeds to the next hash entry. If the data structure exceeds the available memory, and the framework decides spill keys on the hard disk, just will be copied keys of hash entries that have not been verified by Map tasks. So if the copies of keys, is made by the last Map task, will be copied more keys than if the copy is made by first Map tasks. In the worst case, the Map tasks will take to hard disk few keys, for having started the spill too soon. The result of this is a little reduction in memory usage, consequently the increased use of swap and more page faults.

### 4.7 Experimental Method

The measurements have been taken in two environments: (1) a multi-core processor with Intel Core Duo 3GHz and 6GB of main memory in 64-bit Linux, and (2) a dual-socket Intel(R) Xeon(R) E5645 2.4GHz with 6 cores each one, and 96GB of main memory in 64-bit Linux. We use a benchmark like Word Count, used in the original Metis.

For environment with 2 cores, we set a limit of memory usage by 70%, because it is a small environment with only 6GB of memory, and this limit would be reached quickly, being possible check for page faults. While the environment of 12 cores, the limit was set at 30%, 12% and 10%, to be a relatively larger environment, with 96GB of main memory, and the inputs used are the same used in the environment of 2 cores. The idea of the experiments in the second environment, is see how much overhead is created by hard disk access, besides the advantage of using the strategy of soft limit and hard limit.

As dataset for this evaluation, we use a key distribution, where all the input files have different keys, which can be seen in Table 2.

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Keys</th>
<th>Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>228MB</td>
<td>10,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>457MB</td>
<td>20,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>686MB</td>
<td>30,000,000</td>
<td>30,000,000</td>
</tr>
<tr>
<td>915MB</td>
<td>40,000,000</td>
<td>40,000,000</td>
</tr>
<tr>
<td>1100MB</td>
<td>50,000,000</td>
<td>50,000,000</td>
</tr>
<tr>
<td>1373MB</td>
<td>60,000,000</td>
<td>60,000,000</td>
</tr>
</tbody>
</table>

Working on the problem described in section 3, we show in Figure 4 and 5, the reduction in the use of main memory, obtained in the Map and Reduce phases for the optimized version. To the input files of 10, 20 and 30 million keys, it can be seen that there is no change from the original, because for these types of workloads, the memory limit is not reached. As the memory consumption does not exceed the limit, there is no change on the way that the MapReduce framework works.

As seen in Figure 4 and 5, for input files of 40, 50 and 60 million of keys, memory usage is exceeded, the framework this way makes the decision to make use of our optimization.

As can be seen in figure 4, to the inputs of 40, 50 and 60 millions of keys, it is possible to note an increase at execution time Map phase observing the columns of optimized version against the main memory reduction observed by line corresponding. This increase in execution time is justified by the spill of keys on the hard disk, each Map task has responsibility to manage it own keys. While the keys do not end up to be taken to the hard disk, the Map
task does not resume emitting new keys in main memory. Each keyset brought to hard disk allows the release of new memory space, preventing the main memory to be fully consumed, this would force the system to use the swap, increasing the number of page faults. Also in figure 5 we can observe the same behavior as figure 4, but with a smaller reduction in main memory. This situation is justified by the fact that the Reduce phase needs more data in memory to make the reduction. Unlike the Map phase where the main work consists of producing and storing keys in an intermediate data structure, Reduce phase the data must be read from an intermediate structure, processed and stored in a second structure, forcing the framework to hold more keys a time into memory.

Although the Reduce phase is normally faster than the Map phase, becomes difficult to avoid the increase in execution time, because of the constant interaction with the hard disk. First for keys that have been stored by the Map phase in the hard disk, and must be brought back to main memory for processing. And second, by taking the reduced keys back to the hard disk if the memory is close to the limit, i.e., double of interactions with the hard disk that Map phase

The system uses the swap as an extension of memory, which normally causes the pages faults. In contrast, we define in this first model, the need to reduce these pages faults, since the constant interaction of the system with the hard disk becomes costly to the execution time. Sending a set of keys to the hard disk before the main memory to be consumed, prevents the system to need use the swap, this way it is possible to decrease the page faults, also reducing the constant and disordered hard disk access. In Figures 6 and 7 is possible to see the decrease of page faults, both in the Map and the Reduce phase. Take the keys to the hard disk in groups, before the memory is fully consumed, reduces the influence of latency of access to hard disk at execution time, effect that can be observed by the use of swap.

Fig. 4: Execution time and memory usage on Map phase.

![Map Phase](image)

Fig. 6: Map phase major page faults.

![Reduce Phase](image)

Fig. 7: Reduce phase major page faults.

In the figure 8 it is possible see that for execution with a limit of 10%, where only hard limit is used, there is a significant increase in runtime. This situation is result in the use of the 24 threads available to take large amounts of keys
5. Conclusions

Addition to the original implementation where we promote reduction in the use of main memory, and reduction of page faults, we also show in this paper a better way to access the hard disk by the MapReduce framework, using different limits for the threads. The parameters limit, buffer size and number of hash entries taken to hard disk, open a new line of research where it can further improve the execution time of the implementation presented in this paper. This implementation opens the way for improving the MapReduce framework on multi-core architectures, allowing the framework to adapt to different workloads.

As an extension, our implementation does not predict solve all problems for the use of large data sets in systems with limited memory, but rather open up a straight of research for future solutions.

Acknowledgements

This work was supported by the Ministry of Education of Spain under contract TIN2011-28689-c02-01.