International Conference on Computational Science, ICCS 2013

An Optimization for MapReduce Frameworks in Multi-core Architectures

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

Computer Architecture and Operating System Department, Universitat Autònoma of Barcelona, 08193 Bellaterra, Barcelona, Spain

Abstract

MapReduce simplifies parallel programming, abstracting the programmer responsibilities as synchronization and task management. The paradigm allows the programmer to write sequential code which is automatically parallelized. The MapReduce frameworks developed today are designed for situations where all keys generated by the Map phase must fit into main memory. However certain types of workload have a distribution of keys that provoke a growth of intermediate data structures, exceeding the amount of available main memory. Based on the behavior of MapReduce frameworks in multi-core architectures for these types of workload, we promote an extension of the original strategy of MapReduce for multi-core architectures. We present an extension in memory hierarchy, hard disk and main memory, which has as objective to reduce the use of main memory, as well as reducing the page faults, caused by the use of swap. The main goal of our extension is to ensure an acceptable performance of MapReduce, when intermediate data structures do not fit in main memory and it is necessary to make use of a secondary memory.

Keywords: MapReduce, Multicore Processors, Threads, Main Memory, Virtual Memory

1. Introduction

The popularization of multi-core processors in recent years has opened the possibility of creating new parallel applications. These applications force developers to manage a series of low-level details, such as thread creation, synchronization, concurrency, resource management and fault tolerance [1]. MapReduce assumes that the programmer need only express two functions, Map and Reduce, while the framework takes care of the parallelization details [2]. A MapReduce Framework for multi-core architectures is able to take advantage of rapid communication between tasks through shared memory [3]. However, we observed degradation in the performance of MapReduce applications when the number of distinct keys is on the order of millions. In Metis [4], for an input of 100 million distinct keys using the standard distribution of 10 distinct keys per hash entry, 10 million hash entries are needed. With 24 threads in the same shared-memory system, a hash table of 24 rows by 10 million of columns, or 240 million hash entries would be necessary, for a total 5.625 megabytes of main memory. With a good distribution of keys, and a B+tree order 3, that can store a maximum of 7 distinct keys per node, only one node per tree is necessary. The entire structure, hash table with B+tree, would need approximately 39.375 megabytes. In this scenario, the intermediate data structure may exceed the amount of main memory available.

*Corresponding author. Tel.: +34-93-581-1990; fax: +34-93-581-2478.
E-mail address: {tsouza, antonio.espinosa}@caos.uab.es, {juancarlos.moure, porfidio.hernandez}@uab.es.
We propose a solution to this problem when the processed data set is higher than the amount of main memory available, making the framework take advantage of the memory hierarchy, main memory and hard disk. Using Metis [4] as a basis for development, we show that even using a high-latency device, like hard disk, it is possible to get acceptable performance through the key distribution through the memory hierarchy, i.e., moving keys between main memory and hard disk, preventing the main memory be completely consumed.

The rest of this paper is organized as follows. In Section 2, we present a brief summary of related work. In Section 3 we promote a description of our implementation. In Section 4 describes the experimental method implemented in this work. Finally in section 5 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. From the model presented by Google [2], there have been implementations for clusters, such as Hadoop [5] or multi-core shared memory systems like Phoenix [6], Metis [4] and Phoenix++ [7].

3. Implementation

In the Map phase, our strategy start by identifying the amount of main memory. The goal is to adjust memory usage, front of the limitations of the system or user settings. 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, and the amount of keys that must be stored. When a key is emitted by the Map task as can be seen in Figure 1 items 1 and 2, the thread searches the entry in the hash table. When the hash entry is found the thread down on the tree contained in the entry, until find the key position. 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. At the end of the Map phase, the same thread that produced the keys, checks if the memory has reached the limit. If the limit has been reached or exceeded, the same thread copies the keys stored in the last column of its own line on the hash table to the auxiliary buffer, as can be seen in Figure 1 item 3. 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. Since this is only an intermediate buffer, with the purpose of just inserting sequentially, each line consists of an array that stores only key and value. When the copying of keys is completed, if memory usage is not reduced sufficiently, the same thread back to copy 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 when the task is completed or the buffer is full, as can see in item 4. Each thread running generates a file on the hard disk, where it stores its keys.

In Reduce phase the main point is quickly retrieve keys that are on the hard disk, to be reduced with keys that are in main memory. In Reduce phase, the framework must ensure that the limit of the main memory in use will not be exceeded, moving keys between main memory and hard disk when necessary.

Knowing that part of keys that exceed the memory is on the hard drive, and should be brought back to the main memory to be processed by the Reduce workers. To avoid the risk of main memory runs out, the keys must be brought from the hard disk on demand. 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, as can be seen in items 5 and 6.

When the Reduce workers finalize to process the first group of columns, it checks if the memory usage has reached or exceeded the limit set. If the limit has been reached, the worker copies the keys of the first group of columns already reduced to spill buffer, as can be seen in item 8. When the spill buffer is full, the keys are spilled to the hard disk, as shown in item 9, and freeing up memory space. After all columns are processed by Reduce tasks, and before starting the Merge phase, the thread zero retrieves all keys stored on hard disk back to memory, so they can be sorted and joined by merge workers, as is shown in items 10 and 11 of Figure 1. To start the Merge phase, all the keys that were on the hard disk needs to be brought back to memory.
4. 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. As dataset for this evaluation, we use a key distribution of unique keys, with an amount between 10 and 60 million keys.

We show in Figures 2 and 3, the reduction in the use of main memory, obtained in the Map and Reduce phases for the optimized version. As can be seen in Figures 2 and 3, for input files 40, 50 and 60 millions of keys, the memory usage is exceeded, thereby making the framework make use of our extension. In these input, it is possible to note an increase at execution time in Map phase. This increase in execution time is justified by locks made to the threads, with the goal of making the spill on the hard disk of buffered keys.

Each keyset brought to hard disk allows release new memory space, preventing the main memory to be fully consumed. Also in Figure 3 we can observe the same behavior as Figure 2, but with a smaller reduction in main memory. This situation is justified by the fact that the Reduce phases need more keys 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, the Reduce phase data must be read from an intermediate structure, processed and stored in a second structure. Although the Reduce phase is normally faster than the Map phase, it becomes difficult to avoid the increase in execution time, because of the constant interaction with the hard disk. First, keys that have been stored by the Map phase in the hard disk, must be brought back to main memory for processing. And second, reduced keys must be taken back to the hard disk if the memory keeps close to the limit.

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 4 and 5 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.

5. Conclusions

This paper shows an extension of the Metis framework, which distributes keys through the memory hierarchy, main memory and hard disk. We present a first implementation of our model, where it is possible to see a reduction in memory usage and page faults, although there is an addition at execution time. We conclude that our optimization requires a smaller amount of disk accesses than would be necessary by simply using the virtual memory mechanism. This first implementation opens the way for improving the MapReduce framework on multicore architectures, allowing the framework to adapt to different workloads.

Acknowledgements

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

References