I/O Efficient Implementation of MapReduce

CIS5930, Computer Science Department, FSU

Background

MapReduce is a programming model and an associated implementation used by Google for processing their massive data sets. It has a simple yet powerful interface that is amenable to a broad variety of problems. Since 2003, when the MapReduce framework was first created, more than ten thousand distinct programs have been implemented under this model. A large number of MapReduce tasks are now running on Googles clusters at any minute, processing huge amounts of data and gathering lots of useful information. For instance, in the single month of September 2007, more than 2 million MapReduce jobs have been completed, processing over 400,000 TB of input data [2]. The success of MapReduce stems from the fact that it is very easy for the programmer to write a simple program and run it effi-ciently on a thousand machines, greatly improving the engineers productivity.

In this project, (since we dont have a thousand machines,) we will study the problem of how to implement the MapReduce interface efficiently on a single machine. Since the data size could be much larger than memory, I/O-efficient techniques that you have learned from class will be useful (and required!).

The MapReduce interface

We are going to use a simplified interface of MapReduce, which in fact allows for a more efficient implementation and also makes the problem more interesting. The input is simply a sequence of elements, stored in a file on disk. The user specifies the computation through two functions: map and reduce.

Map takes an input element and produces a set of intermediate (*key, value*) pairs. The MapReduce library groups together all intermediate (*key, value*) pairs with the same key and passes them to the reduce function.

Reduce accepts an intermediate key and a set of values for that key. It combines these values to form a possibly smaller set of values and write them to an output file.

In this project, we make the following simplifying assumptions: (a) the map function accepts one input element and produces only one intermediate (key, value) pair; (b) the reduce function combines all values associated with any particular key into one value using the same operator $\Diamond$; and (c) $\Diamond$ is commutative and associative. Thus, the *reduce* function simply accepts two input values x and y, and then returns $x \Diamond y$. Furthermore, assumption (c) implies that the *reduce* function can be called in any order to combine all the values associated with the same key.

Finally, we assume that the types (specified by the user) for the elements, keys, and values all have fixed sizes, e.g., double, int, or class consisting of these fixed-size types.

Example: Counting frequencies of keywords

Suppose we have a large web query log storing the keyword for each query ever received. Assume for simplicity that each query only has one keyword, and all keywords are of the same length, say 8 charac-ters. One basic task in query log analysis is to count the number of occurrences of each keyword. The user could then write MapReduce functions as follows. The *map* function takes in a keyword and

output (keyword, 1) as the (key, value) pair; the *reduce* function then simply defines the operator to be addi-tion.

Task 1: Design the MapReduce interface

You should design your MapReduce interface in a way that allows flexible user defined types for the input elements, intermediate keys, and values. (Hint: Use C++ templates).

You will use the TPIE library for your implementation. TPIE defines the type "stream", which shall be used for all the input, output, and intermediate data. See the course web page (under Programming Notes) for more information about TPIE.

Your library should have three entry points: (1) The user can do only map, in which case the user will provide a stream of input elements and his/her *map* function. The user also passes the pointer to an empty stream to receive the intermediate (key, value) pairs. (2) The user can do only *reduce*, in which case the user will provide a stream of (key, value) pairs and his/her *reduce* function. The user also passes the pointer to an empty stream to receive the output (key, value) pairs. This will also be useful in Task 5. (3) The user can supply both *map* and *reduce* and do the entire computation via one call. In this case the user only gives the input stream and a pointer to an empty stream to receive the final re-sults. Intermediate data should be discarded.

Clearly specify your user interface in C++. Any user should be able to use your library by reading only the .h file that you provide.

Hint: There are a number of ways for the user to provide userdefined functions (the *map* and *reduce* functions in our case). You can use function pointers (the C-style), or ask the user to pass an object of a class that has map and reduce as its member functions (the C++-style). The latter is more efficient because it allows the functions to be inlined.

Task 2: A simple implementation

The easiest way to implement the MapReduce framework is scan + sort + scan: You first scan through the input stream, calling the *map* function for each input element, and then write the (key, value) pairs to an intermediate stream. Then sort this stream by the keys, grouping all pairs with the same key together. Finally, scan through the sorted stream again. Now all pairs sharing the same key are consecutive, so you can simply call the *reduce* function to merge their values together. The final output is a sequence of (key, value) pairs, one for each distinct key.

Your task is to implement this simple solution, and then use it to do the keyword-counting job. Let K be the total number of distinct keywords. Try generating a few data sets with various K and different distributions (from uniform to highly skewed). What do you observe from these experiments? Does the performance of your implementation fluctuate? A good model to generate skewed distributions of keywords is to follow the Zipfs Law [1]. Note that the size of your data set should be larger than the memory size (TPIE allows you to specify the amount of memory available to your program).

You will use the TPIE library for your implementation. TPIE already provides scan and sort primitives, so you dont need to reimplement them. See the course web page for more information about TPIE.

Task 3: Improved implementation

Real world data sets are typically not uniform: They could be highly skewed (a few keywords are very popular), and there could be high locality in the order they appear in the log (some keywords may be very popular in a certain period of time). Try to improve the simple implementation above by taking

advantage of these data characteristics. Experimentally compare your improved implementation with the previous one on the data sets you have generated, and with different memory sizes.

Hint 1: You need to look into the merge sort algorithm and modify it accordingly.

Hint 2: You dont need to make your implementation work for >1TB of data (think about the discussion on the practical implementation of external heaps in class).

Task 4: Analysis of the improved implementation

Analyze the worst-case I/O cost of your improved implementation, given that the i-th keyword appears N_i times, for $i = 1, \cdots, K$. Express your bound in terms of N (the total number of keywords), K (the number of distinct keywords), $N_1, \cdots, N_K$, and model parameters M and B. Does your solution for Task 3 asymptotically improve upon the one in Task 2? If so, in what cases is it an improvement and in what cases is it not?

Task 5: Finding the most popular keywords

Now we have the frequencies for all the keywords from Task 2. A popular index structure used by search engines is the inverted list, which for each keyword, stores the list of web pages (page ids) that contain the keyword. The present task is, for each page, find the most popular keyword in it, i.e., the one with the highest overall frequency that you computed from Task 2. Use your MapReduce library to solve the problem, and implement your solution. Generate some data yourself and test your program. What if the keywords in the inverted list are stored in a different order than that in the (keyword, fre-quency) list?

Hint: You need to construct the intermediate (key, value) pairs yourself, and then only call the reduce part of your library. See Task 1.

Instructions

You should complete this project by yourself. Before the due date you should submit two items. The first will be the C++ code of your MapReduce library and the programs calling the library for Task 2 and Task 5. The second will be a written report. The report should consist of the following contents:

- Problem Definition, which in our case will be the specification of your MapReduce interface, the input/output format, etc.;
- A Simple Solution, which will be the description of your implementation for Task 2;
- Improved Solutions, which will be the description of your improvements for Task 3 and its theoretical analysis for Task 4;
- Applications, which will describe how to use your library to perform the keyword-counting job and the "popular keywords" job in Task 5;
- Experiments, which will describe your experimental findings, including how you have generated the test data, what you have observed from the experiments, and explanations for the experimental results;
- Conclusions.

In short, you should write the report as if you were writing a research paper. You dont need to follow the exact outline above, but the report should include all the contents and be selfcontained. If you obtain ideas and help from other sources (classmates, the Internet, the TA, etc.), **you must give proper acknowledgements, citations, and references, otherwise it will be treated as plagiarism**!

You shall submit your code and report (PDF file produced by LaTex) via email as attachments. In addition, a printed copy of your report shall be handed in on the due day.

When you program and debug, you can use any Linux machine and install TPIE on it. When you want to test your code on large data sets, you can either use your own machine (if you have Linux installed), or use the Linux servers in our department (those dedicated for programming purposes). Please coordinate with other teams so that no more than two experiments shall be run at the same time on the same machine. The experiments are going to take some time, so dont wait until the last minute!

References

[1] Zipf's law. http://en.wikipedia.org/wiki/Zipf's_law.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.