Grid Computing: Simple MapReduce for Image Matching on Peerster

Kush Patel

12/9/2013

Advisor: Bryan Ford

CPSC 426 Final Report
Computer Science Dept.
Yale University

1. Introduction

Grid computing is the utilization of multiple computational resources that may be distributed over a large area geographically to solve a single task. MapReduce is a programming model for processing large data sets in parallel using distributed algorithms on a cluster. This project aimed at extending the implementation of Peerster to perform distributed computing using a simple version of the MapReduce model. Specifically, Peerster was extended to perform distributed image matching.

In this implementation, the node that originated a computation acted as the master. The master node assigned work to all its neighbors who acted as workers. All the workers reported the results of their assigned computations directly to the master who gathered all the results and reported the final result to the user.

This project aimed at building a simple distributed computing system to avoid complexities in code. However, the system can further be enhanced by allowing each worker to further delegate tasks to its neighbors for a more efficient use of the distributed resources.

2. Methodology

2.1 Image Matching Startup

The user opens up a Peerster session and selects two images to match with each other. Note that the program requires that the two images be of the same size (both images x by y pixels) for successfully matching the two images. The distributed image matching process is started by pressing the “Match Images” button. The program also requires that there should be at least one peer online for the computation to succeed.

2.2 Map phase

The master node (the originator of the computation) first reads in the two images from the user provided paths into QImage data structures. The two images are then flattened into one-dimensional arrays of unsigned integers to make organizational logic easier. The next step involves chopping up the one-dimensional arrays into 2 kilobyte blocks contained in a pair of QVectors. The blocks are then stored in pairs in a list. Pairing is required to identify corresponding blocks of the two images. The list of image block pairs acts as a hash table for constant time block retrieval with its keys implicitly being the list index.

These implicit keys numbered from 0 through N-1 (where N is the total number of image block pairs) are divided equally among all the neighbors of the master node. Each of the neighbors is thus responsible for processing image blocks whose indices fall within the neighbor’s assigned range of keys.

2.2 Distributed image processing

Distributed image matching is started by sending all the neighbors their first image block pair to process. Thereafter, image block pairs are sent sequentially to each neighbor that reports its result to a previous block pair. The master sends the image block pairs in messages with an index number (ChunkID = index of the image block pair in the implicit hash table of blocks). On receipt of an image block pair message, a worker generates the block's result by computing the total RGB pixel-wise difference for all the pixels within the image block. The workers report back the block-wise image matching results in messages containing the corresponding index numbers for image block pairs.

2.3 Reduce phase

The master keeps another table of results that stores each block-wise result in the corresponding slot as per the block pair index. On receipt of each result message, the master checks if the results table has been filled. A full results table signals the end of the distributed computation. When the result table is full, the master gathers the block-wise results into one final result by adding up all the block-wise results. This final value is then weighted over the original image dimensions to compute the percentage similarity between the two images that is then reported to the user.

2.3 Simple load balancing

A simple scheme of load balancing was implemented as follows. In the event of sending the next image block pair to a worker who just reported a result, the master might discover that the worker has exhausted its entire load. The master then examines the load for each of the other workers and finds one that still has more than half the load it started off with. The entire load of this slow neighbor is then reassigned to the worker who had finished processing its load.

This simple scheme of load balancing ensures that the process does not get stuck due to peers who are in the master's neighbors list but are not online. However, it does not deal with the problem of a peer going offline in the middle of the computation and having processed more than half of its assigned load.

2.4 Error checking

The special case of no peer available for distributed computing is solved by using a timer. The timer automatically fires after a predetermined timeout value if no result message is received within this time and the computation is not yet done. This is to avoid having the user wait forever for a computation that will not occur due to the lack of workers.

3. Test Run

To run the distributed image processing kit with Peerster, follow these steps:

1. Open up multiple Peerster instances on the same zoo machine or different ones using either QtCreator or Terminal. (Note: If using different zoo machines, make sure that the nodes are connected via adding peers function designed in previous labs).
2. Once the network topology is setup, the system is ready to match any two images provided with the extension jpg, png or xpm. Select images 1 and 2 by clicking the appropriate buttons provided in the Peerster session UI and then click on “Match Images” button to start image matching. Note that the two images selected should have matching x and y dimensions for the matching process to succeed. See test_images directory for test images.
3. The results of image matching are displayed on the UI to the user in terms of percentage of similarity between the two images matched.