CS2020: Data Structures and Algorithms (Accelerated)

Problem Set 3.14159265

Due: Wednesday February 11, 11:59pm

Overview. One of the amazing changes over the last five years has been the ease with which large computations can be outsourced to the cloud. Whether you use the **Amazon Elastic Compute Cloud** or **Microsoft Azure** or the **Baidu Cloud**, it is now amazingly easy (and cheap) to analyze massive amounts of data all remotely stored and processed on the cloud.

In this problem set, we will develop a simple Cloud Computing sorting application. Attached to this problem set (in `CloudManager.jar`) are two library classes: `CloudManager` and `CloudServer` which you will use to manage your cloud computing resources.

The first problem will focus on the actual problem of sorting. We will assume that you are sorting a very, very large dataset which is stored entirely on the cloud. The dataset is so large that no one server can efficiently access the entire dataset. Instead, the dataset is broken up into smaller **pages**, and each server can access up to two pages at a time. Your first task is to write an efficient sorting function that uses the cloud servers to produce a sorted output.

The second problem you will face is determining whether the cloud provider fulfilled its contract: is the resulting list actually sorted? Be careful—do not pay the cloud provider until you have verified that the servers did the work correctly! You will develop an algorithm for checking very quickly whether a very large data set is sorted.

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 1. CloudSort

In this problem set, you will develop a (simple) cloud sorting routine. In Part (a) of the question, you will describe your algorithm and show that it is correct. In Part (b) of the question, you will implement it using the attached cloud computing libraries. In Part (c) of the question, you will think about the performance and consider some improvements.

Cloud Computing Libraries. Attached to this problem set, you will find `CloudManager.jar`, a simple cloud computing library. The library consists of two classes: `CloudManager` and `CloudServer`. The CloudManager implements the `ICloudManager` interface.

You will manage all your sorting via the CloudManager. (The CloudServer operates behind the scenes on the cloud—you do not need to modify or use it in any way.) When you startup the CloudManager, the first think you will need to do is initialize it:

> `CloudManager(CloudProvider provider, int pageSize)`: to construct a CloudManager, pass it a provider name and the page size. The page size is a limit on how much data each server can handle.
>
> `CloudProvider` is an enum (i.e., enumerated type) that currently has three options: AmazonEC2, MicrosoftAzure, BaiduCloud. If you want to use Amazon EC2, specify it as: `ICloudManager.CloudProvider.AmazonEC2`.
>
> `initializeCloud(String fileIn, int numServers)`.
> This method takes three parameters: (i) the cloud resource locator of the data to sort, in this case, a filename; (ii) the cloud resource locator where the output should be stored, in this case, a filename; (iii) the number of cloud servers on which to execute the computation. The CloudManager will return `true` if the server is correctly initialized, and `false` if there is an error.

Each cloud server has a maximum capacity, i.e., the maximum number of records that it can process in one "phase." (Phases are discussed further later.) Thus the data is divided up into pages, where each page has size specified by the `pageSize`. If you imagine that the data to be sorted is stored in an array A, the pages are organized consecutively: page 0 is $A[0\,..\,\texttt{pageSize} - 1]$, page 2 is $A[\textit{textttpageSize}\,..\,2 \cdot \texttt{pageSize} - 1]$, etc. In general, page j starts with item $A[(j)\texttt{pageSize}]$ and ends with $A[(j+1) \cdot \texttt{pageSize}] - 1$.

When you initialize the cloud server, it automatically divides the dataset into pages. You can access the number of pages, the number of elements, and a specific element in the dataset via calls to:

> `numPages()`. Returns the number of pages in the dataset.
>
> `numElements()`. Returns the number of elements in the dataset.
>
> `getElement(int j)`. Returns a single specific element from the dataset.

When using the CloudSorters, You must always specify a valid page number. (Notice that page numbers start with 0.) The CloudManager also contains a secret method `isSorted()` which may be useful for debugging.

Work on the cloud is done in **phases**. In each phase, you can schedule a large number of cloud servers to do some useful work for you. (Remember, when you initialized the CloudManager, you specified the number of severs that you wanted to reserve.) You can schedule some sorting work via the following call to the CloudManager:

> `scheduleSort(int serverID, int pageOne, int pageTwo)`. Schedules the specified server to sort the two specified pages. (ServerIDs start with 0.) During the sort operation, the two pages of data will be copied into the main memory of the server, sorted, and then copied back into the cloud. You may assume that the sorting is done efficiently.

There is one important rule to scheduling work: in a given phase, each page can be accessed by only one server. (Otherwise, there could be all sorts of problems.) Moreover, each server can only be scheduled *once* per phase.

Once you have scheduled work to be done, you can execute the work by calling the CloudManager:

> `executePhase()`. Executes all the work that has been scheduled for the current phase.

When you execute a phase, all the servers that were scheduled for that phase perform their work at the same time, i.e., in parallel. Since all the work happens in parallel, and since the phase length is fixed by the cloud provider, our main concern is minimizing the *number of phases*. To determine how many phases the CloudManager has executed (and hence how much you will have to pay for your data analysis), you can call the CloudManager:

> `getStatus()`. Returns the status of the CloudManager, including how many servers have been started, how many phases have been executed, and the total approximate cost.

When you are done with your computation, in order to shutdown the Cloud Server, call:

> `shutDown()`. Shuts down the cloud servers and calculates your total cost.

See the `ICloudManager` interface for more documentation on how to use the cloud services.

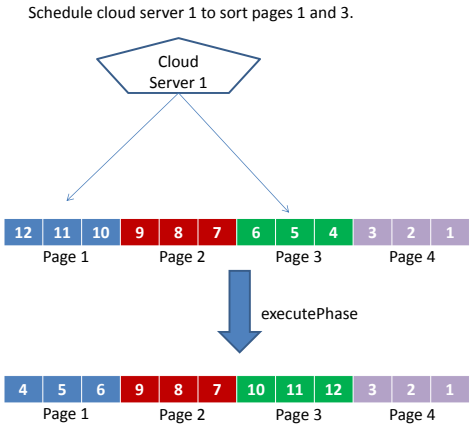

Figure 1: Cloud Server 1 is scheduled to sort pages 1 and 3. On executing the phase, the two pages are sorted together (as if they were part of a single array) yielding the outcome below.

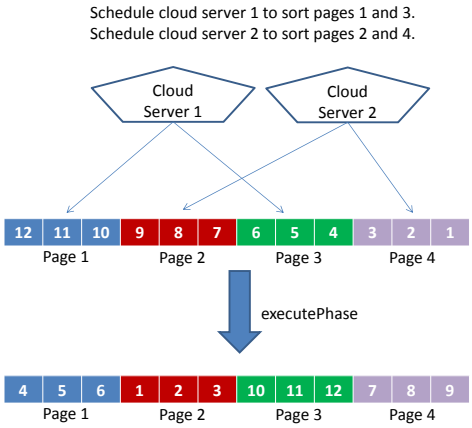

Figure 2: Here we use two cloud servers to accomplish more in a single phase. Cloud Server 1 is scheduled to sort pages 1 and 3, while Cloud Server 2 is scheduled to sort pages 2 and 4. On executing the phase, each cloud server sorts the two specified pages, yielding the outcome below. Note that we accomplish much more in the same amount of time: one phase.

Problem 1.a. Design a (simple) algorithm for performing a CloudSort. Give pseudocode to describe your algorithm. Explain why your algorithm is correct, and explain how many phases your algorithm takes (and why). Your algorithm should work for any number of servers, whether 1 or more. (You cannot, however, sort data that consists of less than two pages.) Your submission for this part should consist of a file containing the pseudocode and the requested explanations.

Consider the case where there are $2k$ pages in your dataset, and that you use k cloud servers. (Since each server can handle 2 pages, this is the maximum number of servers you can usefully use.) In this case, your algorithm should run in $O(k)$ phases. (It should be even simpler to find an algorithm that you can show works in $O(k^2)$ phases.)

Hint: To explain why your algorithm is correct, you may want to consider an invariant of the following form: For a page j and for t where $j < k - t$, after phase $2j + 4t$, all the elements that belong in pages $\geq (k - t)$ are no longer in the first $[1..j]$ pages. Depending on your algorithm, the constants in the invariant may be different.

Problem 1.b. Implement your algorithm using the Cloud Manager libraries. Your implementation should contain a method:

```
void CloudSort(String inFile, String outFile, int numServers)
```

This method should correctly sort the data in the specified file, saving it in the specified location, using (at most) the specified number of servers.

Problem 1.c. *(Extra credit)* Can you think of any simple ways to improve the running time of the sorting algorithm? Assume that you are allowed to modify the CloudServer—but that each cloud server can still only access two pages in each phase.

One area you might think about is the required phase length. Currently, if a page contains m elements, then each phase takes $O(mlogm)$ phases. Can we use shorter phases?

It *may* also be possible to reduce the number of phases, but this will require a much more complicated algorithm, if it is possible at all within the current constraints[1].

[1]Can you use ideas from ShellSort? Or maybe ideas related to Sorting Networks, e.g., Batcher's Bitonic sort? I do not know.

Problem 2. Testing CloudSort

How do you test a CloudSort? What if there is a bug in the sorting code? Or, even worse, what if the cloud provider is cheating and not performing the promised service? Perhaps the cloud provider is trying to save money by doing less work, assuming that you will not be able to detect the fact that the list is not sorted. Unfortunately, the dataset is very large (and lives on the cloud), and hence it is expensive to verify whether or not it is correctly sorted. How can you catch Dr. Evil if he lives on the cloud? Your goal in this question is to develop an efficient routine that tests whether your data is sorted.

In order to improve efficiency, we are going to give up on a 100% guarantee. Instead, we are going to ask for the following: we want to know, with 99% accuracy, with the data is at least 99% sorted.

What does it mean for a list to be 99% sorted? We say that a list of n elements is p-sorted (for $0 < p \leq 1$) if we can create a sorted list simply by deleting some set of $(1-p)n$ elements from the list. That is, the list is close to sorted in the sense that only a small number of elements are out of place.

For example, the list $[1, 4, 2, 6, 8, 7, 10, 12, 14, 20]$ is 80%-sorted since we can delete $\{2, 7\}$ to produce a sorted list $[1, 4, 6, 8, 10, 12, 14, 20]$.

Our goal is to find an $O(\log n)$ time procedure for ensuring that a list is 99% sorted with 99% accuracy. That is, we want our test to have the following guarantees:

- If the list is properly sorted, then 100% of the time, it returns the correct answer: true.
- If the list is not properly sorted, then 99% of the time, it returns the correct answer: false.

We first begin with a simple probability calculation. We then proceed to develop and analyze the algorithm. Finally, you will implement the algorithm using the CloudManager framework. For Parts (a), (b), and (c), submit the requested explanations. For Part (d), submit the specified code.

Problem 2.a. Imagine you are given a bag of n balls. At least 1% of the balls are blue, and no more than 99% of the balls are red. You randomly choose k balls from the bag, one at a time, replacing each ball in the bag after you look at it. For what value of k is it true that you will see a blue ball with probability at least 99%?

Problem 2.b. Assume you are given an unsorted array A, and you perform a "binary search" on this array. (Notice that it is very strange to perform a binary search on an unsorted list. There is no reason to believe it will find anything useful.) That is, you execute the following algorithm:

```
BinarySearch(A, key, left, right)
    if (left == right) then return left;
    else {
        mid = ceiling((left+right)/2)
        if (key < A[mid]) then return BinarySearch(A, key, left, mid-1);
        else return BinarySearch(A, key, mid, right);
```

Assume that you have two keys k_1 and k_2. And assume:

- Binary search for k_1 returns slot s_1 (even though the array is not sorted).
- Binary search for k_2 returns slot s_2 (even though the array is not sorted).

Explain why the following is true: if $s_1 < s_2$, then $k_1 < k_2$.

Hint: Draw a picture. Think about what happens during a search. Think about why this is obviously true if the array A were sorted.

Problem 2.c. Now consider the following procedure for determining if your array is sorted:

```
IsSorted(A, k)
    for (r = 1 to k) do
        i = Random(1, A.length);
        j = BinarySearch(A, A[i], 1, A.length);
        if (i != j) then return false;
    return true;
```

In this procedure, we randomly choose k items in the array A. For each item, we perform a binary search for that item. If the binary search ever fails, then we return false, i.e., the test has failed. Otherwise we return true. Explain why for the proper value of k (derived in Part (a) above), this procedure guarantees the desired accuracy:

- If the list is properly sorted, then 100% of the time, it returns the correct answer: true.
- If the list is not properly sorted, then 99% of the time, it returns the correct answer: false.

Hint: think about what would it would imply if the BinarySearch succeeded for more than 99% *of the elements in the array. Use the result from Part (b) above.*

Problem 2.d. Implement the IsSorted test in the CloudSort framework. Run your test on the different examples with different numbers of sorting mistakes. You might be interested to look at the number of data accesses required by your test. We have included several different sample files you can experiment with.