Lab9

Lab 9

EE 599 Lab9
Parallel Programming & Data Streaming
Spring 2019 Nazarian

Assigned: Thursday Mar. 7
Part 1 Due: Saturday Mar.30 11:59pm
Part 2 Due: Sunday Mar.31 11:59pm
Late submissions will be accepted for two days after the deadline with a maximum submissions between 12am and 1am: 2% penalty, between 1 and 2am: 4%, 2-3am: 8% accepted Apr.1/Apr.2, after 11:59pm.

Notes:

This assignment is based on individual work. No collaboration is allowed (Discussing the technical details of each problem with other students before submitting your work, copying from students of current or previous semesters is not permitted in our lab and HW assignments). You may post your questions on the discussion forums and/or use the office hours. We may pick some students in random to demonstrate their design and simulations. Please refer to the syllabus and the first lecture regarding the USC and our policies, including the penalties for any violation. If you have any doubts about what is allowed or prohibited in this course, please contact the instructor.

What You Will Practice

In this lab, there are two parts, one is related to parallel programming and the other is about data streaming.

Part I - Parallel Programming

Part I - Parallel Programming

In this part, you will learn the basic steps of converting slow sequential programs into fast parallel program utilizing the multi-processing capabilities in the underlying hardware. Besides, you will practice how to

implement multithreading with POSIX Threads (or Pthreads) and how to handle multi-process synchronization problem.

Assume that there are several burger makers in a restaurant, their job is to organize ingredients and make burgers. In this part you will write a code to simulate their work. Each burger maker can be seen as a thread and each ingredient is given as a string of format k[ingredient string]. k is an integer number representing the amount of a certain ingredient. The ingredient string inside the square brackets represents the name of an raw ingredient, such as “c” for cheese, “ba” for bacon, etc., or a combined ingredient such as 1[a]2b, 3[2[bc]] etc.

For example,

- 3[a]means there are 3 pieces of a raw ingredient of name “a”
- 4[bw] means there are 4 pieces of a raw ingredient of name “bw”
- 3[1a2[b]] means there are 3 pieces of a combined ingredient which consists of a piece of “a” and 2 pieces of “b”
- 3a2[b] means there are 3 pieces of “a” followed by 2 pieces of “b”

The square brackets can be neglected when they are given in the input file, if removing them is not confusing. For example, 3[a]4[cb] can be 3a4cb or 3a4[cb] or 3[a]4cb, but brackets of 3[a2b] cannot be neglected because 3a2b and 3[a2b] are totally different.

Input Files

An input file (input.txt) is given which has thousands of lines and in each line, there is a string representing a raw ingredient (such as 3[a]) or a combined ingredient “3[1[a]2[b]]”. Use this input file for the following questions.

In “input.txt” file:

```
3[a]
4[bw]2a
2a3[1[a]2b]
3[1[a]2[b]]
2[4[b]1gd]3cd
```

Q1. Sequential Program (40%)

In this question, there is only one burger maker to organize encoded ingredients. Since the ingredients are encoded as k[ingredient string], they need to be decoded as a spreaded string with only letters but no number or bracket, in order to form a burger. The rule for decoder is that k[ingredient string], where the ingredient string inside the square brackets is being repeated exactly k times. The following table shows some examples. Note that ASCII may be helpful to differentiate characters.

input	output
3[a]	aaa
4[bw]2a	bwbwbwbwaa
2a3[1[a]2b]	aaabbabbabb
3[1[a]2[b]]	abbabbabb
2[4[b]1gd]3cd	bbbbgdbbbbgdcdcdcd

Your task is to write a C/C++ program in the given "EE599_Lab9_part1_q1_USCID.cpp" following the given data structure to decode ingredient strings with the above rule and output the results into "part1_q1_out.txt" file, as shown below. And using time system call, report the time taken by your program in README file. Example:time ./decoder

In "part1_q1_out.txt" file:

```
aaa
bwbwbwbwaa
aaabbabbabb
abbabbabb
bbbbgdbbbbgdcdcdcd
```

For Q2-4, Please include "pthread.h".
You can not include " thread.h"(will lose whole points in Q2 - 4).
We will provide a sample .cpp file to you soon.

Q2. Parallel Programming with 2 Threads (15%)

In this question, there are two burger makers to organize encoded ingredients. Modify the program written in Q1 with the idea of parallelism and named the new file as "EE599_Lab9_part1_q2_USCID.cpp". Use two threads to decode ingredient strings, i.e. one thread to decode the first half of lines in the input files and the other thread to decode the last half of lines. For example, given 1000 lines, the first thread decode the first 500

lines and the second thread decode the remaining 500 lines. Output the decoded result into "part1_q2_out.txt", which should be the same as in Q1.

Report the time(use the test "test->Part1&2->3") taken by your program on the same data used in Q1 in README file. However, keep in mind that parallel programming may not be beneficial if implemented wrongly. Since you run your code in a virtual machine, you must explicitly adjust the number of cores assigned.
For time measurement, use same way you did in lab2.

Q3. Parallel Programming with more Threads (15%)

In this question, there are at most 10 burger makers to organize disordered ingredients. Modify your program to support the following feature: *./decoder_threads 4*, where 4 is the number of threads to spawn to implement ingredient string decoder, and it can be an integer from 3 to 10. The new file is named as "EE599_Lab9_part1_q3_USCID.cpp". No need to output decoded results to text files in Q3.

You will notice that your system can handle a fixed number of threads after which the performance of your algorithm will start degrading because of large overhead in terms of creating extra threads. Try different number of threads (1 to 10) and report the time(Screenshot your command line and paste it in your readme file) taken on the data set(Use the test "test->Part3") and the maximum number of threads after which the performance starts degrading. For this part run your program on server "viterbi-scf2.usc.edu".

To log into the server "viterbi-scf2.usc.edu" in Unix or Mac machines:
Open Terminal and type "ssh -X [username]@viterbi-scf2.usc.edu", where [username] is the username of your USC NetID, such as "david". And then enter password of your NetID.

To log into the server "viterbi-scf2.usc.edu" in Windows machines:
Download X-win from https://software.usc.edu/x-win32/ and follow the instruction on the website to install. And then using host name "viterbi-scf2.usc.edu" and log in with your USC NetID.

To transfer your files to the server, use FileZilla software or use "scp" command in your local terminal. The content of [ ] should be changed respectively.

$ scp [/local/directory/filename] [username]@viterbi-scf2.usc.edu:[/remote/directory/]

After logging into the server, you can use "less /proc/cpuinfo" to check the

CPU information of the server which has 10 CPU cores. And then run your code on the server and report your running time in README file. Refer to https://itservices.usc.edu/scf/ for more information about the USC server.

Q4. Synchronization (30%)

In computing, the producer–consumer problem is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. One consideration is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. Another consideration is to make sure there would be no issues in the existence of multiple producers.

In this question, there are 2 producers and 1 consumer. Write a C/C++ program named "EE599_Lab9_part1_q4_USCID.cpp" to simulate producer-consumer problem, where the size of the buffer is passed through command line. Example: ./producer_Consumer 10, where 10 is the size of the buffer. The output of this program should look like this:

Producer 1 produces item 1 at index 0
Producer 2 produces item 2 at index 1
Consumer consumes 1 from index 0
Consumer consumes 2 from index 1

Producer 1 produces 100 items and Producer 2 produces 50 items, where items are some random numbers (integers in range 0~99). The simulation should terminate after the consumer has consumed all the data. You are **NOT** required to check full and empty buffer using condition variables. If you do so, you will get extra credits.

Extra Credits (10%)

If you feel that the algorithm you implemented in Q1 is efficient and creative, please write the pseudocode of your algorithm in README file. If it's more efficient than our implemented one, you can get the extra credit.
For Question 4, handling full and empty buffer is taken as extra credits, as long as you check full and empty buffer using condition variables to avoid overriding of data in the buffer.

Deliverable

EE599_Lab9_part1_q1_USCID.cpp

EE599_Lab9_part1_q2_USCID.cpp

EE599_Lab9_part1_q3_USCID.cpp

EE599_Lab9_part1_q4_USCID.cpp

README_[USC_ID].txt: report time of using 1 to 10 threads to decode ingredient strings, report the maximum number of threads after which the performance starts degrading, pseudocode for extra credit, references, etc.

Output files

Compiling and running your deliverable files should generate the following output files. Do not submit your output files.

part1_q1_out.txt

part1_q2_out.txt

Notes

- Please use the given data structure in the given "EE599_Lab9_part1_q1.cpp" to write all of your code in this part.
- The number of lines in the input file "input.txt" is not given and not fixed. You should get it by reading the input file.
- Do not submit your output files. Only submit files mentioned in "Deliverable".
- You are required to use POSIX Threads and not allowed to use std::thread in C++11.

samplecpp

test

Part II - Data Streaming

Part II - Data Streaming

Apache Kafka is a distributed streaming system which was originally developed by LinkedIn. It is a scalable, fault-tolerant, publish-subscribe messaging system to build distributed applications. It is used in many web-scale Internet companies such as Hulu, Intel, LinkedIn, and so on. As an example application, the GPU database that is supposed to linearly scale out in the cloud systems, to have connectors for Kafka, so one can read/write data-in-motion and data-at-rest, and the GPU database can function as the fast layer on top of your data stores. Another example can be that, one can pull the data from different social media resources, and push data to Kafka, so that the consumer can read data from Kafka and maybe do some statistics analysis.

Prelab 9:

Please refer to the recommended part (watching my tutorial on computer networks) before starting this lab, as having the knowledge of networks helps a lot with understanding the motivation and challenges with data streaming in cloud (using Kafka, etc.)

Reminder: For Part II, Q2 and Q3, you are allowed to look up for examples online and modify based on them, including examples in the Github link provided for "librdkafka" library. For any citation, please mention in the README file as the reference. Also, remember to preserve the "Copyright" comments in your code if any.

Pre-step: Make sure you have Java

You may encounter "java not found" error when running Kafka if you don't have Java. You can install it following the instructions on the Lecture slides:

> *sudo apt-get install default-jre*

> *sudo apt-get install default-jdk*

Installations

(1) Quick Start of Kafka

First, let's install Kafka step by step and run some examples in the terminal. Here is some guideline about how to install Kafka on your computer. You may also get some reference from Kafka's website (http://kafka.apache.org/quickstart).

Step 1: Download

(1) Download the tar file for Kafka from https://kafka.apache.org/downloads. Choose the 2.11 release.

(2) Un-tar it:

> tar -xzf kafka_2.11-2.1.1.tgz

(3) Open the folder as the current path

> cd kafka_2.11-2.1.1

Step 2: Start the server

Kafka uses ZooKeeper. So if you don't already have one, you need to first start a ZooKeeper server, by:

> bin/zookeeper-server-start.sh config/zookeeper.properties &

Then start the Kafka server by:

> bin/kafka-server-start.sh config/server.properties &

Keep this terminal open.

Step 3: Create a topic

Now we open a new terminal. Let's create a topic named "Topic-1" with a single partition and only one replication:

> bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic Topic-1

We can now see that topic if we run the list topic command:

> bin/kafka-topics.sh --list --zookeeper localhost:2181

You can repeat this to create "Topic-2", "Topic-3" .....

Step 4: Send some messages

Run the producer and then type a few messages into the console to send to the server.

> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic Topic-1

This is Topic-1 message 1

This is Topic-1 message 2

If you have finished with all messages you want to send, use "ctrl+C" to exit. Same for Topic-2, Topic-3…

Step 5: Start a consumer

Kafka also has a command line consumer that will dump out messages to standard output. Now we can open a new terminal to start acting as the consumer.

> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic Topic-1 --from-beginning

If you have each of the above commands running in different terminals, then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal in the real-time.

(2) Install Apache Kafka C/C++ client library (Copyright (c) 2012-2018, Magnus Edenhill)

In this lab, you will be asked to build producer and consumer using C++. **librdkafka** is a C library implementation of the Apache Kafka protocol, containing both Producer and Consumer support. You can download the package from GitHub https://github.com/edenhill/librdkafka. To build the library, follow:

> ./configure

> make

> sudo make install

Problem Description

In this lab, you will mimic the transmission of log files using Kafka.

You need to build both the producer and the consumer. In your producer end, you upload the real-time generated log files to the Kafka clusters. In the consumer end, you act as the computer or software company who's downloading and maintaining log files reported (uploaded) from customers.

Besides, in order to have the real-time generated files, you also need to write a program that generates files automatically. Therefore, there are three parts you need to do:

Q1. Files Generator (40%)

Write a program named "EE599_Lab9_part2_**FileGene**_USCID.cpp" in C++, or "EE599_Lab9_part2_**FileGene**_USCID.py" in Python that will generate a file every 5 seconds. You need to generate 20 such files. The first line of each file should have a topic name which is selected randomly from these 4 available options: **INFO**, **WARN**, **ERROR** and **FATAL**. The second line of each file should have a random number generated between 1 and 1000 which represents the error type index. The files should be named as "logfile-YYYYMMDD-HHMMSS.txt". For example, the log file created at 22:02:34 on March 6, 2019 should have the name "logfile-20190306-220234.txt".

Q2. Build Kafka Producer (30%)

Write a producer application in C++ named "EE599_Lab9_part2_**Producer**_USCID.cpp" that should publish the content of each generated file every 5 seconds onto the Kafka server to the corresponding topic which is indicated in the first line of the file.

Q3. Build Kafka Consumer (30%)

Write the consumer application in C++ named "EE599_Lab9_part2_**Consumer**_USCID.cpp" which should consume the message topic-wise. Print the contents in form of [YYYYMMDD-HHMMSS] followed by topic name and error type index on the console for each received file. For example, "[20190306-220234] INFO 5". The time information is the current time at the consumer end.

Deliverables:

1) EE599_Lab9_part2_FileGene_USCID.cpp or EE599_Lab9_part2_FileGene_USCID.py

2) EE599_Lab9_part2_Producer_USCID.cpp

3) EE599_Lab9_part2_Consumer_USCID.cpp

4) Makefile_USCID

5) README_part2_USCID.txt: Here, include any references (paper, website link, etc) you used.

Slides for part1

Conditional Variables

In addition to the "Slides for Part I" presented by Xuejing, feel free to review the conditional variable pages

submission link

https://classroom.github.com/a/1v7qig4K