Monday,1st, April

CSC4006-SOFTWARE DEVELOPMENT REPORT

By Tianbai Peng 40120405

1.Introduction

The main purpose of this software development report is for helping people who want to implement the relevant Fastflow code and also some stateful and adaptive experiments which refered in the paper.I will talked my implemented software in 4 parts: system specification,design,implement,and testing.

2.System specification

The propose of the system is to do the adaptivity experiments with the stateful task data. We think thorough this system, we can do the adaptivity and stateful experiment easily and accuracy, and get the experiment outputs what we want.

The whole system what I implement base on the Fastflow, which is an very nice parallel programming framework developed and maintained by the parallel computing group at the Departments of Computer Science of the Universities of Pisa and Torino, We choose to implement our program on the Fastflow parallel framework because:

1. the task-farm parallel pattern could be found at the Fastflow library, which is very helpful for our research
2. the overhead of Fastflow is very small so that we can get much speedup on the concur renting application.

This whole programming framework has been developed according to a design on top of Pthread/C++ standard fastflow programming framework - the task-farm pattern,the pattern has 3 main classes: Emitter,Worker,and Collector.The Emitter is the most important function in our development, because it not only send out the tasks to the Workers, but also we can override the load balancer class and used it in the emitter function to schedule the tasks,only in this way

We can implement the function of the adaptive.The workers is for processing the tasks from emitter, these Workers can running at the same time to accomplish the concurrent programming .And finally the Collector function can got the outputs of each task from Workers and combine them together.

The task model class what we define in this system is called Account and Transaction,which is a synthetic input data which simulate the many transaction could be processed in one bank account,the save/withdraw transaction could be seen as stateful task, and the query request could be seen as a stateless task.

3.Design

The main idea of this design is base on the object-oriented the client (test),and fully reflects the three characteristics of object-oriented(encapsulation, inheritance and polymorphism) which could let us directly call the function that defined on the head file and don't need to see the detailed implement stuff,which let the experiments on fastflow safely and easily.

head file: adapt.h

source file: adaptivity.cpp

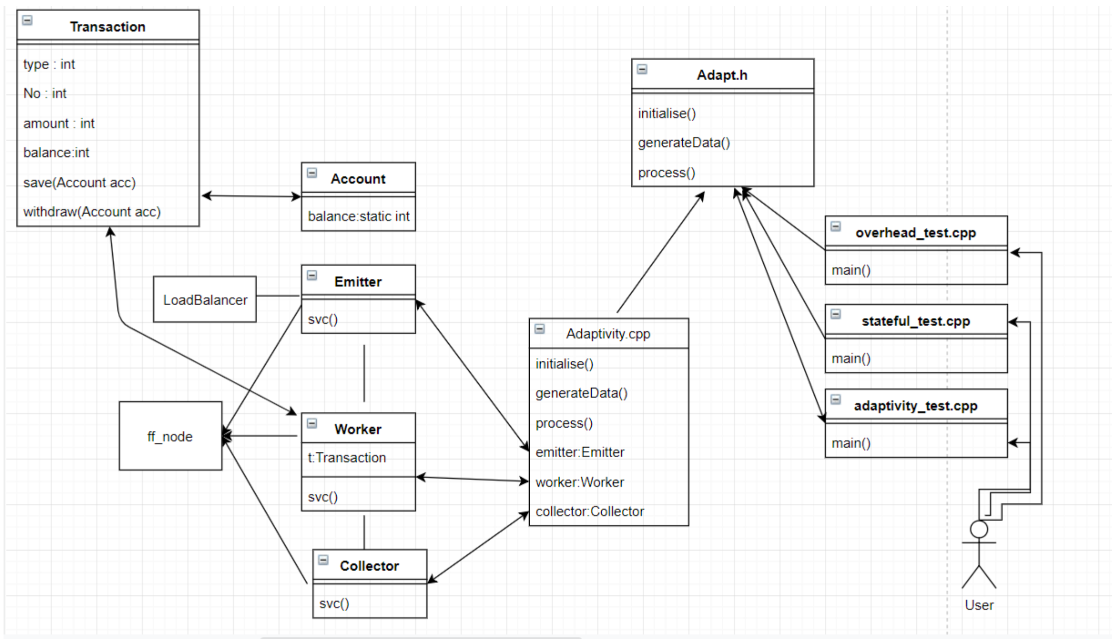
test file:

overhead_test.cpp

stateful_test.cpp

adaptivity.cpp

For easily understand, the simple UML graph as below:



The UML Diagrams of this program gives us an overview of the Fastflow application implement. This displays the proposed main files in the program and shows the connections between the different files, classes and functions.

4.Implementation

All our program is developed by c++ and mainly using the Fastflow c++ library, and the Fastflow framework is provided as a set of header files. The task-farm pattern what I used is also provided inside. Just according to the UML graph above, we can see that the main part of this architecture is the Emitter-Worker-Collector, and all of them is extended by the ff_node class, which is very important and it's the basic of Fastflow.

head file adapt.h

The head file defines some functions that user may call directly:

initiate()

generateData()

process()

source file adaptivity.cpp:

The source file is implemented with every detail

ff_node:

The ff_node is a basic thread which defined on fastflow , and it processes tasks by through its svc method. It processes data items that appear on a single input channel, and passes the relevant results to a single output channel. In multicore, the ff_node object is implemented as a non-blocking thread (or a set of non-blocking threads). This means that the number of ff_nodes running at the same time should not exceed the number of logical cores at hand. The latest version of Fastflow also supports blocking runtimes. You can choose to block the runtime by compiling the code with the flag -DBLOCKING_MODE.

The ff_node behaves as a loop that takes the input task (from the input channel), that is, the input parameter of the svc method, and generates one or more outputs, that is, the return value of the svc method or the call of the ff_send_out method can be called in the svc method. If the supplied output or received input is a special value: "End-Of-Stream" (EOS), the loop terminates. EOS spreads across the channel to the next ff_node. The specific case of ff_nodes can be implemented simply without an input channel or without an output channel. The former is used to install concurrent activities that generate output streams (for example, data items read from the keyboard or from disk _le); the latter installs concurrent activities that consume input streams (for example, displaying results on video, storing them on disk) Or send the output packet to the network)

Emitter and Collector:

We define the Emitter and Collector classes by extending the ff_node class. Emitter encapsulates our scheduling code in the svc method and task scheduling policy, which defines how tasks are sent to workers. Here, I implement adaptive functionality by redefining the schedule. Similarly, the Collector node encapsulates the code and task collection strategy, thus defining how tasks are collected from the staff. The planning and collection strategies for the Fastflow field skeleton can be redefined.

Worker:

The workers are for processing the tasks concurrently, we implement it by get the tasks from emitter then write tasks to the svc() function,

LoadBalancer:

In order to select the worker where an incoming input task has to be directed, we could use the Fastflow farm uses an internal ff_loadbalancer that provides a method int selectworker () returning the index in the worker array corresponding to the worker where the next task has to be directed. Therefore I subclass the ff_loadbalancer and provide our own selectworker() method and then pass the new load balancer to the farm emitter, therefore implementing a farm with a user defined scheduling policy.

Here I used ff_send_out_to method of the ff_loadbalancer class directly in the svc method of the farm emitter . In this case what is needed is to pass the default load balancer object to the emitter thread and to use the ff_loadbalancer::ff_send_out_to method instead of ff_node::ff_send_out method for sending out tasks.

initiate():

This function is for initiating all the necessary variable and get the test type from the user.

generateData():

This function is for generating the artificial input data, the users can assign the percentage of the stateful tasks here.

process():

This function can be called by user to create Emitter, Workers, Collector and task-farm and finally running the pattern to execute the experiments.

libraries

- Adapt.h – the head file of this program
- vector – vectors are sequence containers representing arrays, here for contain the workers
- iostream – input and output stream, which is a basic library in c++
- ff/farm.hpp – the library of task-farm pattern in Fastflow
- ff/pipeline.hpp – the library of pipeline pattern in Fastflow
- Math.h – the math function library, sometimes we will use Random to generate some tasks with random running time
- chrono – a high accuracy timer for recording the consuming of each experiment

5.testing

for testing the relevant experiments which refered on the paper, you can excuted the following command simplify, and also, we can make our own test to run.

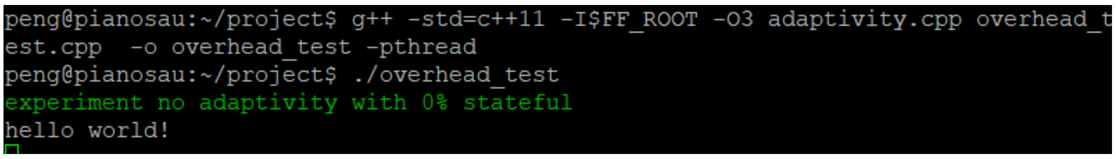
repro steps:

1. set the path of Fastflow directory as a environment variable:

   export FF_ROOT=$HOME/fastflow
2. Go to the current directory of the program
3. Compile the overhead test file: g++ -std=c++11 -I$FF_ROOT -O3 adaptivity.cpp overhead_test.cpp -o overhead_test -pthread
4. Execute ./overhead_test then we can check the overhead experiment output.
5. Compile the overhead test file: g++ -std=c++11 -I$FF_ROOT -O3 adaptivity.cpp stateful_test.cpp -o stateful_test -pthread
6. Execute ./stateful_test then we can check the stateful experiment output.
7. Compile the overhead test file: g++ -std=c++11 -I$FF_ROOT -O3 adaptivity.cpp adaptivity_test.cpp -o adaptivity_test -pthread
8. Execute ./ adaptivity_test then we can check the adaptivity experiment output.

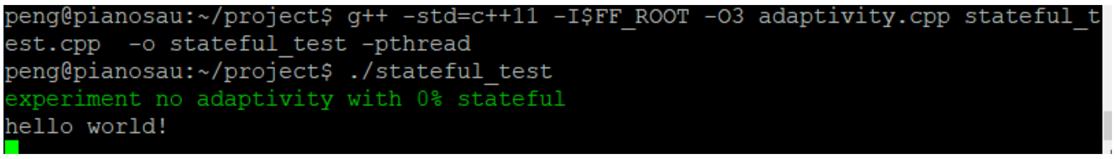
Test result:

1.



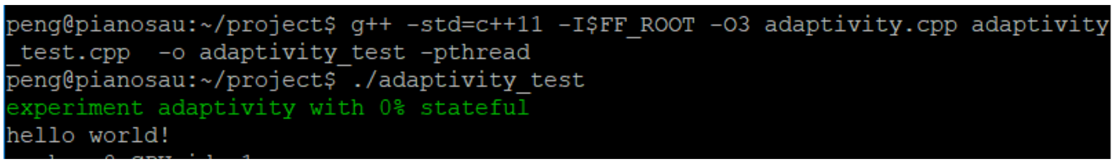
```
peng@pianosau:~/project$ g++ -std=c++11 -I$FF_ROOT -O3 adaptivity.cpp overhead_t
est.cpp  -o overhead_test -pthread
peng@pianosau:~/project$ ./overhead_test
experiment no adaptivity with 0% stateful
hello world!
```

2.



```
peng@pianosau:~/project$ g++ -std=c++11 -I$FF_ROOT -O3 adaptivity.cpp stateful_t
est.cpp  -o stateful_test -pthread
peng@pianosau:~/project$ ./stateful_test
experiment no adaptivity with 0% stateful
hello world!
```

3.



```
peng@pianosau:~/project$ g++ -std=c++11 -I$FF_ROOT -O3 adaptivity.cpp adaptivity
_test.cpp  -o adaptivity_test -pthread
peng@pianosau:~/project$ ./adaptivity_test
experiment adaptivity with 0% stateful
hello world!
```

After the successfully execution, we can prove that the program has no significant weakness.

6.Conclusion

This report clearly describes the development process of the adaptive experiment on Fastflow by describing the system, design, and implementation of these parts, so that people can better understand the experiment and related theory knowledge in the paper.