Pthreads_Report_Team11

Team Member

- 資工大三 108062213 顏浩昀
- 電資大三 108061250 吳長錡

Contributions

Work	Member
Trace code	顏浩昀&吳長錡
Implement	顏浩昀&吳長錡
Debug	顏浩昀&吳長錡
report	顏浩昀&吳長錡

Part I Implement

Producer.hpp

```
void Producer::start() {
    // TODO: starts a Producer thread
    pthread_create(&t, 0, Producer::process, (void*)this);
}
```

在Producer::start()中新增pthread_create(&t, 0, Producer::process, (void*)this)，call start()代表要開始進行這個thread的行為，而行為的定義、作法寫在process()中。

```
void* Producer::process(void* arg) {
    // TODO: implements the Producer's work
    Producer* producer = (Producer*)arg;

    while(true){
        Item *item = new Item;
        item = producer->input_queue->dequeue();
        item->val = producer->transformer->producer_transform(item->opcode, item->val);
        producer->worker_queue->enqueue(item);
    }

    return nullptr;
}
```

Producer是介於input queue與worker queue之間。中間的過程是將input_queue的item透過dequeue取出，然後透過producer_transform改變取出的item的value，再透過enqueue放入worker_queue中。

Consumer_Controller.hpp

```
void ConsumerController::start() {
    // TODO: starts a ConsumerController thread
    pthread_create(&t, 0, ConsumerController::process, (void*)this);
}
```

這邊的start同樣呼叫pthread_create，並執行ConsumerController::process中定義的行為。

```
while(true){
    usleep(consumerController->check_period);
    int queue_size = consumerController->worker_queue->get_size();
    if(queue_size > (consumerController->high_threshold) ){
        Consumer* consumer = new Consumer(consumerController->worker_queue, consumerController->writer_queue, consumerController->transformer);
        old_size = consumerController->consumers.size();
        consumer->start();
        consumerController->consumers.push_back(consumer);
        new_size = consumerController->consumers.size();
        std::cout<<"Scaling up consumers from " << old_size << " to " << new_size << std::endl;
    }
    else if( (queue_size < consumerController->low_threshold) && (consumerController->consumers.size()>1) ){
        Consumer* consumer = consumerController->consumers.back();
        old_size = consumerController->consumers.size();
        consumer->cancel();
        consumerController->consumers.pop_back();
        new_size = consumerController->consumers.size();
        std::cout<<"Scaling down consumers from " << old_size << " to " << new_size << std::endl;
    }
}
```

在process中最重要的就是這個while loop。首先先透過usleep()讓我們每隔固定period會去執行while中的行為。而while loop裡要做的事情是去了解現在Consumer work的情形，去判斷需不需要增加或減少Consumer的數量。判斷的條件是根據worker_queue中的item數量，如果queue size > 我們定義的上限，那便會新增一個Consumer一起進行工作。相

反地，如果queue size < 定義的下限，那我們便會cancel最新產生的一個Consumer。而判斷Consumer產生的先後順序是透過consumers這個verctor，新產生出來的Consumer會放於vector的最後面，如果要cancel也只需要拿掉最後面的Consumer即可。

Consumer.hpp

```
void Consumer::start() {
    // TODO: starts a Consumer thread
    pthread_create(&t, 0, Consumer::process, (void*)this);
}
```

start執行process定義的行為

```
void* Consumer::process(void* arg) {
    Consumer* consumer = (Consumer*)arg;

    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, nullptr);

    while (!consumer->is_cancel) {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, nullptr);

        // TODO: implements the Consumer's work
        Item *item = new Item;
        item = consumer->worker_queue->dequeue();
        item->val = consumer->transformer->consumer_transform(item->opcode, item->val);
        consumer->output_queue->enqueue(item);

        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, nullptr);
    }

    delete consumer;

    return nullptr;
}
```

Consumer負責的事情是將worker_queue的item搬移到writer_queue。跟Producer進行的過程很像，先將前面queue的item透過dequeue取出，經過transform改變value，最後用enqueue放入下一個queue中。

比較不同的是，Consumer的數量會被ConsumerController所控制，所以有可能會在中途被cancel掉。為了避免Consumer搬移其中一個item時被cancel，夠過setcancelstate來避免在中途被cancel的可能性。

```cpp
int Consumer::cancel() {
    // TODO: cancels the consumer thread
    is_cancel = true;
    int id = pthread_cancel(t);
    pthread_join(t, NULL);
    return id; //return thread id
}
```

Consumer的cancel是會cancel自己，與一般網路上看到cancel其他thread的想法較不相同，但其行為簡單，就是呼叫pthread_cancel並指定自己將自己cancel。

Writer.hpp

```cpp
void Writer::start() {
    // TODO: starts a Writer thread

    pthread_create(&t, 0, Writer::process, (void*)this);
}
```

start同上，呼叫pthread_create去執行process。

```cpp
void* Writer::process(void* arg) {
    // TODO: implements the Writer's work
    Writer* writer = (Writer*)arg;

    while (writer->expected_lines--) {
        Item *item = new Item;
        item = writer->output_queue->dequeue();
        writer->ofs << *item;
    }

    return nullptr;
}
```

Writer的行為與Reader及其相似，只是方向相反。在知道預期的輸出大小(expected_lines)情況下，透過while loop將處在writer_queue中的item一個一個取出並逐一寫入output file中。

main.cpp

```
// TODO: implements main function
TSQueue<Item*>* reader_queue;
TSQueue<Item*>* worker_queue;
TSQueue<Item*>* writer_queue;

reader_queue = new TSQueue<Item*>(READER_QUEUE_SIZE);
worker_queue = new TSQueue<Item*>(WORKER_QUEUE_SIZE);
writer_queue = new TSQueue<Item*>(WRITER_QUEUE_SIZE);

Transformer* transformer = new Transformer;

Reader* reader = new Reader(n, input_file_name, reader_queue);
Writer* writer = new Writer(n, output_file_name, writer_queue);

Producer* p1 = new Producer(reader_queue, worker_queue, transformer);
Producer* p2 = new Producer(reader_queue, worker_queue, transformer);
Producer* p3 = new Producer(reader_queue, worker_queue, transformer);
Producer* p4 = new Producer(reader_queue, worker_queue, transformer);

ConsumerController* consumerController = new ConsumerController(
    worker_queue,
    writer_queue,
    transformer,
    CONSUMER_CONTROLLER_CHECK_PERIOD,
    WORKER_QUEUE_SIZE/100*CONSUMER_CONTROLLER_LOW_THRESHOLD_PERCENTAGE,
    WORKER_QUEUE_SIZE/100*CONSUMER_CONTROLLER_HIGH_THRESHOLD_PERCENTAGE
);
```

首先先將各個resource進行initail，其中也需要將queue size傳入進行初始化，以及計算ConsumerController的上下限。

```
reader->start();
writer->start();

p1->start();
p2->start();
p3->start();
p4->start();


consumerController->start();


reader->join();
writer->join();
```

接著就是讓所有Resource開始運作，並且在reader與writer都完成後結束這支程式。

```
delete p1;
delete p2;
delete p3;
delete p4;
delete writer;
delete reader;
delete transformer;
delete reader_queue;
delete worker_queue;
delete writer_queue;
delete consumerController;
```

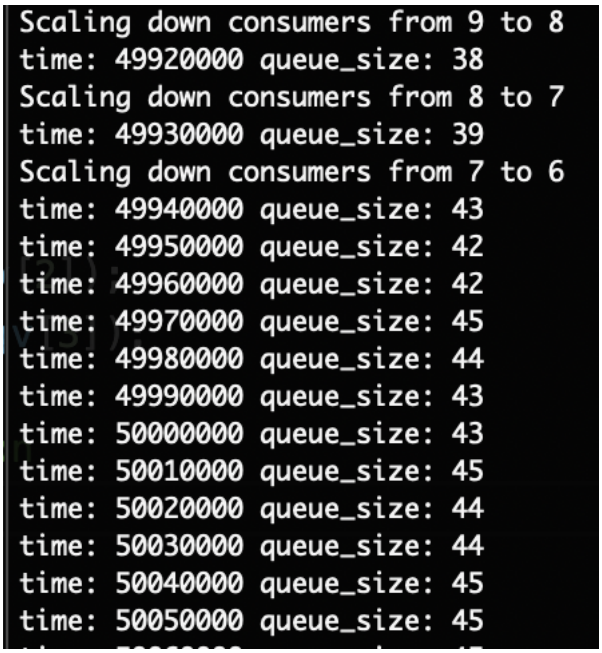
最後也不能忘記將資源釋放

Part II Experiment

因為01的testcase較大，較能明顯看出不同設定的表現差異，以下測試皆以01.in為測試資料。

1. Different period time

```
Scaling up consumers from 0 to 1
time: 1000000 queue_size: 92
time: 2000000 queue_size: 182
Scaling up consumers from 1 to 2
time: 3000000 queue_size: 200
Scaling up consumers from 2 to 3
time: 4000000 queue_size: 200
Scaling up consumers from 3 to 4
time: 5000000 queue_size: 200
Scaling up consumers from 4 to 5
time: 6000000 queue_size: 200
Scaling up consumers from 5 to 6
time: 7000000 queue_size: 200
Scaling up consumers from 6 to 7
time: 8000000 queue_size: 200
Scaling up consumers from 7 to 8
time: 9000000 queue_size: 197
Scaling up consumers from 8 to 9
time: 10000000 queue_size: 185
Scaling up consumers from 9 to 10
time: 11000000 queue_size: 134
time: 12000000 queue_size: 49
time: 13000000 queue_size: 0
```



首先實驗的是不同的period time，這影響ConsumerContoller去查看是否需要增減Consumer的時間間隔。

我們分別使用$10^{7(us),10}$6(us), 10^5(us), 10^4(us)做測試。

我們發覺使用間隔太大的period，會讓ConsumerController很難好好的顧及到實際的需求，會造成worker_queue都已經爆滿仍然只有少少的consumer在運作，而worker_queue都已經清空了卻仍然還有很多的consumer沒有被立即cancel。

而如果period太短，ConsumerController雖然能很快速的加減Consumer數量，但因為check的速度太快了，倒致worker_queue可能只多出一點點上限，卻會在短時間內加入太多Consumer，一下讓queue清空，卻又一下子減少太多Consumer，很快又會需要新增Consumer，在實際作業上應該是很耗費資源的行為。其中我們覺得$10^{5(us)、10}$4(us)是還不錯的選擇。

2. Different THRESHOLD_PERCENTAGE（period = 10^4us）
    1. 低下限（5,80）

        我覺得與正常的設定結果相比差距不大，但如果一直維持比較低的queue_size時可能會較為浪費Consumer，因為需要更低的percentage才會減少

Consumer數量。（例如下圖狀況）

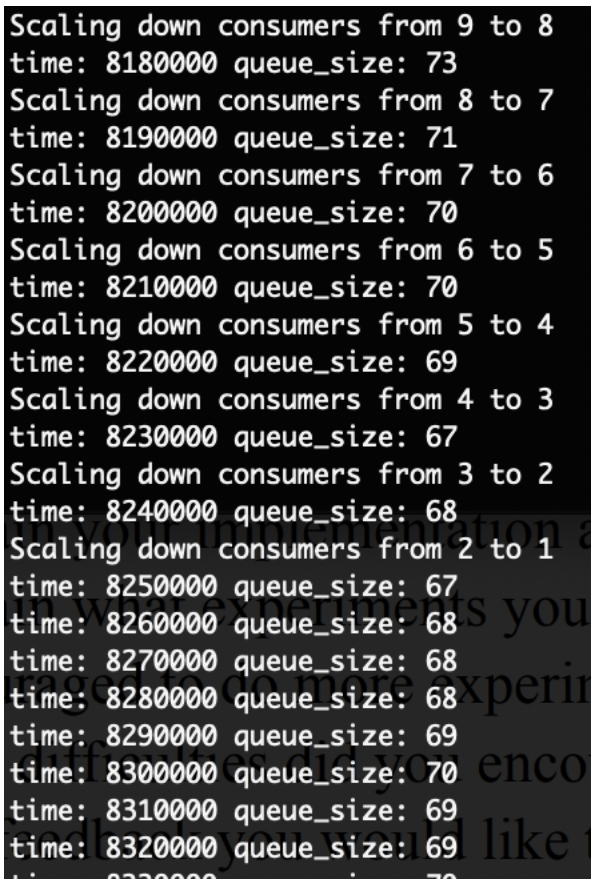
```
time: 49930000 queue_size: 26
time: 49940000 queue_size: 25
time: 49950000 queue_size: 26
time: 49960000 queue_size: 23
time: 49970000 queue_size: 25
time: 49980000 queue_size: 24
time: 49990000 queue_size: 23
time: 50000000 queue_size: 24
time: 50010000 queue_size: 25
time: 50020000 queue_size: 24
time: 50030000 queue_size: 22
time: 50040000 queue_size: 24
time: 50050000 queue_size: 22
time: 50060000 queue_size: 21
time: 50070000 queue_size: 22
time: 50080000 queue_size: 21
time: 50090000 queue_size: 19
time: 50100000 queue_size: 18
time: 50110000 queue_size: 15
time: 50120000 queue_size: 13
time: 50130000 queue_size: 12
time: 50140000 queue_size: 12
time: 50150000 queue_size: 11
time: 50160000 queue_size: 9
Scaling down consumers from 10 to 9
time: 50170000 queue_size: 0
Scaling down consumers from 9 to 8
```

2. 高上限（20,95）

```
time: 1800000 queue_size: 174
time: 1810000 queue_size: 175
time: 1820000 queue_size: 177
time: 1830000 queue_size: 177
time: 1840000 queue_size: 178
time: 1850000 queue_size: 179
time: 1860000 queue_size: 179
time: 1870000 queue_size: 181
time: 1880000 queue_size: 182
time: 1890000 queue_size: 184
time: 1900000 queue_size: 184
time: 1910000 queue_size: 184
time: 1920000 queue_size: 186
time: 1930000 queue_size: 186
time: 1940000 queue_size: 188
time: 1950000 queue_size: 189
time: 1960000 queue_size: 190
time: 1970000 queue_size: 191
Scaling up consumers from 1 to 2
time: 1980000 queue_size: 191
```

很明顯可以看出一開始的時候Consumer數量增加的很慢，有可能導致worker_queue被塞滿的情形發生，並且後續如果減少Consumer的話，會比較難加回來，可能在效率上較差。

3. 高下限（40,80）
   在worker_queue item多或少時比較沒有感覺，但如果item數是中間偏少時，就很容易發生Consumer被減的太少的狀況，效率會變差。



4. 低上限（20,60）
   因為上限低的關係，很容易用太多的Consumer，效

能會變得很差。

```
time: 2840000 queue_size: 46
time: 2850000 queue_size: 45
time: 2860000 queue_size: 47
time: 2870000 queue_size: 46
time: 2880000 queue_size: 44
time: 2890000 queue_size: 43
time: 2900000 queue_size: 42
time: 2910000 queue_size: 43
time: 2920000 queue_size: 43
time: 2930000 queue_size: 43
time: 2940000 queue_size: 41
time: 2950000 queue_size: 40
time: 2960000 queue_size: 39
Scaling down consumers from 12 to 11
time: 2970000 queue_size: 39
Scaling down consumers from 11 to 10
time: 2980000 queue_size: 38
Scaling down consumers from 10 to 9
time: 2990000 queue_size: 39
Scaling down consumers from 9 to 8
time: 3000000 queue_size: 41
time: 3010000 queue_size: 42
time: 3020000 queue_size: 41
```

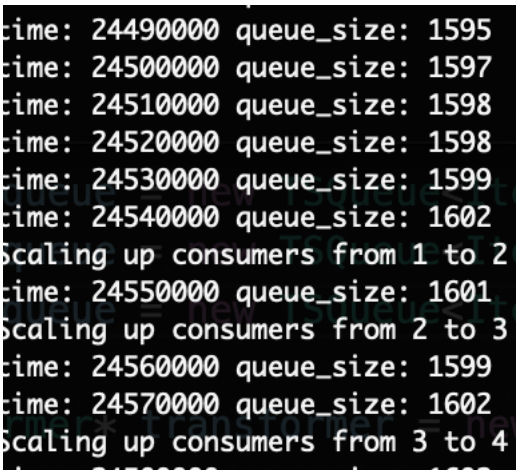
其實單一種高/低的上/下限我覺得比較不會感覺出太過誇張的差距。但想想如果低下限+低上限的組合，會讓Consumer輕易增加，卻很難減少，效能非常的差，也失去ConsumerController調節的意義。
又或者高上限與高下限的組合，會讓Consumer的數量維持的很少，就容易有很差的效率。

3. Different WORKER_QUEUE_SIZE (基本的上下限、period=10^4us)
    1. Worker_Queue = 2000

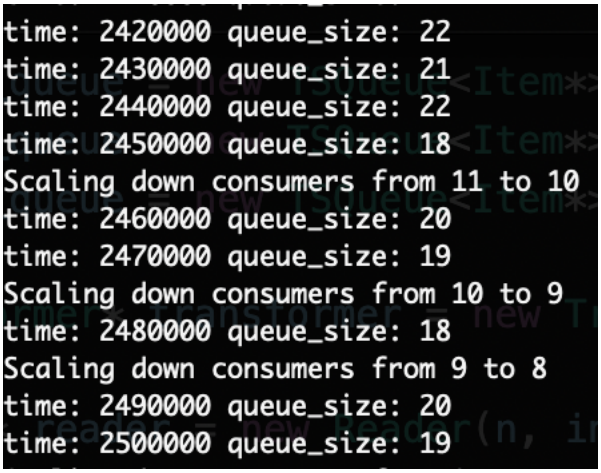
```
time: 49690000 queue_size: 200
time: 49700000 queue_size: 200
time: 49710000 queue_size: 200
time: 49720000 queue_size: 200
time: 49730000 queue_size: 200
time: 49740000 queue_size: 200
time: 49750000 queue_size: 200
time: 49760000 queue_size: 200
time: 49770000 queue_size: 200
time: 49780000 queue_size: 200
time: 49790000 queue_size: 200
time: 49800000 queue_size: 200
```

因為上下限正常，但Worker_Queue_Size放大的因素，增加Consumer的條件很差，效率非常非常的差。可以看見下圖中，都超過1500個item仍然只使用

一個Consumer，搬運效果極差無比



2. Worker_Queue = 100

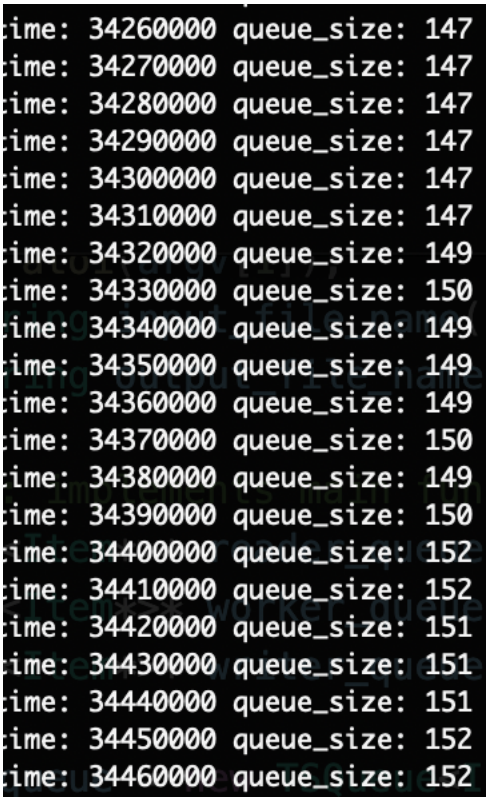


雖然並沒有很明顯，但可以看出會有點浪費太多Consumer的感覺，效能降低。我們認為不明顯的原因是因為100與200的差距不大，但因為要考慮使用百分比的關係，我們暫時不去測量更小的worker_queue。

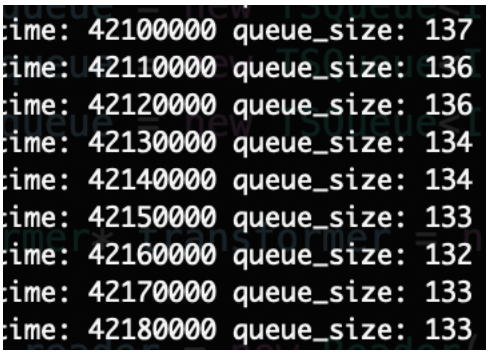
4. Small Writer_Queue and Reader_queue

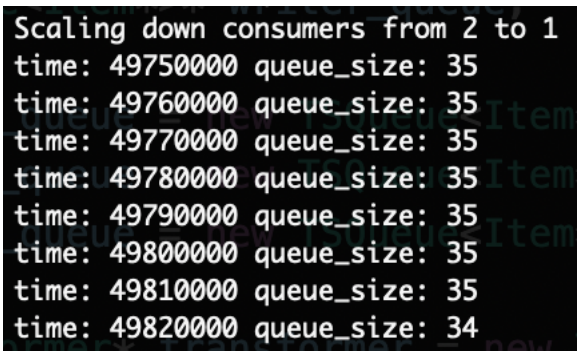
1. Writer_queue=10
   可能中途會有Writer_queue塞滿的情形，整個worker_queue size平均起來較高，雖然好像整體還沒

有遇到Worker_queue爆滿無法搬移的狀況，但仍有這種可能性存在。



2. Reader_queue=1
   沒有感受出明顯差異，但與我們的構想不太相同。原本我們認為會因為reader_queue小，導致沒有東西可以搬入worker_queue中，但看到整體仍有許多worker_queue size>100甚至150的情形發生。





Part III Difficulty

我們覺得這次最難的部分是Debug，因為每個部分不一定會一次做完，也不一定是照我們所想的先後順序運作。因此我們中間寫過很多debugmessage，有效幫助我們好好了解整個程式平行運作的流程。
另外，在程式要結束時，如何結束ConsumerController的問題也十分困擾我們。因為ConsumerController執行過程為infinite while loop。雖然我們在main function有delete，應該是能完整釋放資源，但這樣無法滿足作業要求的cout，因此我們試著在destrcutor的地方顯示output。