CS4204: Parallel Patterns

180007044
University of St Andrews
St Andrews, UK

INTRODUCTION

In this practical, there are two kinds of parallel patterns: farm and pipeline are designed and implemented in few ways. Besides, two similar queue structures are implemented as an essential tool for the implementation of patterns.

DESIGN OF FARM

According to the requirement, the farm pattern enables users specify the worker function and number of workers. In this practical, there are four parameters need to be set to create a farm.

The first parameter is the worker function, while the next one may be the number of workers.

Parameters

The following parameter is supposed to be parameters passed to workers. However, there are several ways to convey them. Hence in this practical, there are three versions of the farm pattern, corresponding to 3 ways of passing parameters.

Queue

Users could use a dynamic queue to pass parameters, which is named as task queues to distinguished them and another kind of queue discussed later. Each node of the task queue consists of the value of a parameter and the pointer pointing to the address of next node. According to the requirement of passing values, reading and writing the queue need to be implemented. So, a divider pointer is used to mark the progress of reading. Specially, there would be a mutex set in each queue to prevent that multiple threads read or write the queue at the same time.

The task queue makes it easier to pass a group of parameters whose size is not quite visible, such as data read from a file, which means users do not need to specify the number of workers, although it still needs them loopily put data into a task queue. The disadvantage is that the output of the farm, which is returned as another task queue, may not be ordered, since the task queue conveying parameters are probably read by several threads and the mutex may mess up the order.

```
FARM: queue parameters: output0:0
FARM: queue parameters: output1:30
FARM: queue parameters: output2:64
FARM: queue parameters: output3:102
FARM: queue parameters: output4:136
FARM: queue parameters: output5:170
FARM: queue parameters: output6:194
FARM: queue parameters: output7:4
FARM: queue parameters: output8:34
FARM: queue parameters: output9:70
FARM: queue parameters: output10:106
FARM: queue parameters: output11:138
FARM: queue parameters: output12:168
FARM: queue parameters: output13:192
FARM: queue parameters: output14:2
FARM: queue parameters: output15:36
FARM: queue parameters: output16:72
FARM: queue parameters: output17:76
FARM: queue parameters: output18:110
FARM: queue parameters: output19:142
FARM: queue parameters: output20:176
FARM: queue parameters: output21:6
FARM: queue parameters: output22:38
FARM: queue parameters: output23:74
FARM: queue parameters: output24:108
FARM: queue parameters: output25:140
FARM: queue parameters: output26:172
FARM: queue parameters: output27:198
FARM: queue parameters: output28:8
FARM: queue parameters: output29:40
FARM: queue parameters: output30:78
FARM: queue parameters: output31:84
FARM: queue parameters: output32:120
FARM: queue parameters: output33:152
FARM: queue parameters: output34:10
FARM: queue parameters: output35:42
FARM: queue parameters: output36:80
FARM: queue parameters: output37:112
FARM: queue parameters: output38:144
FARM: queue parameters: output39:178
FARM: queue parameters: output40:12
FARM: queue parameters: output41:44
FARM: queue parameters: output42:46
FARM: queue parameters: output43:86
FARM: queue parameters: output44:116
FARM: queue parameters: output45:146
FARM: queue parameters: output46:14
FARM: queue parameters: output47:48
FARM: queue parameters: output48:82
```

Figure 1. Example output of the farm with parameters passed by task queues. (Correct order: 0, 2, 4….)

Array

An array of void pointer could also be a virtue and tradition way to pass a group of parameters. Although it may need more steps to process parameters, which could bring more running time and more computation source occupying, the order of output of farm would corresponding to the order of input in this case, because the array would play a role of FIFO structure in this design.

```
FARM: array parameters: output0:0
FARM: array parameters: output1:2
FARM: array parameters: output2:4
FARM: array parameters: output3:6
FARM: array parameters: output4:8
FARM: array parameters: output5:10
FARM: array parameters: output6:12
FARM: array parameters: output7:14
FARM: array parameters: output8:16
FARM: array parameters: output9:18
FARM: array parameters: output10:20
FARM: array parameters: output11:22
FARM: array parameters: output12:24
FARM: array parameters: output13:26
FARM: array parameters: output14:28
FARM: array parameters: output15:30
FARM: array parameters: output16:32
FARM: array parameters: output17:34
FARM: array parameters: output18:36
FARM: array parameters: output19:38
FARM: array parameters: output20:40
FARM: array parameters: output21:42
FARM: array parameters: output22:44
FARM: array parameters: output23:46
FARM: array parameters: output24:48
FARM: array parameters: output25:50
FARM: array parameters: output26:52
FARM: array parameters: output27:54
FARM: array parameters: output28:56
FARM: array parameters: output29:58
FARM: array parameters: output30:60
FARM: array parameters: output31:62
FARM: array parameters: output32:64
FARM: array parameters: output33:66
FARM: array parameters: output34:68
FARM: array parameters: output35:70
FARM: array parameters: output36:72
FARM: array parameters: output37:74
FARM: array parameters: output38:76
FARM: array parameters: output39:78
FARM: array parameters: output40:80
FARM: array parameters: output41:82
FARM: array parameters: output42:84
FARM: array parameters: output43:86
FARM: array parameters: output44:88
FARM: array parameters: output45:90
FARM: array parameters: output46:92
FARM: array parameters: output47:94
FARM: array parameters: output48:96
FARM: array parameters: output49:98
FARM: array parameters: output50:100
FARM: array parameters: output51:102
FARM: array parameters: output52:104
FARM: array parameters: output53:106
FARM: array parameters: output54:108
```

Figure 2. Example output of the farm with parameters passed by arrays, whose order is correct.

Same parameter

It is also possible that users want all workers use the same parameter. Hence a very basic version is that users just pass a void pointer which would be passed to workers directly by *pthread_create*

Number of Threads

Users could limit the max number of threads based on their needs by passing an integer as the last parameter of the farm pattern. According to the relationship between the number of workers and this boundary, the pattern would use different strategies to run workers.

If the number of workers is not larger than the boundary, the pattern would create a thread for each worker, as that shown in Figure 3. Otherwise, it would put multiple workers in each thread, as that shown in Figure 4.

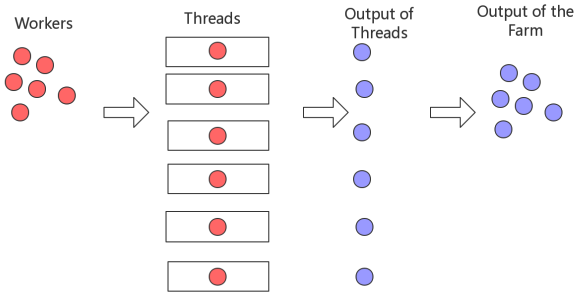

Figure 3. Example of the situation with enough threads

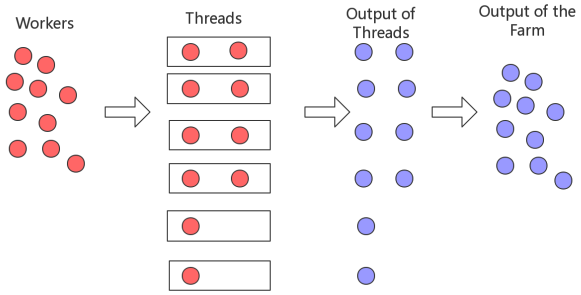

Figure 4. Example of the rearrange strategy

Specifically, the workers are run linearly in a mock way that the thread would yield until the worker running in a sub-thread returns, then the next worker would run in another sub-thread.

DESIGN OF PIPELINE

Since farm is a bit like a single-stage pipeline [1], the design of pipeline mainly focuses on the data stream between stages and each stage would run in parallel as a farm.

To implement the pipeline pattern, another kind of queues is designed, which is called the worker queue. Its structure and logic and of reading and writing is basically as same as the task queue. The difference is that its nodes represent stages of the pipeline. This design enables users add any number of threads to the pipeline according to their requirements.

Similar to the farm pattern, the pipeline is implemented in 3 ways, corresponding to 3 kinds of data stream.

Data Stream

Task Queue

Since the farm pattern always returns a task queues as output, it could be directly set as the input of the next stage. The output of the final stage would be returned. Advantages and disadvantages are similar to the queue farm. Its output is also unordered.

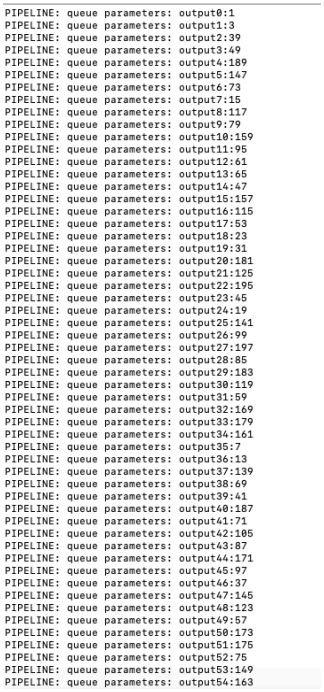

Figure 5. Example output of the pipeline with parameters passed by task queues. (Correct order: 1, 3, 5….)

Array

To convey parameters as arrays, it is necessary to transform the output of each stage from task queues to arrays. It could also give out ordered result as the array farm.

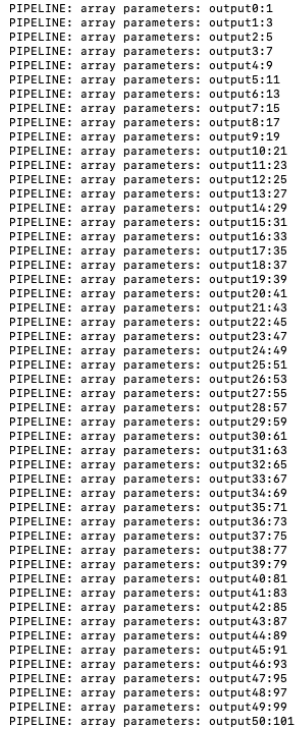

Figure 6. Example output of the pipeline with parameters passed by arrays

Same Parameter

It is worth notice that only the workers of the first stage would use the same parameter, and later stages would also receive parameters as task queues. Because some functions may return different values with the same input, such as random generation. Using task queues but not arrays is because there are no fixed orders in outputs in this case.

Number of Threads

By adding an attribute to the worker node, the pipeline patterns allow users to set different maximum number of threads for each stage. It makes it more suitable for flexible parallel tasks.

USAGE

All functions mentioned in this section would be indicated in the Appendix.

Users need to involve *parpat.h* in their C file by:

#include "parpat.h"

And compile their code by:

gcc mycode.c -lpthread -o mycode-exe

Worker functions are supposed to receive a void pointer as input and return a void pointer by calling *send_result(void *).*

All returned void pointer should be cast to task queues and read by *gettask(tq taskqueue).*

Farm

Users could directly create 3 kinds of farms by calling 3 corresponding functions.

Pipeline

Before creation of pipelines, users need to create a worker queue and add worker nodes as stages.

Users should notice that the name and parameters of functions would be different according to their requirement about parameters and order of output.

EVALUATION

Farm

Farm is the main target of evaluation in this practical. There are 14 test cases based on the provided example which calculate extremely large Fibonacci numbers, including 2 linear cases and other 12 cases to explore influence of ways of passing parameters, boundary of threads or number of workers to the performance result. The performance is measured by the running time. The result is shown in Figure 7.

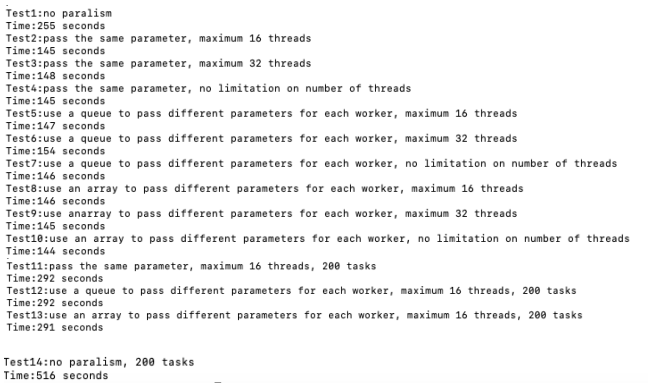

Figure 7. Performance of test cases (test 1-10 run 100 tasks)

It is obvious that test cases utilizing farm patterns all have much better performance (average 146.7s with 100 tasks and 291.7s with 200 tasks) than those run in linear steps (255s with 100 tasks and 516s with 200 tasks). To explore the relation between performance and several characterizes, data are extracted and shown in Table 1.

Table 1. Performance with different configurations

	Boundary of threads (100 tasks)			200 tasks with maximum 16 threads
	16	32	100 (No Boundary)	
Same Parameter	145s	148s	145s	292s
Queue	147s	154s	146s	292s
Array	146s	145s	144s	291s

According to Table1, it may be safe to draw a conclusion that farms passing parameters by arrays have the best performance. Taking its advantage of ordered output, it may be the most practical way. However, the gap is quite small and shrinks with a larger number of tasks, which means that the other two are also robust and could be used for different situations.

Besides, it is quite interesting that the boundary does not influence the result so much. Larger boundary even may have worse performance.

Pipeline

As that mentioned before, the design and implementation of pipelines mainly focus on data stream. Hence the evaluation would also aim at 2 kinds of data streams, task queues and arrays. The pipeline whose workers of the first stage use the same parameter is not involved because data are conveyed by task queues in later stages.

The test cases are also based on the provided Fibonacci example.

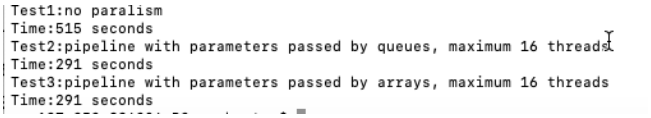

Figure 8. Performance of test cases (100 tasks in each stage)

As it is shown in Figure 8, data flow does not influence the performance of pipeline patterns in an obvious extent, users could choose upon them according to their needs.

REFERENCE

[1] Material of Lecture 8.