Adaptivity in Stateful Parallel Pattern

Tianbai Peng

***Abstract*—**

Nowadays, parallelism has become a new trend in software and architecture design, so that we usually consider a new solution for applications by using parallel patterns which include farm and pipeline pattern. And with cloud and edge deployments of parallel/distributed applications, the need for adaptivity in such applications has become important. For this, previous scientists have already conducted relative experiment to calculate the efficiency about running the stateful tasks on parallel framework. However, Balancing the computational load of multiple concurrent tasks on parallel framework is also one of the critical requirements for efficient usage of such systems [1], and at the moment still no one know what's the efficiency if balancing the stateful tasks on a parallel framework. To address this problem, it's necessary to measure how the ratio of sequence computing and parallel computing influence the running efficiency in a parallel pattern, and need to think about a new methodology or develop a new load-balancing task scheduler. In this paper, a c++ pattern-based parallel programming framework - Fastflow (Danelutto and Torquati, 2015) would be used, it provides developers with a set of parallel programming pattern, such as Map, Task-Farm, and Pipeline. Finally, the experimental results that relative to the stateful pattern with adaptivity implementation built on Fastflow would be present, and we will show that demonstrate the feasibility and efficiency of stateful pattern with adaptivity.

***Keywords*— load balancing, stateful computations, parallel design patterns**

I. Introduction

With the development of processor technology, people can enjoy the benefits of high-performance processors. However, processor technology has reached the bottleneck [2], and it's difficult for standard single-threaded code to increase more the computing speed. Therefore, in the recent years, everyone turned their attention to parallel programming. The theory of parallel computing is firstly dividing the code into each block, executes the blocks of code in parallel through multiple threads, and then integrates each sub-results together, which seamlessly extending computing power from a single processor to an unlimited number of processors. However, the implementation of parallel applications is a tortuous route. First of all, it is difficult to write explicit parallel programs because the programmer must specify how to divide the calculations on multiple processors, and must perform the necessary synchronization and data transfer operations, which not only has great requirements for programming ability, but also the program may contain some ignored error inside. In this case, programming using a parallel programming framework is particularly important, providing great convenience to developers. The popular mainstream parallel frameworks are OPENMP and MPI, etc., but these frameworks also have some various shortcomings. Fastflow as a parallel framework developed by Universities of Pisa and University di Torino [3], which is a multi-core programming framework that implements a lock-free MPMC FIFO queue specification to support advanced application development for multiple cores, and the speed is comparable with established frameworks such as TBB and OpenMP. [4].

In the parallel framework, the algorithm skeleton as the concurrent programming environment is its basic component. It was proposed in the 1990s and has been continuously developed by scientists all the time to make parallel frameworks more speed and advantage [5]. Among the skeletons of the latest research algorithms, pipeline and task-farm are the most common pattern. A parallel pattern is a well-defined programming mode - a form and method to describe parallel programming. However, in society the almost tasks we are processing at the moment by the skeleton of these algorithms is still Stateless, that is, the internal state of the parallel task is not changed when the parallel framework is running, so the current mainstream parallel framework does not support stateful task. However, if you want to perform stateful tasks in a parallel framework, the efficiency will be greatly reduced, and even the entire framework may be raising more errors than when dealing with stateless tasks. Therefore, the relevant experiments and research on stateful are imperative. Previous scientists have actually conducted related experiments and stated the results of running the stateful task in Fastflow [6].

However, in actual operation, the time spent on each task is different. The exact time required for each task depends on the task itself, and even base on the hardware environment at runtime. If you need to perform a lot of heavy tasks for a while,

Tianbai Peng is with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK (e-mail: tpeng01@qub.ac.uk).

you may need more threads, and instead you don't have to use so many threads when no many tasks, this procedure has been shown in Figure 1. If in this way, our utilization efficiency of the CPU will be greatly increased. Therefore, research on adaptivity is also imperative. But the adaptivity in the stateless case is relatively straightforward; adaptivity for the stateful case is more challenging. This is the subject of this study. As we all know, stateful tasks can't get good parallel efficiency. Therefore, adaptive research on stateful tasks is the current top priority. Once we understand their parallel operation efficiency through experiments, and even try to improve their efficiency, it is possible to make an important contribution and breakthrough in the current field of parallel programming.

In this paper, we propose a good idea about implement an adaptive way with the stateful pattern and realize it for Fastflow programming model [7]. The details of the approach are described in the Section II along with Fastflow's existing task scheduler and adaptive task scheduler. My approach is evaluated by a series of benchmarks in Section III Evaluation. The benchmarks include some synthetic model to verify the availability of the adaptivity, Finally, we draw our Conclusion in Section IV.

```
# define MAX_CONSUME
# define MIN_CONSUME

if(current_consume > MAX_CONSUME)
   workers ++;

if (current_consume < MIN_CONSUME)
   workers --;
```

Figure 1 pseudo code of basic adaptivity

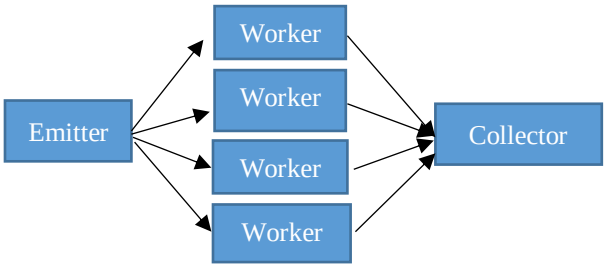

Figure 2 a Fastflow Task-Farm pattern

II. Related work and approach

Fastflow is a C++ framework to write parallel programs [7]. The framework consists many parallel patterns such as task-farm and pipeline, which are composed of ff_nodes, the basic thread in the Fastflow. The task-farm pattern for instance, is composition by an emitter node, several worker nodes, and a collector node as shown in the Figure 2. Note that in Fastflow, all nodes are implemented as individual OS threads. For instance, in the farm pattern, each of the various worker nodes as well as the emitter and collector nodes, respectively, are separate threads. The nodes have input and output queues to connect to each other thus forming basic patterns node, with help of an embedded load balancer, distributes tasks to the workers through their input queues. These workers, in principle, do the job concurrently, and the collector collects the results.[8]

If we want to implement the adaptivity function of this parallel framework, then we must have a deep understanding of how task scheduling works in the emitter of Fastflow. Task scheduling for parallel resources is an active area of research [9],[10]. The default scheduling policy of the emitter's embedded load-balancer is round-robin. We think that the round-robin scheduling policy is not suitable for implement the adaptivity because it assigns the tasks to each worker in turns. Therefore, we used another load-balancing scheme called auto scheduling (AS) within the Fastflow, we will implement the adaptive function base on AS and the pseudo-code of the adaptive implement is shown in Figure 3; the basic working is explained in the following.

The basic idea of the adaptive implement is to focus on the number of active workers, i.e., during a certain period of time, the heavy tasks will use more workers, and easy tasks will use less workers. We do so by recoding the during time between recent 10 tasks, and we set the specific total workers and some workers active at the begin, then if the consume larger than specific period, we need the adaptivity, so the active workers add 1 during every 10 tasks, if the consume less than specific second, that means we don't need the adaptive, so the active workers reduce 1 during every 10 task . Whenever, a task is ready for being emitted onto the workers, it is issued to the worker with the worker id is less than the active workers, i.e., with the active worker of total workers.

We may need to execute the adaptive test with stateful tasks, the stateful task may cannot get much parallel speed up, because to support task's internal state change in many threads (many tasks to access a shared state), a mutual lock must be applied to avoid the computation and deadlock between stateful tasks. Therefore, two tasks with stateful cannot be process at the same time. The task-farm pattern can be used to implement the state access pattern. The task-farm contain n workers and each worker has a copy of the shared state. The emitter sends tasks to workers with the adaptive scheduling policy that we mentioned before. The scheduling policy make sure tasks be send in an adaptive way.

In order to perform a task in parallel in Fastflow and making it perfectly reflects its correctness, it is necessary to make sure the correct order of the various small sub-tasks. That is to say, in what order the tasks in the emitter are input, the collectors must output the results in what order, and only in this way can

the results be aggregated correctly. Since it is difficult to guarantee the order of tasks in parallel execution, and based on the particularity of the stateful tasks, we implement it as the following: because we must ensure that the stateful task and the stateless task cannot be executed at the same time, we set global variable 'current_running_type' to record the current running number of stateful tasks and stateless tasks. In the Emitter, when the task to be sent is stateful, it must be sent when no tasks are running. When the stateless task is to be sent, this must be done. There are no stateful tasks to execute. In this way, the order of execution of the tasks is ensured. The detailed code and comment as shown in the figure 4 below.

```
# define MAX_CONSUME
# define MIN_CONSUME
# define START_LEVEL

int adaptive_level = START_LEVEL;
int currentConsume = 0;

def emitter(the_task){
   if(current_consume > MAX_CONSUME)
      adaptive_level ++;
   if(current_consume < MIN_CONSUME)
      adaptive_level --;

   send_out_to(the_task,taskId% adaptive_level);

}
// the workers process the tasks
def worker(the_task){}

def collector(the_task){
   currentConsume = someTimingFunction();
}
```

Figure 3 pseudo code of adaptivity algorithm

```
//flag for type, 0 is no running task, >0 means tasks are
stateless, <0 means tasks are stateful
Int current_running_type = 0;

def emitter(the_task){

      if (current_running_type<0 &&
the_task==stateless) wait;

      if(current_running_type>0 &&
the_task==stateful) wait;

      if(the_task==stateless)
         current_running_type++;
      if(the_task==stateful)
         current_running_type--;
}
// the workers process the tasks
def worker(the_task){}

//when the task finish running
def collector(the_task){
   if(the_task==stateless)
      current_running_type--
   if(the_task==stateful)
      current_running_type++;
```

Figure 4 pseudo code of task order algorithm

III. EVALUATION

The evaluation steps are divided in to four categories. Firstly, we measured the overhead of running the tasks on Fastflow by creating a synthetic benchmark. Here, we want to see the time spent by different threads (1~32) in the same task dataset to calculate the speedup they get. Secondly, we have defined a stateful access model for calculating the speedup they got when using diffident threads. Thirdly we applied adaptivity to the stateful run and recorded their experimental results. Finally, one real world applications have been exercised with the Fastflow parallel framework with adaptive and observe its result.

For calculate the ideal parallel speedup when doing the stateful pattern, we suppose all tasks running at 1 thread, the total process time is t. Percent of serial tasks is p, so percent of parallel tasks is (1-p), and we also suppose the tasks will be run at n threads. So if (1-p) parallel tasks run at n threads, and their total running time is:(1-p)*t/n + p*t Therefore we can get its parallel speedup:

t/[(1-p)*t/n + p*t] = **1/[(1-p)/n + p]**

In all benchmarks, we did at least run 5 times to calculate the average. Error are estimated from the standard error of mean over the samples. Error bars are shown in plots only if they are relatively large.

A. Evaluation Environment

The experiments have been conducted on the machine of Pisa University called pianosau. This machine has 2 CPU(Intel(R) Xeon(R) CPU E5-2650 @ 2.00Ghz) with 8 cores each. However, since hyper-threading was enabled, operating systems see (16+16) 32 logical cores. Each core is equipped with 32KB L1 data cache and 256KB L2 cache. And the operation system is Ubuntu 16.04.5 LTS linux.

B. Task Overhead on Fastflow

The purpose of this section is to measure the scheduling overhead by the Fastflow default scheduling method AS. To measure the overhead, a Fastflow application with uniform task is devised, which means every task takes same amount of time every time it is executed by a worker.

The Fastflow parallel framework consist of a task-farm pattern, i.e., emitter node, several worker nodes, and a collector node The experiment was exercised with 1, 2, 4, 8 and 16 workers with 100 tasks separately. The emitter of the farm distributes tasks to the workers through a scheduler (RRS or AS). The uniform task consists in imitating the process of bank account transactions, which single task takes around 1~2s time to execute on this machine. Figure 5 shows the performance of AS at this Fastflow application running with1, 2, 4, 8 and 16 workers, respectively.

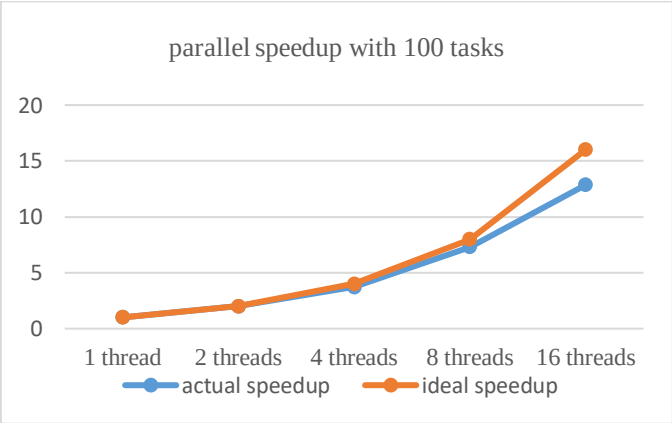

Figure 5 the overhead with different threads

According to the above figure, we can clearly see that the results can fully meet the speedup requirements. In this experiment, all workers participate in the execution of tasks and do nothing else. We assume there is no overhead by the scheduler and collector. As the number of threads increases, the actual speedup is approximately equal to the idel number of threads (speedup = nthreads), therefore the overheads of Fastflow are negligibly small. This way we can prove Fastflow is fully compliant with the requirements of parallel programming, enough for the following experiments.

C. Stateful on Fastflow

This section adds some stateful tasks base on the section B for doing the stateful experiment on Fastflow. And we will record each speedup for stateless, 1% stateful, 10% stateful and 20% stateful in with1, 2, 4, 8, 16 and 32 workers and in this experiment we also use 100 tasks. The results of these experiment can be seen at Figure 6 to Figure 9 as below.

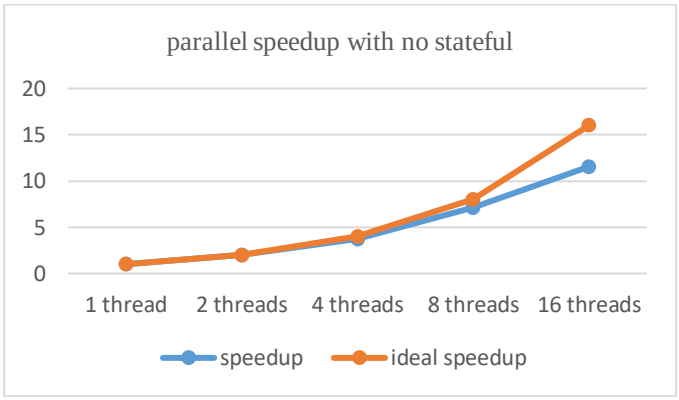

Figure 6 the parallel speedup with no stateful

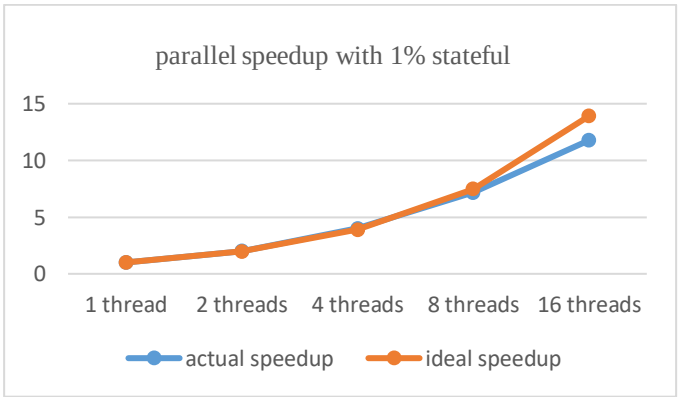

Figure 7 the parallel speedup with 1% stateful

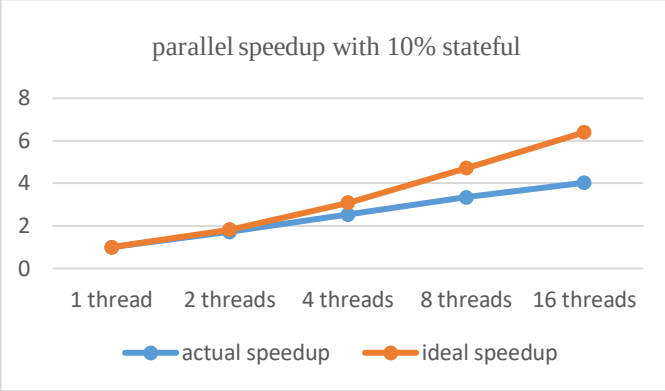

Figure 8 the parallel speedup with 10% stateful

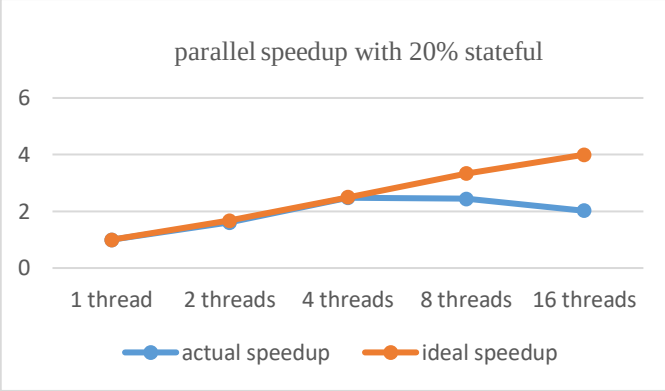

Figure 9 the parallel speedup with 20% stateful

According to the above figures, we can clearly see that the results can meet the speedup requirements but not totally fit the ideal parallel speedup especially in 8 threads and 16 threads. The different happens maybe the reason that in this experiment, we add some different level stateful tasks and those tasks cannot get enough parallel speedup because they couldn't do the parallel with other tasks due to avoid many tasks to access a shared state at the same time as we talked at section II. As the number of threads increases, the actual speedup is approximately equal to the ideal number of threads (speedup = **1/[(1-p)/n + p],** here n is number of thread and p is the percentage of stateful tasks). Although there is some diffidence between actual and ideal when do 20% stateful, we still can prove the stateful task could be done well at Fastflow, and next we will put the adaptivity stateful on these stateful tasks

D. Adaptivity in stateful parallel pattern

In this part, I do the experiments that adding the adaptivity to different degree of stateful(0%,1%,10%,20%), I record the experiment outputs and compare the average consume per 10 tasks with the number of works in each time point (the figure 10 to figure 13) ,which prove the correctness of the adaptive pattern in different degree of stateful. In the no stateful experiment, I set first 50 tasks are normal tasks which cost 4000 around per 10 tasks at 5 workers, 50~100 are heavy tasks, and 100~150 are light tasks .And I set the 5 works active at the begin, then if the consume larger than 5000 second, we need the adaptive, so the #workers add 1 every 10 tasks, if the consume less than 2000 second, that means we don't need the adaptive, so the #workers reduce 1 every task

And for the experiment tests with the adaptive with the 1%, 10, 20% stateful pattern, the experiment consume range is almost same as no state, but just a little difference:
No stateful:2000~5000
1% stateful:2000~5000
10% stateful:2000~6000
20% stateful:4000~10000

Figure 10 adaptivity with 0% stateful

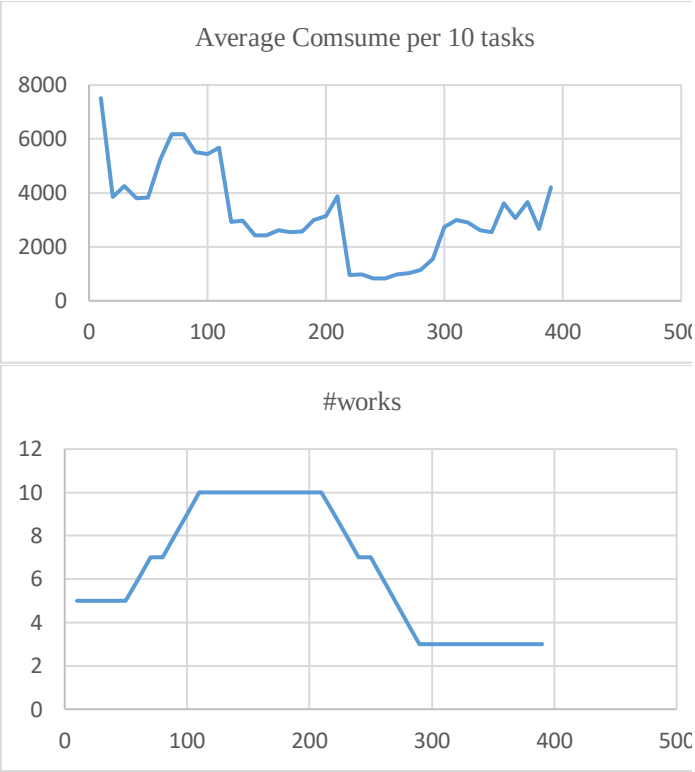

Figure 11 adaptivity with 1% stateful

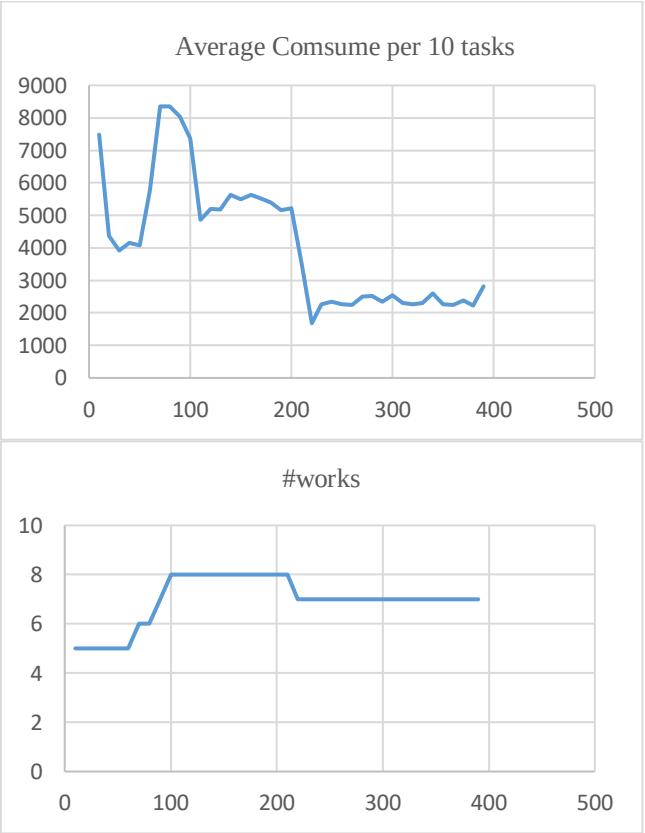

Figure 12 adaptivity with 10% stateful

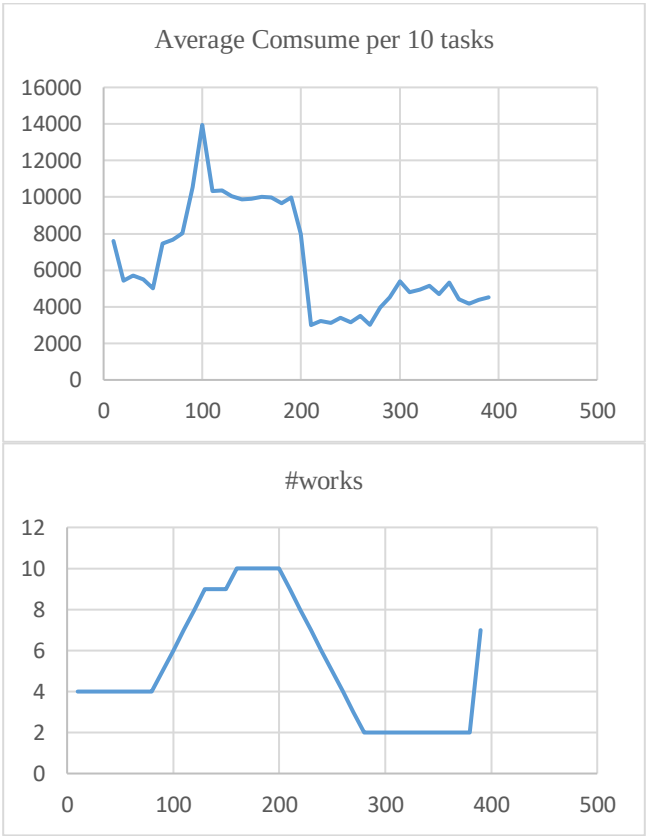

Figure 13 adaptivity with 20% stateful

From these figures, we can clearly see that when the consumption reaches the maximum value, the number of workers will increase, which will accelerate the execution and return the consumption to the range. Otherwise, when the consumption is lower than the minimum, we do not need so many workers, so the number of workers is will decrease. In the above experiments in different stateful, the consumption of the experiment was always kept within the expected range, which proves our adaptivity algorithm is almost correct. But in the high degree of stateful, it may need to used more threads to keep the adaptivity.

IV. Conclusion

In this paper, we mostly have presented an adaptive load-balancing task-scheduler, and applied it with different degree of stateful tasks to the Fastflow application. We have also done some experiment to test the overhead of Fastflow, and. We have benchmarked task-scheduler with the aim to assert 1) the overheads under conditions where it will not show performance improvements by design, and 2) the efficiency of our implementation comparing with the idealized model.

Our benchmarks prove the conclusion that Fastflow overheads are very small and that our implementation can reach the theoretically expected efficiency. However, the efficiency of 20% stateful with 16 thread was not very well in the experiment. Careful analysis of our implementation reveals that the cause for this behavior is the scheduler may need more time for scheduling the stateful tasks when using 16 or more threads, and therefore a high percentage of stateful may also not suitable for the adaptivity.

Future work will address the short-comings of our approach– it's possible that the adaptivity may not be very well with the high percentage stateful. We could solve this by improving the scheduling algorithm and may pay more attention to the scheduler's block size.

Appendix

The minutes of meeting:

1. Meeting called to order at 2:00 PM on 11/10/2018

the basic concept of parallel computing: If the workload is very large, we can broke down the calculation process in to small parts and solved in a concurrent way.

2. Meeting called to order at 2:00 PM on 23/10/2018

Analysis the reason of failure that running in parallel is taken much time than running in series. I didn't totally understand the pipeline pattern and farm pattern and write all the stuffs on first stage , and all the code running on first stage and just pass to the end so that it actually didn't do the parallel work.I need to think about how to break up a algorithm to many parts and write them to each stage.

3. Meeting called to order at 2:00 PM on 30/10/2018

talk about the ideal situation about 2 patterns and how to calculate their efficiency , whose the ideal situation is the running time on parallel should equals to the running time that total CPU time divide by number of CPU in serial .And also explain the reiterated again about basic process of 2 patterns that why farm pattern has more efficiency than pipeline pattern.

4. Meeting called to order at 15:30 PM on 07/11/2018

discussed how the lock will influence the test result and performance and could input some specific datas to check the implement correction

.

5. Meeting called to order at 16:00 PM on 14/11/2018

we talked about something about my last time implement on Fastflow: reiterated again the importance of making output data under a certain sequence, and the "feedback channel" in the task-farm pattern is a good way to implement this function.

6. Meeting called to order at 15:30 PM on 28/11/2018

we talked some possible idea to implement the feedback channel, such as using a flag to check the response. I may set a flag to decide whether the specific transaction could pass to the workers, and feedback from the collector can change the flag state.

7. Meeting called to order at 14:00 PM on 12/12/2018

we talked about something about my last time implement:Talked about the correctness of my implement, and it also points out that global variables may be mutually exclusive sometimes.

8. Meeting called to order at 14:00 PM on 19/12/2018

Need to make several experiments relating to the state access pattern aimed at demonstrating that the Fastflow program work successfully and that the performance results are as predicted.

9. Meeting called to order at 10:00 AM on 03/01/2019

Talked some detail about my experiment design: more than 8 threads cannot get more parallel speedup because my local machine only has 8 cores. And need to record a high-accuracy time by using a better timer on c++

10. Meeting called to order at 14:00 PM on 10/01/2019

Need to apply more [transactions] on my program and improve the experiment results format as a charts and tables.

11. Meeting called to order at 9:30 AM on 17/01/2019

only do the experiments with less than 4 workers due to the limit of 4 actual cores on my laptop.and the process time of stateful pattern and no stateless pattern should be same

12. Meeting called to order at 14:30 PM on 24/01/2019

Talked about the which combine of Fastflow can take more efficiency, it's also necessary to see if the speedup can be more than 4 in my laptop, which would be very helpful in my further experiment.

13. Meeting called to order at 17:00 PM on 31/01/2019

Although the experiment result is almost fit the ideal situation, it's also may be some bug exist in my implement so it's also needs to find bug from code and do the experiment continually every week.

14. Meeting called to order at 14:00 PM on 14/02/2019

A way to implement the adaptive: we can think about a policy that the workers "ask" for a task to be computed rather than passively accepting tasks sent by the emitter according to some scheduling policy. And this scheduling behavior may be simply implemented by rewrite the method in the Fastflow.

REFERENCES

[1] Md M, Kamran I, Michael R,Jose G, *Adaptive Load-balancer for Task-scheduling Fastflow* INFOCOMP 2015 : The Fifth International Conference on Advanced Communications and Computation page 6-12

[2] Herb S ,*The Free Lunch Is Over A Fundamental Turn Toward Concurrency* in Software.Dr. Dobb's Journal, 30(3), March 2005

[3] Massimo T, Parallel Programming Using FastFlow,September 2015

[4] OSChina *The Fastflow 2.0 released* https://www.oschina.net/news/35597/fastflow-2-0 12th October 2012

[5] Marco D, Peter K, Gabriele M and Massimo T *access patterns in stream parallel computations* The International Journal of High Performance Computing Applications 2018, Vol. 32(6) 807–818

[6] Marco D, Peter K, Gabriele M and Massimo T *access patterns in stream parallel computations* The International Journal of High Performance Computing Applications 2018, Vol. 32(6) 807–818

[7] Aldinucci, M Massimo T, and Massimiliano M. *FastFlow: Efficient parallel streaming applications on multi-core.* arXiv preprint arXiv:0909.1187 (2009).

[8] Md M, Kamran I, Michael R,Jose G, *Adaptive Load-balancer for Task-scheduling Fastflow* INFOCOMP 2015 : The Fifth International Conference on Advanced Communications and Computation page 6-12

[9] Goli, Mehdi, John McCall, Christopher Brown, Vladimir Janjic, and Kevin Hammond. "Mapping parallel programs to heterogeneous CPU/GPU architectures using a monte carlo tree search." In Evolutionar Computation (CEC), 2013 IEEE Congress on, pp. 2932-2939. IEEE, 2013.

[10] Niethammer, Christoph, Colin W. Glass, and Jos´e Gracia. "Avoiding serialization effects in data/dependency aware task parallel algorithms for spatial decomposition." In Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, pp. 743-748. IEEE, 2012.