Several Parallel Computation Methods for Checkerboard Problem

Jiapei Yao

May 9, 2017

Abstract

The project processes the Checkerboard Problems with the combination of functional programming and several parallel computation methods including asynchronous, task based, and actor-based programming. Then the project analyse their performance on different conditions.

1 Introduction

The checkerboard problem is that given M, a matric of costs, which are non-negative number, what is the largest cost to reach the last line of the matric from the first line, while M/[i/]/[j/] can only go to M/[i+1/]/[j-1/], M/[i+1/]/[j/], or M/[i+1/]/[j+1/]. Using checkerboard as an example, the project contrast asynchronous, task based, and actor-based programming.

2 Algorithms

2.1 Sequential

Sequential implementation is the very basic algorithm without any parallelization. And the parallel algorithms below are based on the sequential algorithm. The algorithm uses the idea of the dynamic programming which stores the results of the sub-question to avoid redundant computation. The dynamic programming is more optimized than loop through all the possibility of the path. Starting from the first line, for each entity M[0]/[j/], the algorithm replace it by the largest value among his left neighbor, right neighbor, and itself. Then the consequent line is stored and pass and add to the next line. After i loops, the final consequent line stores the largest cost going from the first line to those corresponding entities. Then, the largest number of this line is the answer.

2.2 Naive Parallel

Based on the sequential algorithm above, this algorithm use .NET Parallel.For to implement the parallelization. In each inner loop, which is going from M/[i/]/[0/] to M/[i/]/[COL/], the "replacement" operations of all the entities can be done parallel. A temperary line is used to store the result after the replacement to get rid of the race condition. The addition operations of all entites can be computed parallel too.

2.3 .NET Parallel For with Range Partitions

Based on the naive parallel algprithm introduced above, this algorithm use range patitioner to seperate the processes of one line to several ranges. And inside the a range /[low, high), the algorithm process from M/[i/]/[low/] to M/[i/]/[high-1/] parallel with the .NET Parallel.For calls . An outer loop is used to loop through all the ranges.

2.4 Asynchronous with Range Partitions

This algorithm uses range patitioner to seperate the processes of one line to several ranges. And inside the a range /[low, high), the algorithm process from M/[i/]/[low/] to M/[i/]/[high-1/] parallel with the Asynchronous calls . An outer loop is used to loop through the all the ranges.

2.5 Mailbox Processor with Range Partitions

This algorithm is based on the actor model where each actor store and process data on their own, and communicate with other actors by sending and receive message. The algorithm is implemented by generate a number of mailbox processors who takes charge of all the work on a range of columns. The mailbox processors embrace a Asynchronous call that make the processes fo the mailbox processor parallel. When the borders of one mailbox processor need value from their neighbors who are in other mailbox processors, other mailbox processors sends the value to them, then the mailbox processor receives the messages and transfers them into the values it needs. And the mailbox processor also need to proactively send message to those who need its border values.

2.6 Akka actor with Range Partition

This algorithm is based on the actor model, too. The algorithm is implemented by generate a number of actors who takes charge of all the work on a range of columns. When the borders of one actor need value from their neighbors who are in other actors, other actors sends the value to them, then the actor receives the messages and transfers them into the values it needs. And the actor also needs to proactively send message to those who need its border values.

3 Experiments

With A sample input matrix of 100 columns * 3000 lines of a repeating small matrix, shown below. For each one setting, which includes one kind of algorithm and one partitioning number, the experiment tries 10 times and takes the average time cost. The result of the problem itself (not the performance or running time) is 300. The time cost of Sequential algorithm is about 29 ms.

$$\begin{bmatrix} 3 & 0 & 0 & 4 & 0 & 0 \\ 0 & 3 & 0 & 0 & 4 & 0 \\ 0 & 0 & 3 & 0 & 0 & 4 \\ 1 & 0 & 0 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 3 \end{bmatrix}$$

3.1 Naive Parallelization vs Range based Parallelization

The data collected is shown in the table below. As the data shown, the Naive parallelization is faster than Range based parallelization. Naive parallelization theoretically run all the processes in each inner loop parallel. The range partitioner seperate one inner loop into several partitions. Inside each partition, the algorithm runs the processes parallel, but it must loop through the partitions. Therefore, optimum number of partition ranges is 1, which is identical to the default partitioning.

Table 1: Naive Parallelization vs Range based Parallelization

Partitions	1	2	3	50
Naive Parallel (ms)	14.0	14.0	14.0	14.0
Range Based (ms)	29.5	33.6	34.9	73.3

3.2 Range based Parallelization vs Asynchronous Parallelization

The data collected is shown in the table below. As the data shown, the Asynchronous parallelization is slower than the range based algorithm. They have similar instructions because they are both based on similar model. The time that the Asynchronous method spends the most is very likely to be time of the async calls. The async calls may spend time on generating the asynchronous tasks. The optimum number of partition of async based algorithm is 2 as the time cast is less than that of 1 partition, and as the partition grows larger, the time cost increases again.

Table 2: Range based Parallelization vs Asynchronous Parallelization

Partitions	1	2	3	50
Async Based (ms)	746.2	684.3	715.9	744.0
Range Based (ms)	29.5	33.6	34.9	73.3

3.3 Mailbox Processor vs Akka Actor

The data collected is shown in the table below. The time cost of mailbox processor and the Akka Actor is very close no matter the partition number is small as 1 to 3 or as large as 50. The reason may be their model is similar, so implementing of the actor model require similar amount of functions to be called (in other word, similar complexity). The average cost seems to decrease while the partitioning number increases. However, after trying 50 as the partitioning number, the average cost of 50 is far larger than the time cost with small partitioning numbers. After trying some other numbers, 7 is found to be the critical partitioning number where the time cost increase significantly. From 1 to 6, the time costs of them are around 1525 ms with insignificant change, but when the partition number reach 7, the time cost increases to about 1730 ms.

Table 3: Mailbox Processor vs Akka Actor

Partitions	1	2	3	50
Mailbox (ms)	1529.1	1529.5	1522.8	7141.3
Akka Actor (ms)	1530.4	1529.9	1522.0	7149.3

4 Conclusion

4.1 What is wrong with the performance of the Parallelizations

As the sequantial algorithm have averrage time cost of only 29 ms. It is very hard to imagine some implementation that are even faster than 29 ms. And in the data collected in the experiments, only one algorithm, the naive parallelization with .NET Parallel For, is faster than the sequential one. The parallelization with both .NET Parallel For and range partition has a close running time to the sequential algorithm, but its time cost increase while the partitioning number increases. The Asynchronous, Mailbox Processor, and the Akka Actors method are far slower than the sequantial algorithm. After encoutering some errors while I'm programming the project, the disadvantages of using parallelization appear.

4.2 Generating Actors Requires Time

The first error encountered was after inputing a large partitioning number, the Akka Actor method will throw exception and dead letters appears a lot. The problem was found to be that the series of actors are generating with a loop, and if the loop is long, the actors that has been generate will start running. Some message will be send to an actor that hasn't been generated yet or will require receiving a message to continue its process. The solution that can solve the problem is using Thread.Sleep timeInMs to put it to sleep and wait until all the actor is generate. The naive way is put Thread.Sleep 10 or Thread.Sleep s with some s corresponding to the number of actors would be generated. Later, a better version is used to save time, which is using Thread.Sleep (NumOfActor - i)*(s'). With this instruction is put before the loop inside the actor begins, the actor that are generated earlier (so has less i) will wait longer. The number of Actor is the partitioning number input from the user, and the s is the average time that one actor need to be generated. I assume it to be 1 ms.

4.3 Race Condition

The second problem is that after solving the first problem, the two algorithms that are based on the actor model sometimes had a result of 301. Since 301 is very close to 300, and comes out

random, I suppose the problem is the race condition. The more actor generated, or the larger size of the input, the more possibilities that the race conditions occur. Some sleep or wait instructions have to be used to decrease the possibilities of having some race condition. However, there are two problems. The first one is that it is hard to find out the exactly best time the actors need to sleep. The second one is that even if the sleeping or waiting time is small, the performance of the whole program can decrease a lot. In this case, if for each recursive loop inside an actor sleep for 1 ms, and there are 1000 lines in the input matrix, the 1 ms's sleep will be run 1000 times. Then the performance will decrease a lot. At least, it is impossible for it to be faster than 29 ms's sequential algorithm.

4.4 Going Back to Checkerboard

The more complexy parallelization the problem use, the more amount of time the problem have to cost to load the library, generate class instance, and etc. In the Checkboard problem case, the computaton inside one enttity is simple. one round of a comparison, a replacement, and an addition only requires a little bit amount of time, but spending far more time sending messages, receiving messages, verifying their correctness, and sleeping or waiting in the actor algorithms. Therefore, one problem that will run better parallel, may have computation that takes time. Then looping through all the computaion will take a lot of time, so it's better to compute those computations parallel.