COMP90025 Parallel and Multicore Computing

Project 1 - Multiple Sequence Alignment

Kaixun Yang (1040203)
The University of Melbourne
kaixuny@student.unimelb.edu.au

ABSTRACT

This report mainly introduces the parallel programming technology OpenMP to solve the Multiple Sequence Alignment problem. Firstly, the report briefly introduces the sequential algorithm used for optimization, the experimental platform and the data set. Secondly, the report introduces methods the author tried for this problem. The author uses the diagonalize tiled method, combined with thread affinity and memory allocation strategies, to experiment on the mseq-big13-example.dat data set on a Spartan node. Through experiments with different number of threads, placement strategies, affinity policies, the author finally proposed a method that can provide a speedup of 4.43 with 16 threads, spread affinity policy and cores placement strategy. In addition, the author also discussed the methods of poor performance and the analysis of the reasons for failure.

1 Introduction

1.1 Problem Review

Sequence alignment is a string-matching problem which provides two sequences of symbols. It is required to match the two strings by inserting gaps. The penalty value is calculated based on the product of the number of gaps inserted and the gap penalty value, the product of the number of mis-matched characters and the mis-matches penalty value. The problem requires us to minimize the penalty value and offer the corresponding two strings. K sequence alignment problem offers k strings, which needs to solve the sequence alignment problem between all k*(k-1)/2 sequence pairs.

1.2 Related Work

Rognes proposed the ParAlign algorithm, which is based on SIMD technology. The algorithm performs a computation of the exact optimal ungapped alignment score for all diagonals in the alignment matrix and use a novel heuristic to compute an approximate score of a gapped alignment by combining the scores of several diagonals, which is used to select the most interesting database sequences for a subsequent Smith–Waterman alignment parallelly[1]. Haidong presents parallel algorithms for protein sequence alignment based on the dynamic programming concept which can be efficiently mapped onto Xeon Phi clusters. The algorithm divides the problem into 3 modules: dispatcher used to divide the tasks into a number of chunks in preprocessing steps and distribute them to processing nodes, worker used to solve the chunks by dynamic programming method and collector used to collect all results from workers[2]. Therefore, there are two main directions here, one is the optimizing the sequence alignment algorithm, and the other is the distributing multiple matching tasks.

1.3 Sequential Algorithm

The sequential algorithm compares k sequences in pairs through a two-level loop. For each pair of sequences, the dynamic programming method is used. The two-dimensional dynamic programming array dp[i][j] represents the minimum penalty value between the substring of the first string (index from 0 to i) and the second string to the substring (index from 0 to j). For each dp[i][j], there are two situations: if the (i-1)th character of the first string is the same as the (j-1)th character of the second string, then dp[i][j] is equal to dp[i-1][j-1]; If the characters do not match, the penalty for the inserting gap into the first string , the penalty for the inserting gap into the second string and mis-matches penalty are required to be calculated, we will choose the smallest penalty case. The transfer function is as follow:

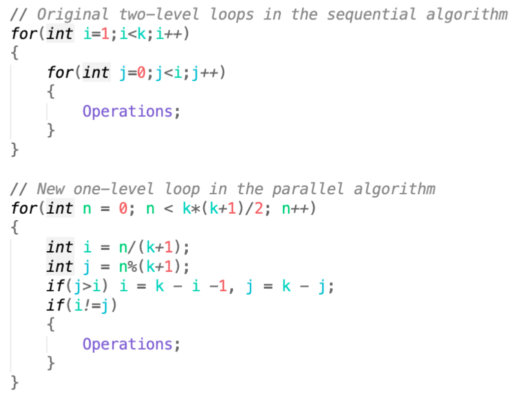
$$dp[i][j] = \begin{cases} dp[i-1][j-1] \quad if\ (s1[i-1] = s2[j-1]). \\ \min\left(dp[i-1][j-1] + mismatch\ penalty, dp[i-1][j] + gap\ penalty,\right. \\ \left. dp[i][j-1] + gap\ penalty\right) \quad otherwise. \end{cases}$$

1.4 Approach

Based on the two directions mentioned in 1.2, the author tried two methods.

1.4.1 *Parallel Processing Multiple Sequence Pairs*

Since the sequential algorithm uses two-level loops to execute the sequence alignment algorithm on different sequence pairs, there is no dependency between the execution of different sequence pairs, the author considers parallelizing this process. There is a correlation between the two-level loops, so it cannot use collapse(2) to deal with the nested loops. Therefore, the author converts the two-level loops into one-level loop.



```
// Original two-level loops in the sequential algorithm
for(int i=1;i<k;i++)
{
    for(int j=0;j<i;j++)
    {
        Operations;
    }
}

// New one-level loop in the parallel algorithm
for(int n = 0; n < k*(k+1)/2; n++)
{
    int i = n/(k+1);
    int j = n%(k+1);
    if(j>i) i = k - i -1, j = k - j;
    if(i!=j)
    {
        Operations;
    }
}
```

Figure 1: Comparison of two loop methods

Through the method of Figure 1, the author can use multiple threads to perform sequence alignment algorithm on multiple sequence pairs at the same time, and through the following formula to ensure that different array spaces are accessed by different threads, all results can be saved in the same array in order.

$$Array\ Index = (i-1)*i/2 + j$$

However, this method requires a lot of space overhead, because multiple sequence alignment algorithm will be performed parallelly, multiple dynamic programming arrays are required, and memory will be exceeded.

1.4.2 *Diagonalized Tiled Dynamic Programming*

The second method was inspired by the tutorial. Due to the dependency of dynamic programming as shown in Figure 2, the DP array can be divided multiple tiles, and multiple tiles on each diagonal can be executed in parallel.

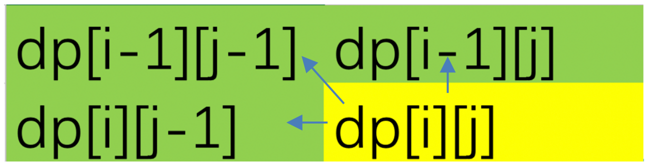


Figure 2: Dependency of dynamic programming

As shown in Figure 3, the tiles on the blue line will be processed in parallel and proceed in the direction of the red line in sequential. The height and width of the tile is equal to the height and width of the original DP array divided by the number of threads. In addition, thread affinity control is also considered to optimize the performance of the algorithm.

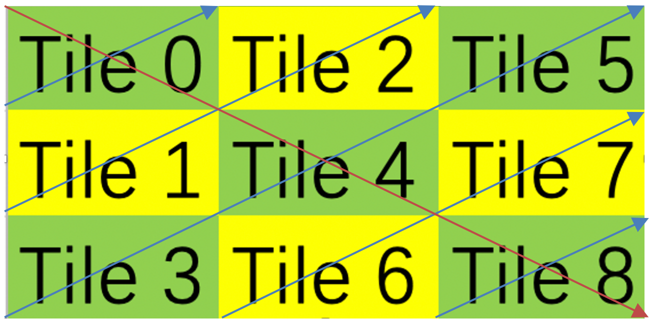


Figure 3: Diagonalized tiled dynamic programming

2 Experiments

All experiments are run on a multi-socket Spartan node (NUMA node), a High-Performance Computing system provided by the University of Melbourne. The data set for evaluating performance is mseq-big13-example.dat, which contains 13 sequences, and the length of each sequence ranges from 30,000 to 90,000. Because this data set will make the Parallel Processing Multiple Sequence Pairs method exceed the memory, the author also designed another small data set, containing 5 sequences, each sequence length is 10000. For each method, the author ran 5 times and calculated the mean to ensure the accuracy of the results. All methods are compiled with -O3 flag.

2.1 Experiments on the small data set

According to Table 2, the author found that the Parallel Processing Multiple Sequence Pairs method can get the best speedup, but this is not the final method, because it will cause the memory to exceed the limit when running on mseq-big13-example.dat. The author believes that in the sequential algorithm and Diagonalized Tiled Dynamic Programming algorithm, only one pair of sequences is compared at a time, so the DP array only needs to be created once, and this part of the memory will be released after the result is obtained, so the memory will not exceed the limit. However, for Parallel Processing Multiple Sequence Pairs algorithm, multiple pairs of sequences are calculated in parallel, it is necessary to create multiple DP Arrays, which will exceed the memory limit. Although Parallel Processing Multiple Sequence Pairs algorithm provides us with the best speedup, considering its memory consumption, we choose the Diagonalized Tiled Dynamic Programming algorithm for our follow-up experiments.

Method	Parameters	Time (us)	Speedup
Sequential Algorithm	None	3490219	1
Parallel Processing Multiple Sequence Pairs	8 threads affinity policy: spread placement strategy: sockets	**1452146**	**2.403**
Diagonalized Tiled Dynamic Programming	16 threads affinity policy: spread placement strategy: sockets	2777142	1.257

Table 1: Performance on small data sets

2.1 Experiments on mseq-big13-example.dat

In this part, the author focuses on the Diagonalized Tiled Dynamic Programming (DTDP) method, implemented by different number of threads, placement strategies and affinity policies. The author chooses the sequential algorithm as the baseline. The results are shown in Table 2.

Method	Threads Number	Affinity policy	Placement Strategy	Time(us)	Speed up
Baseline	None	None	None	1154365106	1
DTDP	16	spread	cores	336434048	3.43
DTDP	16	spread	threads	336530663	3.43
DTDP	16	spread	sockets	260755601	**4.43**
DTDP	16	close	cores	338557413	3.41
DTDP	16	close	threads	337262384	3.42
DTDP	16	close	sockets	261691633	4.41

DTDP	16	primary	cores	1098102809	1.05
DTDP	16	primary	threads	1116691148	1.03
DTDP	16	primary	sockets	374288610	3.08
DTDP	4	close	sockets	581989485	1.98
DTDP	8	close	sockets	410380469	2.81
DTDP	32	close	sockets	810775315	1.42
DTDP	4	spread	sockets	616535156	1.87
DTDP	8	spread	sockets	573508681	2.01
DTDP	32	spread	sockets	330945524	3.49

Table 2: Performance on mseq-big13-example.dat

Through Table 2, the author found that the number of threads is 16 can provide the best speedup. The author guesses that when the number of threads is too small, the number of DP tiles that can be processed in parallel each time is smaller, so the processing time will be longer. At the same time, the number of threads will also affect the height and width of the tile. The elements in the tile are executed sequentially. When the number of threads is small, there are more elements in each tile that need to be executed sequentially, and the more time it takes. When there are too many threads, the overhead is too large, even greater than the speedup brought by parallelism, so the speedup becomes lower. In the author's opinion, 16 threads are a trade-off between thread overhead and parallelism.

The author determines the places where the thread is assigned by setting the variable OMP_PLACES. In addition, the author decides how threads are bound to OpenMP places through proc_bind(). Through Table 2, the author found that when the affinity policy is close or spread, the speedup is obviously better than the speedup with affinity policy primary, while the performances of close and spread are similar. The author believes that spread and close arrange threads to different places, while primary concentrates all threads in a core or socket. Putting all threads in the same place may increase the computational load and obtain a bad speedup. Through Table 2, when the placement strategy is sockets, the speedup performance is better than other two strategies. The author believes that this may be related to the architecture of Spartan. A single machine on Spartan is a multi-socket NUMA node. By setting the place strategy to socket, each thread can be allocated to a different socket, which is more in line with the system architecture. When the place strategy is threads, the architecture needs to provide hardware threads, then each OpenMP thread can be tied to a specific hardware thread. The author believes that this may not conform to the Spartan node architecture.

Considering the experimental results and analysis, the author chose the Diagonalized Tiled Dynamic Programming method with 16 threads, spread affinity policy and cores placement strategy.

3 Conclusion

Through the problem analysis, the relevant literatures, and the experimental results, the author finally provides a parallel method to solve Multiple Sequence Alignment problem with a speed up of 4.43. In addition, the author also provides a feasible method to execute the calculation of multiple pairs of sequences in parallel, although this method will lead to out of memory. In subsequent research, the author may continue to think about how to solve the problem of insufficient memory.

Finally, the author believes that this is a very challenging project. Through this project, the author has a deeper understanding of parallel programming, especially OpenMP. The author hopes to get more feedbacks about the project which can help the author correct some incorrect understanding.

REFERENCES

[1] "ParAlign: a parallel sequence alignment algorithm for rapid and sensitive database searches." Nucleic Acids Research 29.7(2001):1647-1652. Conference Name:ACM Woodstock conference

[2] "Parallel algorithms for large-scale biological sequence alignment on Xeon-Phi based clusters." BMC Bioinformatics (2016).