parallel word analyzer

Yiwei Zhang
Chenbo Zhang
Shizhe Yang

Abstract

This lab is to find out the parallel algorithm for key word searching in a long complex text. This is not only applied in word searching inside the text, but also applied in the serial number or character in the complex marking system.

Task Management

Chenbo is responsible for constructing serial version code of the algorithm, analyzing the hashmap and AC-tree algorithm, and constructing the parallel version of the algorithm..
Shizhe is responsible for looking for the efficiency of MPI and ACC useful platforms.He managed to compare whether a parallel word analyzer could be faster on either platform.Also he is responsible for collecting data and putting them into a file for the program to run.
Yiwei is responsible for finding and do research on two ways of parallel algorithm and compare with other algorithm like kmp, and get conclusion of better performance on large data text input word searching system consider time complexity.

Introduction

Algorithm:

1. Hashmap word search:

The Algorithm then read in pattern line by line using `ifstream` and store them in a `unordered_map<string,int>` where `int` is the word count set initial to zero. Then it read the file to be search and use a `vector<string>` to store it. Then it use `#pragma omp parallel for` to loop the vector and add to `map[string]->second` to count numbers.

2. Aho Corasick Algorithm:

For practical situation, people seldom search only one keyword, most search algorithms are fast but can not deal with multiple pattern. So they need to go over the data multiple times for multiple patterns. Suppose there are M patterns of lengths L1, L2, …, Lm. We need to find all matches of patterns from a dictionary in a text of length N. A trivial solution would be taking any algorithm from the first part and running it M times. We have complexity of O(N +

L1 + N + L2 + … + N + Lm), i.e. O(M * N + L). We find Aho Corasick Algorithm can handle this situation with complexity O(N + L + Z), where Z is the count of matches.
Aho Corasick algorithm basically build a tree, where every node has a link to parent, a map to multiple children, a single character value, and a "redirect link", we will explain it later.

The core Idea is that, when matching failed at some character, instead of going over, it use "redirect link" to get to the longest possible substring (count from the last character of the pattern) , go that substring branch and continue.

For example, suppose we have pattern ATG, TGC, and GCA, we can construct a tree like this:

====R====

A===T===G

T===G===C

G===C===A

That R stand for Root. If we failed at ATX, that is, T(3,1) has no child X, we will follow its redirect link, which in this situation, will link to T(2,2). Also, if we fail at TGX, we go to G(1,3).

If any attempt gets to the bottom node of the tree then a pattern is matched.

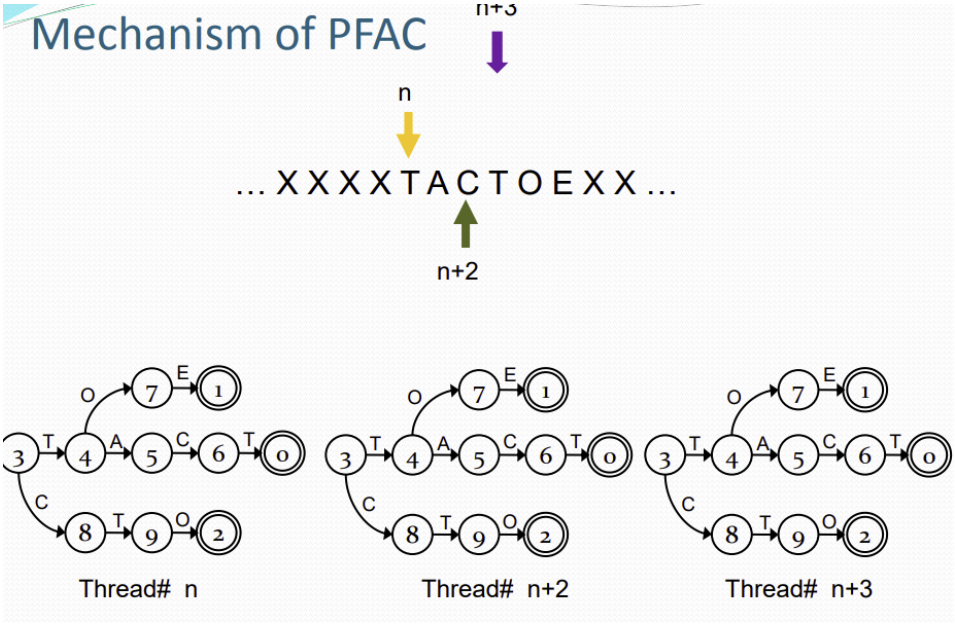
So, by going over this tree, we can find matched pattern by only scanning the data for one time.

parallelization:

There are two solutions for parallelization: The one is String cutting and matching cut text into p different part from the edge which used to cut text, forward and backward the max length of pattern string to form a collection of new searching part. use p+1 thread to analysis different text part.

The other one we called it PFAC algorithm(parallel failureless aho-corasick algorithm), Allocate each byte of input an individual thread to traverse a state machine. Remove all failure transitions as well as the sel floop transitions backing to the initial state. It will give

minimum number of valid transitions and thread is terminated when no valid transitions, so quickly go into next thread to deal with.



The time complexity for two ways is shown below: data parallel ac algorithm is n+m*s and pfac parallel algorithm is n*m(n is size of input text, m is longest key word, s is the number of part cutting the input text). So if the key word is a long word, pfac will not be prefer.

Implementation and Testing

Data structure:

The class CAhoTree in is defined in AhoCorasickAlg.hpp, with properties and functions below:

Private:

1. Structure SNode:
    i. nId int, the id of the node;
    ii. nValue char, the value for matching;
    iii. nHeight int, the height of node on the tree;
    iv. pParent pointer, point to parent node;
    v. pRedirect pointer, point to the redirect node, if the matching fails, go this way for next match;
    vi. mapChild map<char, SNode*> link parent to their children;
    vii. bEndNode decide if the node is a end of a matching pattern;

Public:

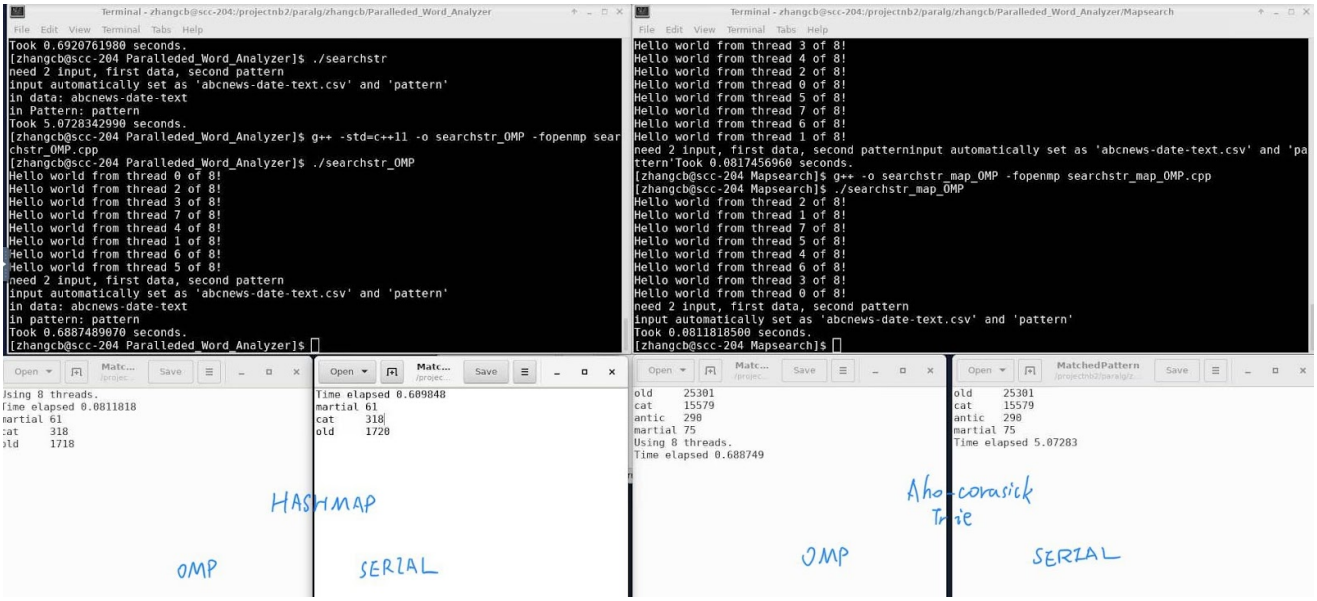
1. Construct method CAhoTree and destroy method ~CAhoTree;
2. AddPattern
    i. from root, walk the tree by the character of the pattern, if walk into a dead end, build a branch using what is left in input string, the last character of the string is matched as bEndNode;
3. Redirecting
    i. go through every exist node with same key (every new node will be pushed into a local map, key is char nValue , value is the set of nodes with char nValue);
    ii. Try connect them, Call function ConnectRedirectNode;
4. MatchPattern
    i. Return if there exist any match, stops at first match, faster than searching method;
5. SearchPattern
    i. search all matches, find matched patterns, call GenerateOutput to return a output string, insert it in a map, return this map, key is matched pattern, value is this pattern's matched count;
6. SearchCount
    i. return the sum of matched times of all pattern;

Private:

1. NewNode
2. Called by function AddPattern;
3. create a new node pointer with initial value and push it into node map and node vector;
4. ConnectRedirectNode
5. Called by function Redirecting;
6. evaluate this 2 nodes and connect them if passed evaluation;
7. CaseConvert
    i. in case-not-sensitive situation, transfer upper case to lower case
8. GenerateOutput
    i. send a endnode in, go from it to the root and get the whole matched word, return it as a string.

And In these project, we use mpi to help with dealing parallel algorithm to speed up.

Conclusion



The OMP test is running on 8 threads.

The time usage of OMP hashmap search is 0.0811818, about 1/8 of serial version time 0.609848.

The time usage of OMP Aho-Corasick search is 0.688749, about 1/8 of serial version time 5.07283.

From above result, we can see that the parallel aho-corasick is much faster than serial aho-corasick algorithm.