Parallel Boggle

Project Proposal for Parallel Functional Programming

Khyber Sen, ks3343

11/22/2019

S	E	R	S
P	A	T	G
L	I	N	E
S	E	R	S

aspergers
integrals
strainers
.
.
.

1 Introduction

My proposed (individual) project is to create a parallel boggle solver. That is, given a dictionary and an $m \times n$ boggle board, find all the words in the board that are in the dictionary, where words can be formed by concatenating adjacent (including diagonally) letters without repetition of cells. For example, in the above example 4×4 board, the word "aspergers" can be found with the path

$$[A(1,1), S(0,0), P(1,0), E(0,1), R(0,2), G(1,3), E(2,3), R(3,2), S(3,3)] \tag{1}$$

In finding all the words in the board, I will also calculate each word's score and the maximum total score for the board, where the score is determined by the length of the word.

A simple DFS search algorithm (see below) can be used to find all the words, so for normal 4×4 boards, the board can be solved quite fast. But if we consider arbitrarily large $m \times n$ boards and/or larger, richer dictionaries, the search becomes much harder to the point where parallelization will be immensely helpful. There is not many data dependencies or communication needed when exploring different branches of the DFS tree, so the word finding algorithm should parallelize easily.

However, if the parallelization of the word finding is too trivial, I would like to solve a more interesting boggle problem: for a given board size, find the board with the maximum possible score. There are far too many possible boards, even for small sizes like 4×4, so this will be done by using a stochastic optimization technique like simulated annealing or a genetic algorithm, where the energy or fitness function is the maximum score of that board. This is the expensive part of the optimization, so we can limit it to just calculating the score and not storing the found words. It might be possible to calculate the score of a board partially by using a similar board, but this is highly nontrivial, so I won't consider this possible optimization for this project. Once an optimal candidate has been found, we can go back and find all the words. The above example board, for example, is the highest scoring 4×4 board I found using serial simulated annealing, with some of the longest words found listed below. Simulated annealing is a much more inherently serial algorithm than a genetic algorithm, however, so when I parallelize the optimization, I will use a genetic algorithm. I will check through benchmarking if it's better to parallelize the word finding or the stochastic optimization, because the word finding is more trivially parallelizable than the genetic algorithm, but it's generally more efficient to introduce the parallelism (since the number of cores is very limited) at the highest level.

2 Word Finding Algorithm

We can create a trie from the dictionary file, which will provide fast prefix lookups which we will need, and are easy to implement functionally, unlike a hash map, for example. Starting from a virtual root node whose neighbors are all the letters on the board and a starting empty string, we add a neighbors letter, and if the resulting string is a prefix in the trie dictionary, then we continue exploring from that letter, passing along the prefix trie for that letter. Then, for each of these neighboring letters, we recursively explore its diagonally and directly adjacent neighbors, and apply the same prefix test to determine if we should keep going. If a prefix is ever actually a whole word, then we locally (on the stack) record that we've found this word. We don't want to add this to a global found words set because we're going to be running this in parallel. Once the search finishes, as we recurse back up the search tree, we combine the found words into a set (we need to remove duplicate words), and then go through each of them and calculate their score, and then sum up all the scores. If we want to store the word's path as well, we can just keep track of the letter's locations along with the letter we're appending. Furthermore, we'll accumulate the strings and paths backwards, so that appending the next letter or location can be done in $O(1)$ time to a linked list.

We can introduce the parallelism at any point where we explore a letter's neighbors, since we've removed any inter-thread contention except for the final reduction (combining the found words into a set). We want to make sure each core has the same amount of work, so we can't just split the starting letters among the number of cores, because some letters will have a much smaller search tree and will finish early. Thus, we'll continue the search a bit deeper before parallelizing the remaining searches.