Robert McCaull

Bio 131 Final Project Report (Spring 2020)

My project was to modify one of the algorithms we learned in class (a greedy motif-finding algorithm) so that it would spread its work across several execution threads, rather than just a single one. In theory, by doing this, a computer could execute certain intensive parts of the algorithm in parallel to each other, allowing a computer with multiple processors to fully take advantage of its resources when executing the algorithm. This could potentially result in much faster computations, if many processors were available. In practice, there are complicating factors to achieving this, some of which I wasn't able to solve. However, I was able to produce a correct algorithm that (theoretically) achieves the goal of parallelization.

The biological problem that my project is directed towards is the problem of finding commonalities between different DNA sequences. If we have a set of sequences which all serve a similar function, and we have the ability to find common factors between them, we can narrow in on which parts of the sequences contribute towards the common function.

To solve this biological problem, we first solve a related computational problem, and then apply our solution to the particular case we're interested in. The related computational problem is as follows: given a list of strings and an integer k, we would like to come up with a list k-length substrings (called motifs or kmers), one for each string, which are as similar to each other as possible. By 'as similar as possible', we mean the ones which differ as little as possible from the consensus string they encode.

Searching through all possible lists of motifs to find the best one would take a prohibitively long time. To deal with this, we use a greedy algorithm which only considers a subset of the possible lists of motifs. The lists of motifs this algorithm outputs are not guaranteed to be perfect – such is the nature of a greedy algorithm – but they don't need to be perfect to be useful, and the gains we make in

reduced running time are considerable. My project is just an extension of this greedy algorithm, which is (theoretically) faster still, since it can be run concurrently on multiple processors.

The non-parallel greedy algorithm works by starting with a motif from the first input string, and then greedily adding the best motif (given the motifs that were already chosen) from each following string in turn, to build a full list of candidate motifs. It then repeats this process for each other motif in the first string. As it does this, it keeps track of which full list of motifs it has seen so far that was best, and at the end it returns that list.

My parallel algorithm follows almost exactly the same approach. It builds up the candidate lists of motifs with the same greedy process, and uses all the same starting points. The difference is that my algorithm parallelizes this greedy list-building process.

The parallel algorithm takes the same inputs as the non-parallel algorithm (a list of strings and an integer k), as well as an integer numThreads, which tells it how many parallel execution threads to create. It begins by dividing the kmers of the first string (the starting kmers) into numThreads even groups. For each of these groups, it creates a separate execution thread, in which it runs the greedy list-building process of the nonparallel algorithm. As these threads finish executing, they return the best list of motifs that they found using their share of the starting kmers. The main process compares these best efforts to each other, keeping track of the overall best one. Once all the threads finish executing, the main thread will have the best list of motifs reached from any of the starting kmers, and it will return that.

This whole process is not as simple as it sounds. With parallel programming, the implementation details are almost always far more complicated than a description of the algorithm, and that is certainly the case here. For example, the subthreads don't really just return their results when they finish executing – they have to use a special object called a 'pipe' to communicate their results to the main thread, which follows a special procedure to receive them.

The code for the both versions of the greedy motif-finding algorithm, as well as a set of tests, can be found here:

https://repl.it/@RobertMcCaull/Final-Project

The file ‘README.txt’ contains information about how the code is structured, how the tests work, and a few comments on the results of the project. The code itself contains documentation in the form of comments, which should specify how the parallel algorithm works in more detail. The readme also contains links to resources I used while writing the code.

In my view, the project had mixed results. As the readme discusses, the tests show that I was able to make a correct algorithm, meaning that my parallel version has the same outputs as the non-parallel version. I also believe that the parallel algorithm does in fact run its code in separate threads – my experiences debugging definitely seem to suggest that. I’m very happy with this. However, to my disappointment, the parallel algorithm does not in fact run faster than the non-parallel algorithm, and seems to take substantially more time to run when instructed to create more threads. In the readme, I discuss two possible reasons for this – overhead in creating the threads, and the fact that despite my best efforts my OS may be running all of the threads on the same processor. Ultimately, I don’t know what the problem is, but I would guess the second is more likely.

Anna: I’d be fine with you posting my report on the course webpage.