CMSC 441 Project 2:
Multithreaded RNA Secondary Structure

Alex McCaslin, Brendan Waters

Table of Contents

I. Algorithm Design

Given a sequence of RNA bases, this algorithm will determine an optimal RNA secondary structure which matches the maximal number of valid pairings in the sequence. This algorithm also uses openMP parallelization for optimization.

Description of the important variables throughout the algorithm:

- solution - Matrix which holds the largest number of valid pairs of each subsequence letters[i...j]
- solution[0][n-1] - The maximum number of pairs that can be matched on the entire sequence
- result - Solution for (i,j) pair on current iteration
- n - Refers to the length of the RNA sequence
- i - Refers to the beginning of the sub-array
- j - Refers to the end of the sub-array

Code:

```
1 int OPT(int n, char letters[], int **solutions){
2   // ***SECTION 1***
3   //preprocess the matrix
4   for(int i = 0; i < n; i++){
5     for(int j = 0; j < n; j++){
6       if(j-i <= 4){
7         solutions[i][j] = 0;
8       }else{
9         solutions[i][j] = -1;
10      }
11    }
12  }
13  // *** SECTION 2***
14  //The algorithm
15  #pragma omp parallel
16  {
17  int subresult = 0;
18  int subresult1 = 0;
19  int subresult2 = 0;
20  int result = 0;
21  //loop thru each next unsolved diagonal of the matrix
22  for(int diag = 4; diag <= n; diag++){
23  //compute entire diagonal in parallel
24    #pragma omp for
25    for(int offset = 0; offset <= n - diag; offset++){
26    //calculate i,j values of current iter
```

```
27    int i = n - diag - offset;
28    int j = n - offset - 1;
29    //find maximum # of pairs over subproblems of problem (i,j)
30    result = 0;
31      for(int t = j-5; t > i - 1; t--){
32        bool valid = validPair(letters[t], letters[j]);
33        if(!valid){
34          subresult = solutions[i][j-1];
35        }else{
36          //double check t-1 is in bounds of matrix
37          if(t-1 >= 0){
38            subresult1 = solutions[i][t-1];
39          }
40          else{
41            subresult1 = 0;
42          }
43          subresult2 = solutions[t+1][j-1];
44          subresult = subresult1 + subresult2 + 1;
45        }
46        //take max over sub results
47        if(subresult > result){
48          result = subresult;
49        }
50       }
51       solutions[i][j] = result;
52      }
53     }
54    } //end parallel section
55    //the answer is in the top right corner of the solutions matrix
56    return solutions[0][n-1];
57  }
```

Description:

Starting with the first section of code, we preprocess the matrix by inserting zeros into (i,j) pairs that cannot possibly have any valid line foldings. This way of setting up the matrix is valid because no pairing letters[i] and letters[j] can have j - i <= 4. Now that the matrix is preprocessed, we move onto the parallel algorithm.

On line 15 we use `#pragma omp parallel` to spawn a group of threads, the number of which is determined by the user (`export OMP_NUM_THREADS=<n>` in the terminal or using SLURM). From there until the end of the algorithm, all variables are private to each individual thread so we setup our result values and begin the iterative for loops. The first two for

loops allow us to loop through each entry in each diagonal of the matrix starting at the first diagonal that does not contain preprocessed entries, moving from the bottom-right to the top-left of the diagonal. Since each entry of a single diagonal can be computed in parallel we use `#pragma omp for` to parallelize the inner loop using the threads we spawned earlier.

It is important that the algorithm perform calculations diagonally because the calculation of each diagonal (i, j) pair requires the entries directly to the left and the entries directly below to have already been computed. Thus entries on a particular diagonal do not depend on each other but do depend on previous diagonals. Thus looping in the order as described above we avoid race conditions where one entry is computed before some entry which it is dependent upon, resulting in an incorrect solution matrix.

From the innermost for loop on line 31 the algorithm works as described for the non-parallelized version in project 1 by looping through each subproblem of the current (i, j) problem and taking the maximum over these subproblems to find the maximal folding on letters[i...j].

II. Algorithm Analysis

Serial Algorithm Runtime Analysis:

- Work = T_1 = runtime of the serial algorithm
- The first for loop iterates from 4 to n, therefore its length is n - 4.
    - Also, we know that the minimum value of diag is 4 and the maximum is n
- Since the second for loop iterates from 0 to n - diag. The maximum number of iterations of this loop occurs when diag is minimal, therefore the maximal length is n - 4
- The third for loop iterates from j - 5 to i where j = n - offset - 1 and i = n - diag - offset
    - So the number of times this loop runs is equal to
        - (j - 5) - i = (n - offset - 1 - 5) - (n - diag - offset) = diag - 6
    - Then the maximum length of this loop occurs when diag is at its max
    - Therefore the max length of this loop is n - 6
- ∴ $T_1 = O(n - 4) * O(n - 4) * O(n - 6) = O((n - 4)\ (n - 4)\ (n - 6)) = O(n^3)$

Parallel Algorithm Pseudocode:

```
//constant work
for diag = 4 to n
     parallel for offset = 0 to n - diag
          //constant work
          for new t = j - 5 to i
               //constant work
          //constant work
return solutions [0][n-1]
```

Parallel Algorithm Runtime Analysis:

Span = T_∞ = runtime with infinite processors

Since the parallel loop is within the outer O(n) diagonal loop, we have
T_∞ = O(n) * (cost of parallelization overhead + cost of iterations)

- Cost of parallelization overhead ≈ 2 lg(n) = Θ(lgn) with the divide and conquer approach
- The cost of all iterations of the parallelized loop is equal to the cost of the longest iteration
- ∴ Cost of iterations = max{iter(offset)} over offset from 0 to n- diag
- There is another loop within each iteration which we have already shown to take at most n - 6 iterations which is O(n) time
- ∴ max{iter(offset)} = max{O(1), ... , O(n)} = O(n)

∴ T_∞ = O(n) * (Θ(lgn) + O(n))
= O(n) * O(n)
= $O(n^2)$

Analysis of Parallelism:

Parallelism = $T_1 / T_\infty = O(n^3)/O(n^2) = O(n)$

III. Empirical Performance

Fig. 1 Empirical run-time with input size n on p cores

n	T_1	T_2	T_4	T_8	T_{16}
128	0.01438	0.00855	0.0046	0.002804	0.00117
256	0.067	0.041	0.022	0.014	0.0037
512	0.535	0.277	0.1427	0.077	0.024
1024	4.24	2.165	1.093	0.571	0.22
2048	34.36	17.15	8.65	4.46	1.75
2560	69.46	34	17.08	8.93	3.4
3072	123.17	61.07	31.13	16.17	5.946
3584	198.6	101.1	52.08	27.7	9.65
4096	300	155.59	80.1	42.6	14.45

The above table shows the actual runtime with an input of size n with p cores. In each case where we double the number of cores we see the runtime about cut in half (as shown in fig. 3). This is to be expected from our theoretical results since the parallelism is n and our p-values are much less than n. So each increase in p-values will yield a similar reduction in runtime. Below we have more ways to visualize the empirical data.

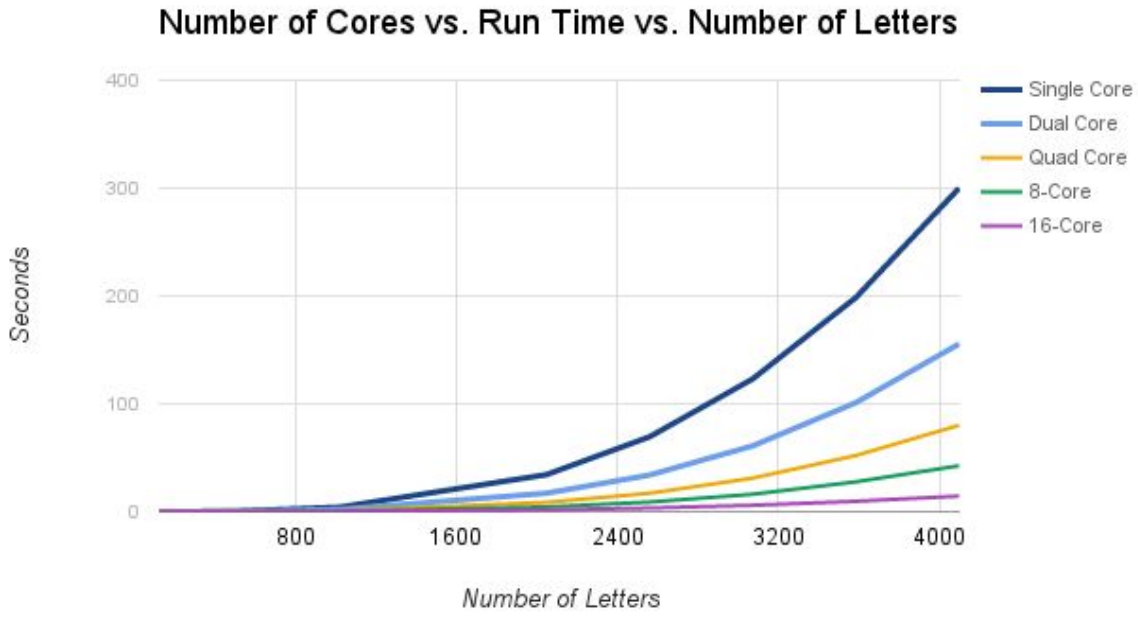

Fig. 2 This graph shows runtime as the input size increases for single and multi-core tests

Fig. 3 T_1/T_p for the varying number of processors based on actual results

n	T_1/T_1	T_1/T_2	T_1/T_4	T_1/T_8	T_1/T_{16}
128	1	1.68	3.126	5.128	12.29
256	1	1.634	3.045	4.78	18.108
512	1	1.93	3.749	6.95	22.292
1024	1	1.958	3.88	7.43	19.27
2048	1	2.003	3.972	7.704	19.63
2560	1	2.043	4.066	7.778	20.429
3072	1	2.0169	3.957	7.617	20.715
3584	1	1.964	3.813	7.170	20.580
4096	1	1.938	3.745	7.042	20.761

We have also calculated curves of best fit for each number of cores using the least-square-fit approximations by plugging (n, t) pairs into wolfram alpha. We can see as the number of processors increases, the runtime approaches $O(n^2)$down from $O(n^3)$. Taking a look at the coefficients, we see that where T = 1 the leading coefficient is $4.828 * 10^{-9}$ and for T = 4 the leading coefficient is now $1.554 * 10^{-9}$. Thus the coefficients on the n^3 term approach zero, at which point the term vanishes leaving us with a second order polynomial, or $O(n^2)$ runtime.

Fig. 4. Curve of Best Fit

T	Curve of Best-Fit
1	$T_1 = 4.828 * 10^{-9}n^3 - 2.14 * 10^{-6}n^2 + .00109n - .0760$
2	$T_2 = 1.9205 * 10^{-9}n^3 - 8.789 * 10^{-7}n^2 + .00104n - .185$
4	$T_4 = 1.554 * 10^{-9}n^3 - 2.023 * 10^{-6}n^2 + .00185n - .325$
8	$T_8 = -1.263 * 10^{-6}n^2 + .00115n - .199$
16	$T_{16} = 1.377 * 10^{-6}n^2 - .00237n + .6521$

IV. Conclusion

We saw in part II that the theoretical runtime of the serial algorithm is $O(n^3)$ and the parallel runtime is $O(n^2)$. Looking at the data in part III we see as the number of cores p increases, the empirical run time approaches $O(n^2)$. In addition, the empirical speedup for a small number of cores ($p << n$) our algorithm achieves near-linear speedup. Thus we have designed a multithreaded algorithm which solves the RNA Secondary Structure problem in optimized time using parallelization.