Project 2: Multithreaded RNA Secondary Structure

Patrick Custer
Lee Brown

CMSC 441: Design and Analysis of Algorithms
Dr. Christopher Marron
November 18, 2016

Abstract

In this paper, we describe and analyze the theoretical and empirical work of a RNA folding algorithm. We then determine its theoretical work, span, parallelism, and parallel slackness. We compare this theoretical information with empirical data that we gathered from the UMBC HPCF system "Maya". We then analyze and discuss the difference between the theoretical and empirical data.

1. Background

The RNA folding problem is a classic dynamic programming that involved finding the secondary structure of an RNA strand. Ribonucleic acid, or RNA for short, is a molecule that is important for coding and decoding genes inside of the body. These strands are made up of the bases guanine (G), uracil (U), adenine (A), and cytosine (C). These bases can only match up in specific pairings. C can only pair with G, and vice versa. A can only pair with U, and vice versa. This is very similar to the Glen Burnie COMIC-CON problem that we attempted to find a solution to in Project 1.

2. Algorithm Description

2.1. Pseudocode

```
procedure OPT(r, line, n)
    for j = 1 to n − 1 do
        b = line[j]
        parallel for i = 0 to j − 1 do
            best = max(r[i][j], r[i][j − 1])
            for k = i to j − 5 do
                if line[k] and b are a match then
                    new = max(r[i][j], r[i][k − 1] + r[k + 1][j − 1] + 1)
                    if new > best then
                        best = new
                    end if
                end if
            end for
        end parallel for
    end for
end procedure
```

2.2. Algorithm Description

The Algorithm used in this project is a multithreaded implementation of the RNA substructure algorithm outlined in our Project 1 paper. The algorithm has shifted slightly from our recursive implementation previously used. To begin we will consider how the variables in our iterative code relate to the recursive code.

- b - In the recursive algorithm, this is represented by the 'j' variable passed into our OPT(i, j) calls, which acts as as the 'end' of our current substring. In our serial algorithm, this refers to the same.
- best, new - These are used in our serial algorithm in order to avoid race conditions. In the recursive implementation, they are equivalent to the recursive OPT(o, j) calls.
- r - This is a lookup matrix for previously solved sub problems. In our recursive implementation, this is filled in via a binary crawl type of motion as the recursive calls find subproblems to solve. The exact method of updating this board in our parallel code is described below.

In order to parallelise the code, it was necessary to create r in such a way that the new values depend only on values that have already been determined. This was achieved by switching the order of the first two loops. This change guaranteed that we would fill in value of r by column instead of by row, and since new values of r will never be defined by another value in its column, only by those that precede it, we can be certain that the necessary values have already been calculated. With this knowledge, we parallelised the second loop, essentially parallelising the work done over each column.

3. Theoretical Analysis

3.1. Work

To determine the work of our algorithm as described in the textbook, we can look at it as if the algorithm has not been made parallel. Looking at the **for** loops in the pseudocode, we can see that they all see that the loops all run n minus some constant times in the worst case. Therefore,

$$T_1(n) = O\left(n^3\right) \tag{1}$$

3.2. Span

To determine the span of our algorithm, we must look at the recurrence where

$$max(interior) = O(n) \tag{2}$$

$$T_\infty(n) = O(n) * O(lg(n) + max(interior))) \tag{3}$$

$$T_\infty(n) = O(nlg(n)) + O\left(n^2\right) \tag{4}$$

The n^2 term dominates and we are left with

$$T_\infty(n) = O\left(n^2\right) \tag{5}$$

From this, we can determine that $T_\infty(n)$ is n^2

3.3. Parallelism and Parallel Slackness

We know that the parallelism can be found using the formula

$$parallelism = T_1(n)/T_\infty(n) \tag{6}$$

$$parallelism = \Theta(n) \tag{7}$$

$$parallelslackness = parallelism/P = \Theta(n)\,/P \tag{8}$$

3.4. Linear Speedup

$$T_p = O(n) * O(n/P) * O(n) = O\Big(n^3/P\Big) \tag{9}$$

$$speedup = T_1/T_P = O\Big(n^3/(n^3/p)\Big) = P \tag{10}$$

This looks like liner speedup, but is only achievable when n = p. Since n grows significantly, as shown in our examples, it is not impossible to attain linear speedup, but it cost prohibitive to do so.

4. Empirical Data

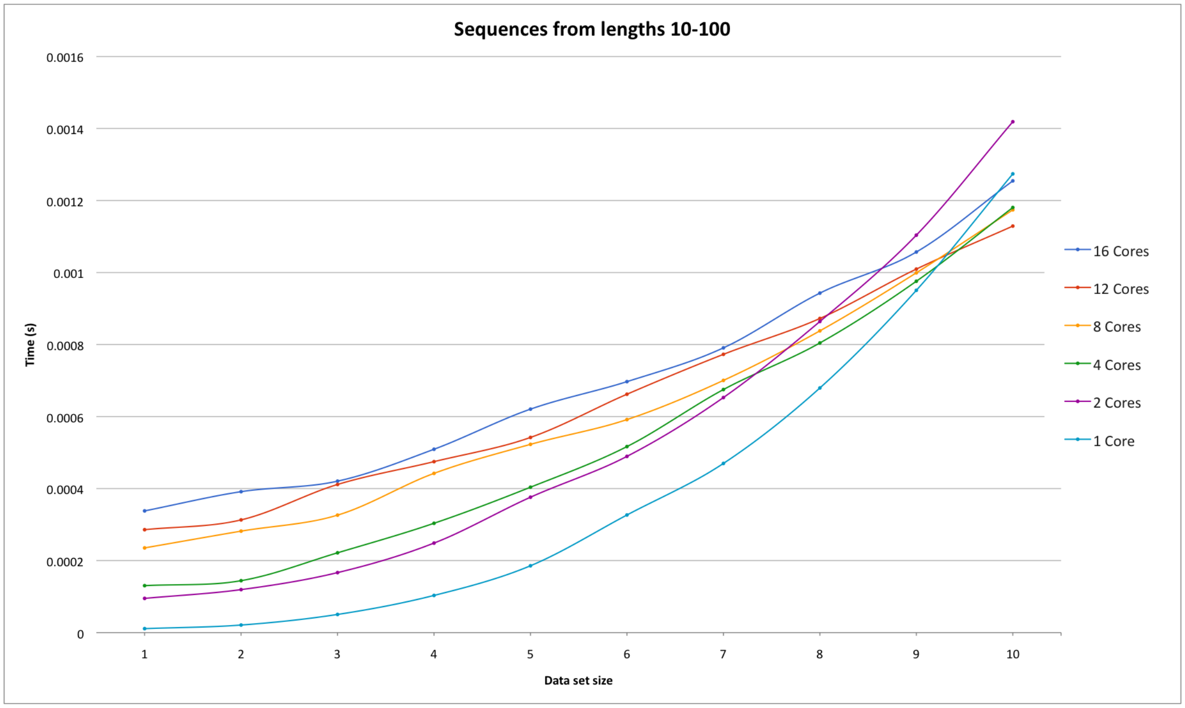
4.1. Average Times for Different n Sizes (vertical) and Number of Cores (horizontal)

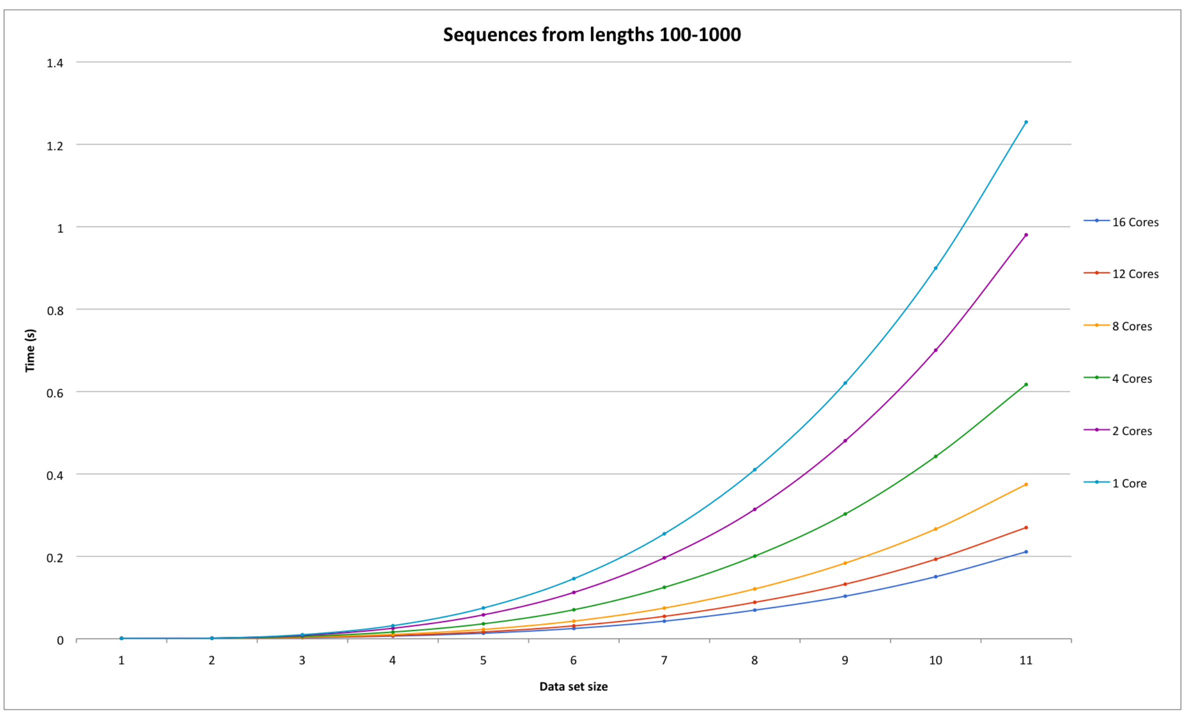
	16	12	8	4	2	1
10	3.38E-4	2.858E-4	2.352E-4	1.304E-4	9.5E-5	1.1E-5
20	3.914E-4	3.13E-4	2.818E-4	1.442E-4	1.194E-4	2.1E-5
30	4.206E-4	4.114E-4	3.262E-4	2.216E-4	1.666E-4	5.04E-5
40	5.092E-4	4.748E-4	4.422E-4	3.036E-4	2.486E-4	1.032E-4
50	6.206E-4	5.42E-4	5.228E-4	4.038E-4	3.76E-4	1.856E-4
60	6.97E-4	6.62E-4	5.916E-4	5.166E-4	4.894E-4	3.266E-4
70	7.908E-4	7.728E-4	7.004E-4	6.75E-4	6.528E-4	4.698E-4
80	9.428E-4	8.724E-4	8.38E-4	8.044E-4	8.64E-4	6.794E-4
90	0.001057	0.0010096	9.992E-4	9.758E-4	0.0011038	9.506E-4
100	0.0012546	0.001129	0.001174	0.0011806	0.0014188	0.001274
200	0.0028566	0.003364	0.0041468	0.0058138	0.0081044	0.0097624
300	0.0064614	0.0077646	0.0103484	0.0162458	0.025433	0.0316742
400	0.0133502	0.0165654	0.0226074	0.0362492	0.0579704	0.0746082
500	0.0248688	0.0312858	0.0427904	0.0703664	0.1123688	0.1458828
600	0.0426994	0.0543574	0.0744752	0.1247076	0.1962798	0.2547372
700	0.0694144	0.0884258	0.1209244	0.2005806	0.3140546	0.4103878
800	0.103312	0.132434	0.1835592	0.3027262	0.4805272	0.6206398
900	0.1505242	0.1929004	0.2663264	0.442513	0.70038	0.8995434
1000	0.2109972	0.2697602	0.3745338	0.617096	0.9802446	1.2540264
2000	2.0588464	2.6637476	3.7044036	6.2169482	9.7703296	12.1720754
3000	7.8256028	10.073538	14.0397696	23.9133478	38.0005694	47.1517308
4000	19.5042958	25.051856	35.0916376	60.85821	97.9984904	122.3818838
5000	39.5915984	50.6091976	72.4611494	126.0403056	204.0596998	255.1679514
6000	71.9139006	91.7620446	130.2503802	227.558441	370.1582934	463.8263414
7000	117.9212504	150.2121566	213.358007	380.7063826	610.9940806	765.3033586
8000	178.9009772	230.8318962	326.1055382	581.9391694	939.8900666	1180.200582
9000	262.7539082	332.4356676	483.8734712	872.4421866	1391.454055	1728.827148
10000	366.0108026	459.7807468	665.981527	1196.199551	1914.876967	2420.820684

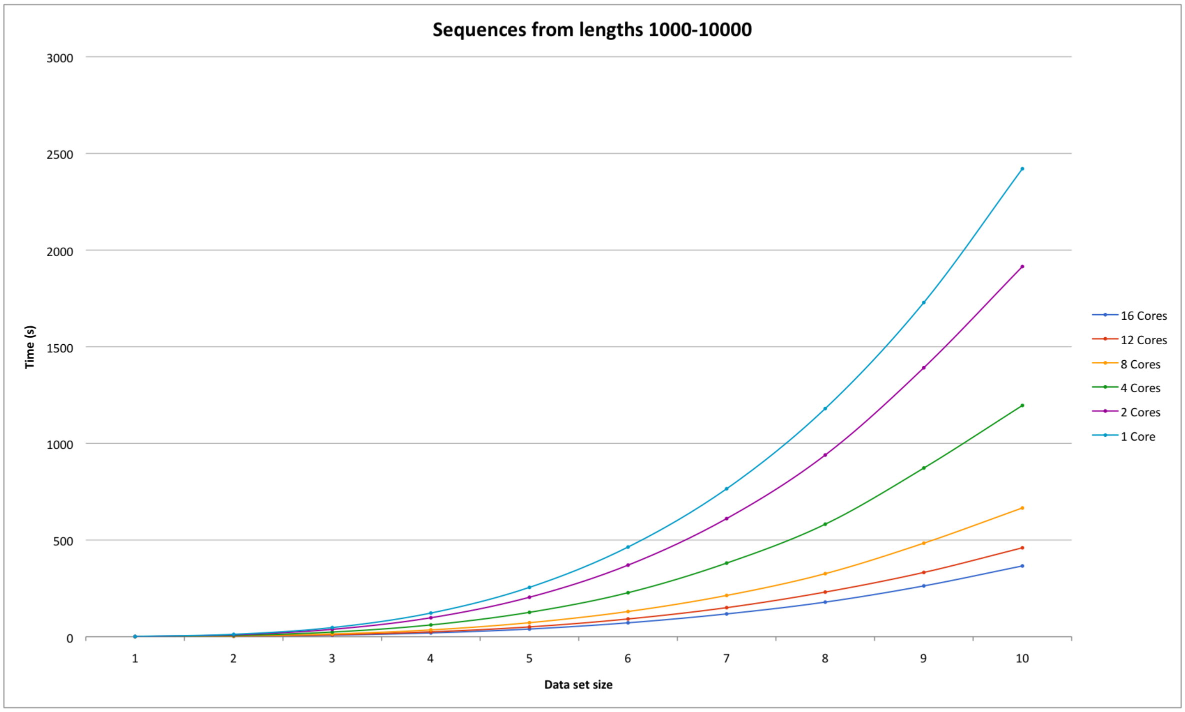
4.2. Running Times of an $O(1)$ Operation in an n^2 and n^3 Function With Various Sizes for n

	n^2	n^3
10	0.0000000	0.0000000
20	0.0000000	0.0000000
30	0.0000000	0.0000000
40	0.0000000	0.0000000
50	0.0000000	0.0000000
60	0.0000000	0.0000000
70	0.0000000	0.0000000
80	0.0000000	0.0000000
90	0.0000000	0.0000000
100	0.0000000	0.0000000
200	0.0000000	0.02
300	0.0000000	7.0000000000000007E-2
400	0.0000000	0.16
500	0.0000000	0.3
600	0.0000000	0.53
700	0.0000000	0.83
800	0.0000000	1.25
900	0.0000000	1.77
1000	0	2.4500000000000002
2000	0.01	19.53
3000	0.03	66.010000000000005
4000	0.04	156.24
5000	0.06	304.52
6000	0.09	601.04
7000	0.12	1212.08
8000	0.16	2435.16
9000	0.19	4852.32
10000	0.25	9696.7999999999993

4.3. Graphs of Average Running Times on Different n Values and Number of Cores







4.4. Analysis of Empirical Data

We can draw a few conclusions from the data presented in the table and the graphs. We can see that for lengths between 10 and 100, there isn't that much difference in the average runtimes over 5 test runs with the different numbers of cores. Surprisingly, however, we can see that 12 cores had a better runtime than 16 cores on average.

However, as we increase the n values to be greater than 100, the pattern is that the more cores, the better runtime on average. This shows the advantages of parallelising a program amongst many cores, as opposed to running it on one core.

5. Comparison of Theoretical Data and Empirical Data

When looking at our empirical results versus the theoretical data we came up with in Section 3, we can see that our calculations were relatively accurate. We can easily see that our algorithm took considerable advantage of the parallelism, completing operations on a large dataset well within the constraints of T_1.

6. Conclusion

What we have learned from these tests is that though the advantages of parallelization cannot be seen too well on small data sets, we can clearly see that on large values of n, there is considerable speedup when taking advantage of parallelization. On a problem of size $n = 10000$, we can see that parallelization with 16 cores can result in the process taking almost a fifth of the time.