2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum

Finding Common RNA Secondary Structures: A Case Study on the Dynamic Parallelization of a Data-driven Recurrence

Steven T. Stewart, Eric Aubanel and Patricia A. Evans
Faculty of Computer Science
University of New Brunswick, Fredericton, New Brunswick, Canada E3B 4A3
Email: {steven.t.stewart, aubanel, pevans} @unb.ca

***Abstract*—This paper presents the dynamic parallelization of a sequential algorithm for finding common RNA secondary structures that initially does not appear to be amenable to parallelization. A critical insight into the problem structure, which at first appears to be inherently top-down, leads to the development of a revised sequential algorithm that uses both bottom-up tabulation and top-down memoization. This novel combined approach proves well-suited for parallelization, overcoming the inherent difficulties in parallelizing the original top-down algorithm. The improved algorithm also eliminates two factors from the space complexity to fit into quadratic space, enabling the comparison of lengthy and complex RNA structures. Experimental results demonstrate that the parallel algorithm scales well, achieving speedup of up to 32X using 64 processors for contrived worst-case data containing structures having up to 1600 nested arcs. This algorithm illustrates the significant benefits that can be achieved by designing an underlying sequential dynamic programming algorithm with parallelizability in mind, instead of directly parallelizing an existing sequential algorithm. Our results also show the usefulness of combining both bottom-up and top-down perspectives when designing a parallel dynamic programming algorithm.**

***Keywords*-parallel dynamic programming; parallel algorithms; RNA structure comparison;**

I. Introduction

Dynamic programming is an algorithm design technique for discrete optimization problems such as scheduling, packing, and string comparison, with diverse applications in areas ranging from graph and network algorithms, compilers and parsers, bioinformatics and others. It is a powerful technique for optimization problems in which the optimal solution is composed of optimal solutions to subproblems [2]. Dynamic programming algorithms can be high in complexity and computational demands; therefore, implementations of these algorithms are often impractical for sequential machines. Parallel dynamic programming promises to tackle these difficulties, but great care must be given to the design of such algorithms.

Dynamic programming recurrences can be computed either by iteration ("bottom-up" approach) or recursion ("top-down" approach), and the principal idea is to store or memoize optimal results in order to avoid unnecessarily repeating the computation of subproblems. The design of parallel dynamic programming algorithms is typically regarded as difficult, and this can be exacerbated when the recurrence unfolds in an unpredictable manner due to irregular dependencies. This occurs when the recursive terms reference the input to the problem (data-driven), and, for the bottom-up approach, can lead to a significant amount of wasted computation. Although for some recurrences the amount of wasted computation can be tolerable, for others, such as RNA secondary structure comparison [1], [3], a recursive top-down approach can be the most viable option. Unfortunately, when this occurs, effective parallelization is difficult to achieve due to the challenges in managing a distributed data structure for storing intermediate results, and the problem of determining the order in which to assign subproblems to processors in an irregular data access pattern. For such parallelization to be effective, how the computations are distributed must be determined at run-time, essentially making it a *dynamic parallelization*. Finding a good dynamic parallelization is also made more difficult by the memory needs of large dynamic programming problems.

In [8], a general approach to top-down dynamic programming for a shared memory parallel computer is presented, but their approach is not suitable for distributed memory computers, which have the available memory needed to compute particularly demanding dynamic programming formulations. Additionally, recent work [7] has tackled this problem for heterogenous computing environments by developing a manager-worker approach in which workers are responsible for task creation and a manager handles dynamic load-balancing; however, although its memory usage is highly scalable, the approach is still essentially top-down and speedup is limited.

In this paper, we describe a parallel dynamic programming algorithm for RNA secondary structure comparison, a problem that initially does not appear to be amenable to parallelization due to its greater suitability for the top-down approach over the bottom-up approach. A parallel algorithm would be highly desirable, because the $\Theta(n^4)$ space complexity is impractical for reasonable problem sizes, but a straightforward parallel implementation is not immediately obvious, and the existing sequential algorithms either have significant wasted computation [1] or do not parallelize well due to top-down organization [3], [7]. This changes when a critical problem insight emerges, leading to a new perspective on how the RNA structures are traversed – instead of decomposing the problem strictly into individual subproblems, two-dimensional "child" slices of the dynamic programming table, consisting of groups of subproblems, are treated as independent problem instances that can be solved recursively, but are themselves computed in a bottom-up manner. Additionally, a computation order is determined that permits child slices to be discarded, reducing the space complexity from $\Theta(n^4)$ to $\Theta(n^2)$, and leading to a sequential algorithm with a flavour of both top-down and bottom-up approaches. This suggests a new approach to consider when parallelizing data-driven recurrences, and the redesigned sequential algorithm then proves to be readily amenable to a coarse-grained parallelization by computing child slices in parallel.

The rest of this paper is organized as follows. In Section II, we introduce background information and related work pertaining to parallel dynamic programming and the problem of finding common RNA secondary structures. In Section III, we decompose the dynamic programming formulation into its respective cases. In Section IV, we introduce our new sequential algorithm that combines both the top-down and bottom-up approaches, and we then improve upon this sequential algorithm by reducing its overhead, exploiting additional insights into the computation order. In Section V, we parallelize the sequential algorithm based on the insights acquired in developing the sequential algorithms, and the experimental results are presented in Section VI. Conclusions about the completed work and its implications for similar problems are discussed in Section VII.

II. Background and Related Work

For computer scientists, dynamic programming refers to an algorithm-design technique for solving discrete optimization problems with an underlying problem structure that satisfies Bellman's principle of optimality [2]. The solution space is implicitly explored by breaking the problem down into a series of subproblems and then building up optimal solutions to larger subproblems based on optimal solutions to smaller subproblems [6], which suggests a bottom-up approach to formulating a recurrence equation. Dynamic programming recurrences are typically implemented by iteratively tabulating optimal solutions to subproblems beginning with the base case(s), and building upon previously computed solutions towards the optimal result.

The effective parallelization of complex dynamic programming formulations is difficult to achieve, and so the use of a representative model or classification scheme would be very helpful, although (to date) there still does not exist a parallelization framework for the domain of possible dynamic programming recurrences. Grama *et al.* [5] proposed a classification scheme in which a dynamic programming recurrence can be broadly classified based on the number of recursive terms for alternate solutions and the number of levels separating dependencies among subproblems. The recurrence equation can be classified as serial monadic, non-serial monadic, serial polyadic, and non-serial polyadic. The non-serial polyadic class is considered the most difficult to parallelize, and this is further complicated when there is a data-driven characteristic, with the recurrence cases based in part on the input values themselves, which is exactly the case for the problem of finding common RNA secondary structures. With subproblem dependencies based on the input, an appropriate distribution of the computation between different processors cannot effectively be determined until run-time, and this issue must be dealt with when designing a parallel algorithm.

The serial and non-serial characteristics hint at the lifespan of tabulated or memoized results when computing a recurrence. The serial characteristic indicates that subproblems at one level in the dependency graph are dependent on subproblems that are at most one level away, suggesting that a dimension (or dimensions) of the dynamic programming table can be discarded. The non-serial characteristic implies that dependencies between subproblems cross many levels, requiring that a greater portion of the table remains active, or suggests that, when combined with the data-driven characteristic, the lifespan of tabulated results is either difficult to predict or unpredictable. For this reason, recurrences having the non-serial characteristic are considered more "difficult" than those having the serial characteristic. This increased difficulty emerges both in terms of a greater memory burden and an increased likelihood of interprocessor communication across table boundaries when dependencies exist on multiple processors over non-contiguous portions of the distributed table.

The conventional approach to parallelizing dynamic programming recurrences is to use the bottom-up strategy [5]. Parallelization is achieved by distributing the dynamic programming table among processors, each of which is responsible for computing the subproblems belonging to its portion of the table. The success of this approach greatly depends upon how well tasks are mapped to processors, such that interprocessor communication is minimized and processors are kept as busy as possible. This approach is in contrast to adopting a top-down strategy that implements the recurrence directly and uses memoization to avoid unnecessarily recomputing results. In the presence of the data-driven characteristic, the difference between the two approaches is that the bottom-up approach ignores features of the problem and input structure and simply fills the table with results, which leads to overtabulation; namely, subproblems are computed that in no way contribute to the final result. The top-down approach, although subject to additional overhead due to repeated memoization lookups and recursion, has the advantage of performing an exact tabulation because only those subproblems that contribute to the final solution are visited in the depth-first traversal of the dependency graph.

The top-down strategy and memoization are not typically adopted for parallel dynamic programs because of the difficulties in managing a shared memoization table and in determining how to assign the computation of the recursive terms to processors when adopting a depth-first traversal. That said, a recent paper [8] describes a general approach to parallel dynamic programming that can be used for a shared memory parallel computer. A shared data structure is used for memoizing the results of computed subproblems, and each processor calls the dynamic programming recurrence using identical parameters for the targeted outcome. The objective function, which is a maximization or minimization of the available choices, is applied at each stage of the process in the top-down manner. Parallelism arises by randomizing the order in which subproblems are computed, essentially sending each processor down a different path in the decision structure. The more divergence among the parallel threads, the greater the amount of parallelism, but this approach does not appear to scale well, because as the number of processors increases, so, too, does the likelihood of multiple processors following identical paths.

Other recent work has explored the difficulties inherent in parallelizing dynamic programming recurrences of the non-serial polyadic classification, including [3] and [7] who investigate the problem of finding common RNA secondary structures that permit pseudoknots, requiring both four- and eight-dimensional dynamic programming tables. The extremely large tables and the mutual dependencies between independent segments make the use of the bottom-up approach impractical, and so a top-down approach is adopted that selectively allocates portions of the table on demand. This approach is effective in reducing the amount of memory needed to compute the data-driven recurrence, although there is still a substantial accumulation of overhead due to dynamic memory allocation and memoization, and the approach is quite complex. The research discussed in this paper resolves these shortcomings for a more restricted model that requires only the portion of the problem structure associated with the four-dimensional table.

Ultimately, both the bottom-up and top-down approaches to dynamic programming have significant drawbacks for producing an efficient parallel algorithm. For data-driven recurrences, where the recurrence indices and subproblem dependencies are affected by the input, the bottom-up sequential algorithm itself inefficiently requires unnecessary computations and space. A top-down strategy, on the other hand, requires significant overhead both in interprocessor communication and in distributing the subproblems. In our work, the complexity and additional overhead needed to parallelize the top-down algorithm is eliminated by redesigning the underlying algorithm to use a combined bottom-up and top-down approach; the use of a top-down perspective in designing the algorithm eliminates the wasted space and computations that would be used by a strictly bottom-up algorithm, while still allowing for a straightforward parallelization.

III. Finding Common RNA Secondary Structures

A. Problem Formulation

A single-stranded RNA molecule is a sequence of bases over the four-letter alphabet $\{A, C, G, U\}$ [6], [7]. The RNA secondary structures can be viewed as a set of ordered sequence positions, using arcs between positions to represent bases that are bonded. A bond structure X is said to be a substructure of bond structure Y if the positions of X can be mapped onto positions of Y while preserving both the sequence order and the bonds. The basic problem is that of finding the Maximum Common Ordered Substructure (MCOS) between RNA sequences [3].

The MCOS problem can be formulated as follows, where the objective is to maximize the length of the common substructure [3]:

Input: structures S_1 and S_2.
where S_1 is the arc structure for a sequence of n positions and S_2 is the arc structure for a sequence of m positions, so $S_1 \subseteq \{1,\ldots,n\} \times \{1,\ldots,n\}$ and
$S_2 \subseteq \{1,\ldots,m\} \times \{1,\ldots,m\}$.

Output: substructure S_c, maximizing $|S_c|$
where $S_c \subseteq \{1,\ldots,n_c\} \times \{1,\ldots,n_c\}$ for some positive integer n_c, such that: $\exists$ one-to-one functions
$f_1 : \{1,\ldots,n_c\} \to \{1,\ldots,n\}$ and
$f_2 : \{1,\ldots,n_c\} \to \{1,\ldots,m\}$ where $\forall i,j \in \{1,\ldots,n_c\}$,
$i < j$ if and only if $f_1(i) < f_1(j)$ and $f_2(i) < f_2(j)$ and
if $(i,j) \in S_c$, then $(f_1(i), f_1(j)) \in S_1$
and $(f_2(i), f_2(j)) \in S_2$.

Although MCOS is NP-complete, polynomial time algorithms exist for solving this problem for restricted models that do not permit arcs to share endpoints or cross [1], [3].

The problem of finding common RNA secondary structures is concerned with the structures formed between bases in RNA chains. Figure 1 presents an example structure, illustrating how arcs link the bases of the underlying RNA sequences, which are, respectively, of lengths n and m. The model used here is restricted to non-pseudoknot secondary structures, which allow each base to be linked at most once, and permits pairs of arcs to be either sequential ($(1,8)$ and $(9,18)$) or nested ($(0,19)$ and $(9,18)$).

B. Dynamic Programming Formulation

The dynamic programming formulation presented by Bafna et al [1], which computes similarity between two RNA secondary structures by aligning the sequences with respect to their common substructures, forms the basis on which our sequential and parallel algorithms for finding common

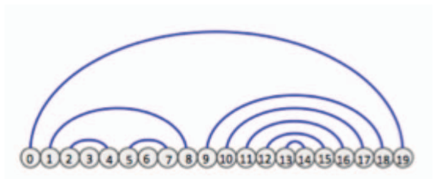


Figure 1. Example RNA secondary structure with positions labelled.

RNA secondary structures are devised. The problem is decomposed into the comparison of different intervals (or substructures) of the two sequences under consideration. We modify the Bafna formulation to reflect our goal of finding common RNA secondary structures (i.e., by counting matched arcs) as opposed to aligning RNA sequences, the latter of which takes into consideration both the bond structure and applies weight functions when comparing symbols in the alignment. The two modifications are: (1) the weight functions used by Bafna are removed; (2) Bafna also included an additional subproblem used to align interval endpoints without matching arcs, which is not required when weight functions are discarded.

The dynamic programming recurrence for finding common RNA secondary structures, which computes the exact solution to the MCOS problem for the non-pseudoknot model, is illustrated in Figure 2. The formulation is broken down into static dependencies and dynamic dependencies, the latter of which characterizes the recurrence as data-driven because these cases are only inspected when matched arcs are encountered while scanning the input, and they lead to irregularity in the data-dependency graph.

Common RNA secondary structures are identified in terms of the bond structure by taking into consideration both the order and relative position of arcs. For example, if one structure has three nested arcs followed by two nested arcs in an interval, and the other structure has two nested arcs followed by three nested arcs in an interval, then the maximum number of matched arcs, when comparing the two intervals and taking into consideration both order and structure, would be four. If the ordering of the two different-sized groups of nested arcs were identical, then the optimal solution would be five.

When devising the dynamic programming formulation of Figure 2, we define $F(i_1, j_1, i_2, j_2)$ as the maximum number of arcs of the common substructures when comparing interval (i_1, j_1) of the first structure with interval (i_2, j_2) of the second structure. There are two possibilities to consider, which are then further decomposed into specific cases.

The first possibility is that $(i_1, j_1) \notin S_1$ or $(i_2, j_2) \notin S_2$; in other words, either position j_1 does not correspond with an endpoint of an arc in S_1, or position j_2 does not correspond with an ending point of an arc in S_2. When this occurs, our count does not increase, and so it is necessary to obtain the maximum result over smaller instances of the

$$F[i_1, j_1, i_2, j_2] = max \begin{cases} F[i_1, j_1 - 1, i_2, j_2], \\ F[i_1, j_1, i_2, j_2 - 1] \end{cases}$$

if $\exists k_1, k_2$ s.t. $i_1 \leq k_1 < j_1, i_2 \leq k_2 < j_2$ and $(k_1, j_1) \in S_1, (k_2, j_2) \in S_2$ **then**

$$F[i_1, j_1, i_2, j_2] = max \begin{cases} F[i_1, j_1, i_2, j_2], \\ 1 + F[i_1, k_1 - 1, i_2, k_2 - 1] + \\ F[k_1 + 1, j_1 - 1, k_2 + 1, j_2 - 1] \end{cases}$$

end if

Figure 2. Dynamic programming formulation for finding common RNA secondary structures.

problem by recursively comparing intervals $(i_1, j_1 - 1)$ with (i_2, j_2) and (i_1, j_1) with $(i_2, j_2 - 1)$. This is achieved by computing the maximum of both $F(i_1, j_1 - 1, i_2, j_2)$ and $F(i_1, j_1, i_2, j_2 - 1)$. These two subproblems are referred to as static dependencies s_1 and s_2, which must always be inspected in order to account for arcs matched over the incrementally smaller intervals, regardless of whether or not an arc has been matched.

The second possibility is that $(i_1, j_1) \in S_1$ and $(i_2, j_2) \in S_2$). When this occurs, we have to consider the substructures both before and underneath the matched arcs because, as noted, our model permits only non-pseudoknot substructures, and so it is necessary to account for arcs that appear either in sequence or that are nested. Matched arcs can begin at position i_1 or later and i_2 or later, and before j_1 and j_2, and so variables $k_1 \geq i_1$ and $k_2 \geq i_2$ are introduced as the beginning endpoints of the matched arcs (k_1, j_1) and (k_2, j_2). Since these cases arise depending on the input to the problem, they are called *dynamic* dependencies and labelled with a d. Thus, we compare intervals $(i_1, k_1 - 1, i_2, k_2 - 1)$ to account for arcs that occur in sequence, and we compare intervals $(k_1 + 1, j_1 - 1)$ with $(k_2 + 1, j_2 - 1)$ to account for arcs that may be nested, and so the dynamic dependencies d_1 and d_2 are obtained by computing both $F(i_1, k_1 - 1, i_2, k_2 - 1)$ and $F(k_1 + 1, j_1 - 1, k_2 + 1, j_2 - 1)$.

Finally, $F(i_1, j_1, i_2, j_2)$ is computed by first obtaining the maximum of s_1 and s_2, which accounts for any matched arcs that were encountered over the incrementally smaller intervals, and then determining the maximum of that result with $1 + d_1 + d_2$ whenever a matched arc is encountered.

IV. Sequential Algorithms

Before delving into the discussion of the sequential algorithms, consider the dependency graph illustrated in Figure 3. A top-down traversal visits the subproblems (or nodes) in a depth-first order, which can be thought as unfolding the problem structure from top to bottom. As the unfolded structure is folded back, the results of subproblems are tabulated, leading to an exact tabulation. The difficulty of this approach is that there is no apparent way of knowing

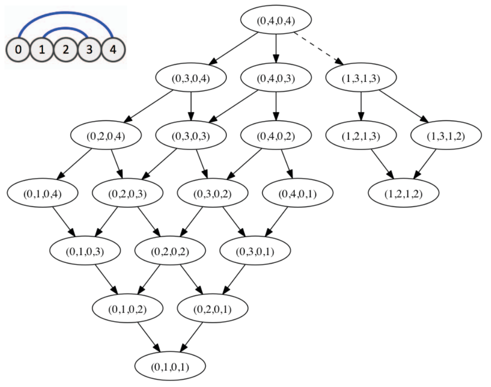


Figure 3. Dependency graph for the sequence (upper-left) aligned with itself. Top-down (or depth-first) traversal begins at node $(0,4,0,4)$, and results of subproblems are memoized to avoid following identical paths multiple times. Directed edges indicate the unfolded problem structure, and the dashed edge indicates when a matched arc is found.

how much of the memoization table to allocate in advance, and so the obvious choice is to allocate memory for all n^2m^2 possible subproblems. For most computers, it would not take long to exhaust available memory resources for practical problem sizes.

Next, consider solving the problem using the bottom-up strategy. To do so, the computation order must be defined such that we count matched arcs specifically in the manner in which the substructures are encountered, preserving both the order and structure. This computation order is referred to as the order of increasing interval widths. First, the structure of the smallest interval of the first sequence is compared with the structure of the smallest interval of the second sequence, anchored at positions i_1 of the first sequence and i_2 of the second sequence. Using these anchored beginning positions, the interval width is increased incrementally across the second sequence before then aligning the structure of the second smallest interval of the first sequence with that of the smallest interval of the second sequence, and so on; thus, when computing (i_1, n, i_2, m), subproblems are computed in the following order: $(i_1, i_1+1, i_2, i_2+1), (i_1, i_1+1, i_2, i_2+2), \ldots, (i_1, i_1+1, i_2, j_2), (i_1, i_1+2, i_2, i_2+1), (i_1, i_1+2, i_2, i_2+2), \ldots, (i_1, i_1+2, i_2, m), \ldots, (i_1, n, i_2, m-1), (i_1, n, i_2, m)$. The beginning positions (i_1 and i_2) only change when matched arcs are encountered, triggering the inspection of cases d_1 and d_2.

Unfortunately, the amount of overtabulation quickly accumulates when traversing a four-dimensional table, which can be extremely large for practical problem instances (i.e., sequences of lengths in the hundreds or low thousands), when no consideration is given to the actual structure of a problem instance. By slightly changing one's perspective on the problem structure, it turns out that it is possible to completely narrow the gap between overtabulation and exact tabulation, resulting in a much faster bottom-up algorithm and reducing the space complexity to $\Theta(nm)$. This is achieved by letting the input drive the computation, and we refer to this algorithm as SRNA1.

A. SRNA1

SRNA1 incorporates aspects of both the bottom-up and top-down strategies (i.e., both tabulation and memoization). A critical insight emerges when viewing the four-dimensional table in terms of two-dimensional slices, which can be expressed as (i_1, i_2) pairs (the indices for the beginning of the two intervals being aligned). In other words, for each of the possible (i_1, i_2) pairs, there is a corresponding two-dimensional slice of the table. Since the problem is solved by computing $F(0, n-1, 0, m-1)$, the $(0,0)$ pair is said to correspond with the parent slice, referred to as $slice_{0,0}$. All other (i_1, i_2) pairs are said to correspond with child slices.

Algorithm 1 SRNA1: Table M, Structures S_1 and S_2, Endpoints i_1 and j_1 of the first structure, Endpoints i_2 and j_2 of the second structure

```
Allocate memory for $slice_{i_1,i_2}$
**for** each arc $(k_1, x) \in S_1$ with $i_1 \le k_1 < x \le j_1$ (by increasing order of $x$) **do**
    **for** each arc $(k_2, y) \in S_2$ with $i_2 \le k_2 < y \le j_2$ (by increasing order of $y$) **do**
        $max \leftarrow$ MAX($slice_{i_1,i_2}[x-1][y], slice_{i_1,i_2}[x][y-1]$)
        $d_1 \leftarrow slice_{i_1,i_2}[k_1-1][k_2-1]$
        $d_2 \leftarrow M_{k_1+1,k_2+1}$
        **if** ($d_2$ is KEY_NOT_FOUND) **then**
            {Spawn child slice}
            $d_2 \leftarrow$ SRNA1($M, S_1, S_2, k_1+1, x-1, k_2+1, y-1$)
        **end if**
        $slice_{i_1,i_2}[x][y] \leftarrow$ MAX($max, 1 + d_1 + d_2$)
    **end for**
**end for**
$M_{i_1,i_2} \leftarrow slice_{i_1,i_2}[x][y]$ {$x$ and $y$ refer to the last arc end-points}
Deallocate memory for $slice_{i_1,i_2}$
Return $M_{i_1,i_2}$
```

If only subproblems s_1, s_2, and d_1 are considered, then there would be a single two-dimensional slice to be concerned with, $slice_{0,0}$; however, whenever a matched arc is encountered, subproblem d_2 requires the tabulation of $slice_{k_1+1,k_2+1}$. The top-down approach naturally uncovers the subproblems of child slices (for example, the subtree rooted at $(1,3,1,3)$ in Figure 3), but the overtabulated bottom-up implementation naively accounts for all possible matched arcs regardless of the actual problem structure. This shortcoming is resolved as follows: (1) tabulate $slice_{0,0}$ in a bottom-up manner based on the static dependencies and d_1; (2) tabulate the two-dimensional $slice_{k_1+1,k_2+1}$ only when a matched arc is encountered (see Algorithm 1).

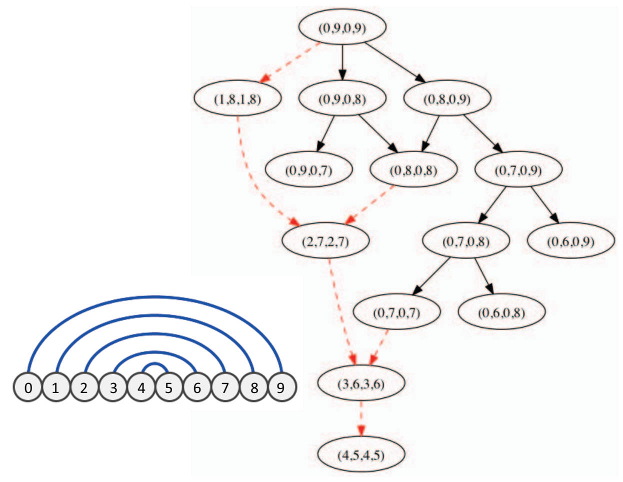
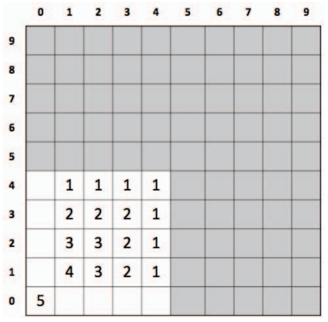


Figure 4. Partial dependency graph when self-comparing the depicted RNA secondary structure. Dashed lines indicate recursive calls to SRNA1 and the spawning of child slices. Multiple levels of recursive calls are possible, as indicated by following dashed paths of length greater than one.



	0	1	2	3	4	5	6	7	8	9
9										
8										
7										
6										
5										
4		1	1	1	1					
3		2	2	2	1					
2		3	3	2	1					
1		4	3	2	1					
0	5									

Figure 5. Example nested structure S and matrix M, where each position (i_1, i_2) in M contains the result obtained by tabulating $slice_{i_1,i_2}$.

Whenever a child slice is tabulated, it is said to have been *spawned*, meaning that memory is dynamically allocated for the two-dimensional slice of the table. By calling the function recursively, each child slice is tabulated in the identical manner of the parent slice. The result is that only those subproblems that appear in the dependency graph are visited in the bottom-up traversal, leading to an exact tabulation.

So far, this revised bottom-up approach has a glaring flaw – it is inefficient. The same child slice can potentially be spawned multiple times, amounting to a significant amount of redundancy in the computation. This occurs when a subproblem within a child slice corresponds with a matched arc and leads to an additional level of recursion. This is illustrated in Figure 4 in which the dashed lines indicate recursive calls to SRNA1 and the spawning of child slices by multiple levels of recursion. It would be wasteful to spawn child slices again and again for the inner arcs, which would have already been encountered earlier in the computation order. This is not dynamic programming at all, and so it is instead necessary to memoize the results of child slices so that they need only be computed one time.

It turns out that only the last tabulated subproblem of each two-dimensional child slice needs to be memoized, and the result of each child slice can thus be stored in a two-dimensional memoization table M. The reason for this is that any time we match nested arcs, the number of matched arcs underneath an arc between the two sequences, corresponding with the (i_1, i_2) pairs, never changes – we only need to know the final result and (unless we are interested in backtracing the subproblem that spawned the child slice) we do not need the details of how that last result was obtained.

The rows and columns of memoization table M (Figure 5) correspond with the values of i_1 and i_2 for the possible (i_1, i_2) pairs. This is illustrated for the self-comparison of a structure of the form depicted in Figure 4, which consists of a large group of nested arcs. In Figure 5, given $i'_1 > i_1$, $M(i_1, i_2)$ is dependent on $M(i'_1, i'_2)$ when, as depicted in Figure 4 by dashed lines, a subproblem in $slice_{i_1,i_2}$ leads to the spawning of $slice_{i'_1,i'_2}$. The bottom-up tabulation implies that $M(0, 0)$ is the last position memoized, ensuring that child slices need not be spawned again if they have already been memoized at an earlier point in the computation order.

SRNA1 solves the problem in $\Theta(n^2m^2)$ time, and does so having most of the advantages of using the bottom-up strategy (i.e., no overhead due to a lengthy chain of recursive calls, and a minimum of redundant visits to previously computed subproblems). Indeed, memoization generally is not associated with the bottom-up strategy and, in this regard, SRNA1 possesses an intriguing flavour of both the bottom-up and top-down strategies and combines both approaches to gain advantages of each. Additionally, by memoizing the result of the last computed subproblem in a child slice, it is guaranteed that the depth of recursive calls never exceeds one, and it is a certainty that if a particular row i_1 in M is dependent on values from another row i'_1, then the elements in row i'_1 would have been updated at an earlier point in the computation order. This naturally occurs because the most deeply nested arcs are encountered first. For example, if subproblem (1,8,1,8) (which corresponds with position $M(1, 1)$ in the memoization table) depends on subproblem (2,7,2,7) (which corresponds with position $M(2, 2)$ in the memoization table), then endpoints 7 of the first structure and 7 of the second structure are encountered before endpoints 8 and 8, implying that $M(2, 2)$ is updated before $M(1, 1)$. When implementing SRNA1, all that is required is to look up the (i_1, i_2) pair in the memoization table before deciding whether or not to spawn a child slice.

Given that the depth of recursive calls never exceeds one and that only the result of the last subproblem of a slice needs to be memoized, the original space complexity of

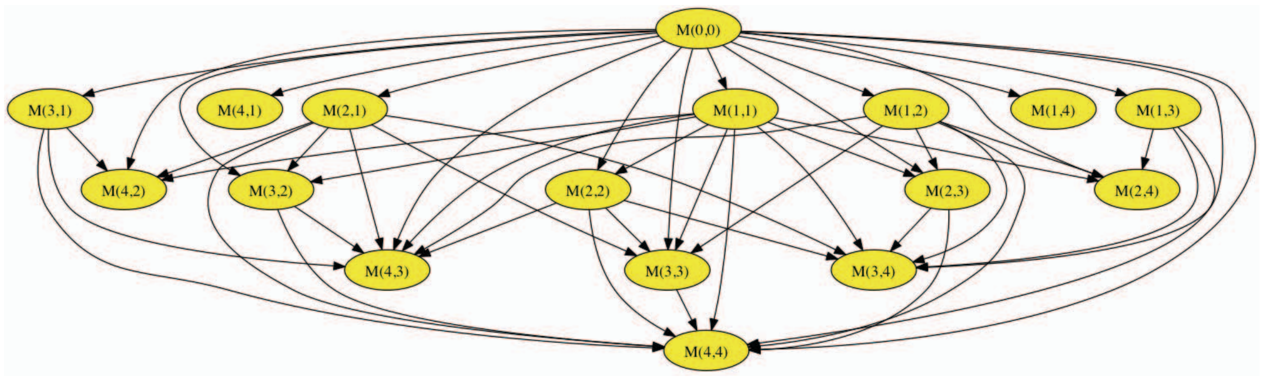


Figure 6. Dependency graph for the memoization table M when self-comparing the RNA secondary structure depicted in Figure 4 using algorithm SRNA2.

$\Theta(n^2m^2)$ can be reduced to $\Theta(nm)$.

B. *SRNA2*

The second sequential algorithm, which forms the basis for the parallel algorithm discussed in Section V, improves upon SRNA1 by eliminating overhead due to memoization lookups and removing the need for the recursive call to the function. As noted in Section IV-A, SRNA1 needs to retrieve $M(i_1, i_2)$ before deciding whether or not to spawn the child slice ($slice_{i_1,i_2}$), and both the conditional expression and the lookup occur within the inner loop of the computation whenever a matched arc is found. This implies that the amount of overhead accumulated by executing these statements is $\Theta(n^2m^2)$. The lookup expression returns KEY_NOT_FOUND whenever a value has not been previously memoized; SRNA2, instead, ensures that the lookup function is guaranteed to always return a memoized value, thereby eliminating the need for the conditional expression and the recursive spawning of a child slice. An auxiliary routine, called $TabulateSlice()$ (see Algorithm 2), is called to perform the bottom-up tabulation of a slice over the intervals $(i_1 + 1, j_1 - 1)$ and $(i_2 + 1, j_2 - 1)$.

SRNA2 (Algorithm 3) is divided into two stages: (1) for each pair of arcs, tabulate the child slice spawned by matching the pair, and memoize the result of the last subproblem in memoization table M; (2) tabulate the parent slice ($slice_{0,0}$), looking up values in M only when needed. The key to this algorithm is the *order* of memoization. In stage one, the tabulation of child slices is completed in an order that ensures that if some child slice has a dynamic dependency requiring the tabulation of another child slice, that result is already available in M. The bottom-up tabulation of M could instead occur on a row-by-row basis beginning with the higher numbered rows (indexed by i_1) and working towards the lowest numbered row. This order of memoization exactly corresponds with traversing the two structures in a right-to-left order as opposed to a left-to-right order. Both orderings are acceptable and reach the same outcome.

Algorithm 2 TabulateSlice: Table M, Structures S_1 and S_2, Endpoints i_1 and j_1 of the first structure, Endpoints i_2 and j_2 of the second structure

```
Allocate memory for $slice_{i_1,i_2}$
for each arc $(k_1, x) \in S_1$ with $i_1 \leq k_1 < x \leq j_1$ (by increasing order of $x$) do
    for each arc $(k_2, y) \in S_2$ with $i_2 \leq k_2 < y \leq j_2$ (by increasing order of $y$) do
        $max \leftarrow$ MAX($slice_{i_1,i_2}[x-1][y], slice_{i_1,i_2}[x][y-1]$)
        $d_1 \leftarrow slice_{i_1,i_2}[k_1 - 1][k_2 - 1]$
        $d_2 \leftarrow M_{k_1+1,k_2+1}$
        $slice_{i_1,i_2}[x][y] \leftarrow$ MAX($max, 1 + d_1 + d_2$)
    end for
end for
$M_{i_1,i_2} \leftarrow slice_{i_1,i_2}[x][y]$ {$x$ and $y$ refer to the last arc end-points}
Deallocate memory for $slice_{i_1,i_2}$
```

In order to exactly tabulate M, a preprocessing step is performed that determines all of the possible rows and columns that correspond with matched arcs. This is achieved by determining all of the ending points of arcs in the two structures.

C. *Experimental Results for the Sequential Algorithms*

SRNA1 and SRNA2 were implemented in the C programming language using the Portland Group 64-bit C compiler (version 8.0-6), and tested on a Dual-Core AMD Opteron 2.8 GHz computer. The objective was to determine whether or not SRNA2 would outperform SRNA1 as expected, and also to gather sequential results to be used as a comparison baseline for the parallel algorithm (see Section V). Contrived worst-case data, consisting of the maximum number of possible nested arcs for a given sequence length (for example, the structure depicted in Figure 5), was used to fully exhaust the two algorithms. These contrived structures ensured the

Algorithm 3 SRNA2: Structures S_1 and S_2, Length n of the first sequence, Length m of the second sequence

```
Allocate memory for $n \times m$ table $M$
{Stage One - Tabulate child slices}
**for** each arc $(i_1, j_1) \in S_1$ (by increasing order of $j_1$) **do**
  **for** each arc $(i_2, j_2) \in S_2$ (by increasing order of $j_2$) **do**
    TabulateSlice$(M, S_1, S_2, i_1+1, j_1-1, i_2+1, j_2-1)$
  **end for**
**end for**

{Stage Two - Tabulate parent slice}
TabulateSlice$(M, S_1, S_2, 0, n-1, 0, m-1)$
Return $M_{0,0}$
```

	100	200	400	800	1600
SRNA1	0.015	0.238	4.008	76.371	1434.856
SRNA2	0.008	0.128	2.323	37.799	660.696

Table I
EXECUTION TIMES (IN SECONDS) OF SRNA1 AND SRNA2 FOR SEQUENCES OF LENGTHS 100 TO 1600 USING CONTRIVED WORST-CASE DATA.

greatest number of spawned child slices by being as dense as possible in terms of matching arcs. Sequences of length up to 1600 were tested, which required about 10 MB of allocated memory, and Table I shows their execution times. When compared to the worst-case $\Theta(n^2m^2)$ bound on the space complexity for the original formulation, this amounts to a substantial savings, enabling lengthy non-pseudoknot structures to be compared on a sequential machine.

The contrived worst-case data is representative of the kind of substructures that appear in RNA secondary structures on a much smaller scale (i.e., groups of nested arcs), and so it is expected that the execution times for real data should be significantly faster than for the contrived worst-case data. Table II contains the sequential execution times when self-comparing two sample RNA secondary structures. The first RNA secondary structure tested was an example of 23S ribosomal RNA having 4216 bases and 721 arcs (Fungus or *Suillus sinuspaulianus*; Accession #L47585), and the second RNA secondary structure tested was an example of 23S ribosomal RNA having 4381 bases and 1126 arcs (Malaria Parasite or *Plasmodium falciparum*; Accession #U48228). SRNA2 is observed to require roughly half the amount of time to compare RNA secondary structures than SRNA1. As noted, these time savings are a consequence of eliminating overhead associated with memoization table lookups and removing the use of recursion when adopting the two-stage algorithm.

An additional objective of this experiment was to identify the amount of time the SRNA2 program spends during each stage of its execution. These stages include preprocessing,

	Fungus (721)	Malaria Parasite (1126)
SRNA1	49.149	86.887
SRNA2	25.472	39.028

Table II
EXECUTION TIMES (IN SECONDS) OF SRNA1 AND SRNA2 FOR SEQUENCES OF LENGTHS 4216 (721 ARCS) AND 4381 (1126 ARCS).

	100	200	400	800
Preprocessing	0.1814	0.0488	0.0052	0.0002
Stage One	99.6131	99.9055	99.9844	99.9963
Stage Two	0.1693	0.0434	0.0102	0.0034

Table III
PERCENTAGE BREAK-DOWN OF EXECUTION FOR SRNA2 USING CONTRIVED WORST-CASE DATA.

stage one (tabulation of child slices), and stage two (tabulation of the parent slice). Results in Table III show that the sequential program spends over 99% of its execution time in stage one, identifying this stage as the greatest opportunity for parallelism.

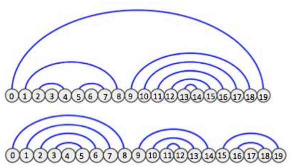
V. PARALLEL ALGORITHM

A. Design

The parallel algorithm for finding common RNA secondary structures (PRNA) is based on SRNA2, and is likewise divided into three parts: preprocessing, stage one, and stage two. Stage one accounts for the most significant percentage of the program that can be executed in parallel, and so emphasis is placed on parallelizing this stage. By using SRNA2 as the basis of the parallel algorithm, we are both using the most efficient sequential algorithm and also choosing the underlying design that is easier to parallelize.

In parallelizing stage one, a child slice is characterized as a primitive task, and the workload is shared among processors by distributing the columns of the parent slice that correspond with matched arcs, which are determined in the preprocessing stage. The relative amount of work between the columns is identical from row to row (see Figure 7), and so a static load balancing scheme can be applied by determining the workload distribution during a preprocessing step, for which we use a greedy approximation algorithm [4]. Whenever a matched arc is encountered, a child slice is spawned and sequentially tabulated by calling $TabulateSlice()$, and the result of tabulating the last subproblem of the child slice is stored in the memoization table. Given that child slices themselves are primitive tasks, parallelism arises when multiple processors are simultaneously tabulating child slices.

During stage two of the algorithm, primitive tasks correspond with each element of the parent slice, which is tabulated by calling $TabulateSlice()$. Since all of the possible dynamic dependencies that require the spawning of child slices (case d_2) have already been memoized in



	5	6	7	8	12	13	14	18	19
19		15	30	45			30		15
18		4	8	12			8		4
17		3	6	9			6		3
16		2	4	6			4		2
15		9	6	3			6		3
14									
8		4	8	12			8		4
7									

Figure 7. The non-empty entries in the table (right) indicates the relative amount of work required to tabulate the spawned child slices.

stage one, tabulating the parent slice in stage two is very straightforward. Although stage two performs operations that could be parallelized, the small percentage of execution accounted for by stage two and the amount of time required for parallel overhead is so great that it is not worth the additional programming effort.

Figure 7 depicts the view of the parent slice when aligning the two example structures. The non-empty table entries contain the number of elements (or subproblems) that are tabulated in the corresponding spawned child slice, which appear in the positions whose indices correspond with matched arcs between the two structures. Since child slices are treated as primitive tasks, it is desirable to balance the workload as much as possible, and the non-empty entries in the table are indicative of the relative amount of work between the columns.

B. Implementation

In the implementation of PRNA described here, we use the message-passing interface (MPI) for parallel programming, though the decisions made in the parallel design would also be appropriate for other parallel platforms. Given the substantial reduction in the amount of allocated memory required to compute this problem, the memoization table M can easily fit in memory for practical problem sizes; thus, the simplest approach to implementing this algorithm is to allocate the memoization table on each processor and then to synchronize the results after each row of the traversal in stage one is completed. This ensures that each processor has up-to-date values when looking up dependent subproblems.

The synchronized memoization table is initialized to 0, and the order in which rows are memoized is data-dependent, but one can be certain that if a particular row is dependent on values from another row, then that other row would have been updated at an earlier time. This falls into place naturally due to the computation order, just as is the case with the sequential algorithm (Section IV-B). The completion of a row in M occurs when moving to the next endpoint in the traversal of the two structures, at which point the table M can be synchronized by performing a reduction operation over the completed row. When programming using MPI, this can be accomplished by calling MPI_Allreduce with the beginning address of the row and number of columns, and using the MPI_MAX operation to ensure that all updated values end up in the receive buffer for Algorithm 4.

Algorithm 4 PRNA: Table M, Structures S_1 and S_2, Length n of the first sequence, Length m of the second sequence

```
{Preprocessing}
Allocate memory for $n \times m$ table $M$
Determine column ownership by calling load_balance

{Stage One – Parallel}
**for** each arc $(i_1, j_1)$ in $S_1$ (by increasing order of $j_1$) **do**

    {Child slices are spawned in parallel}
    **for** each arc $(i_2, j_2)$ in $S_2$ owned by this processor (by increasing order of $j_2$) **do**
        TabulateSlice($M, S_1, S_2, i_1+1, j_1-1, i_2+1, j_2-1$)
    **end for**
    Synchronize row $i_1$ in $M$ across all processors

**end for**

{Stage Two – Sequential}
TabulateSlice($M, S_1, S_2, 0, n-1, 0, m-1$)
Return $M_{0,0}$
```

VI. Experimental Results

Figure 8 illustrates the speedup achieved by PRNA using the contrived worst-case data on a distributed memory parallel computer. PRNA was implemented in C and Open-MPI using the Portland Group 64-bit C compiler (version 8.0-6). The testbed for these experiments was the hybrid parallel cluster *Fundy* at the University of New Brunswick's ACENet-sponsored high-performance computing facilities. Speedup of up to 32X was achieved for the sequence of length 3200 containing 1600 nested arcs, and speedup of up to 22X was achieved for the sequence of length 1600 containing 800 nested arcs. The trend of the results illustrated in Figure 8 suggests scalability, as more speedup is attained when increasing the problem size and the number of processors.

VII. Conclusions

This research demonstrates that, given a careful inspection of how the top-down problem structure unfolds and the interactions of irregular dependencies, a data-driven recurrence for RNA secondary structure comparison can be computed without wasted memory, and can be well-suited for parallelization. This approach leads to problem insights that enable us to eliminate the wasted space and unnecessary computation that result from the conventional

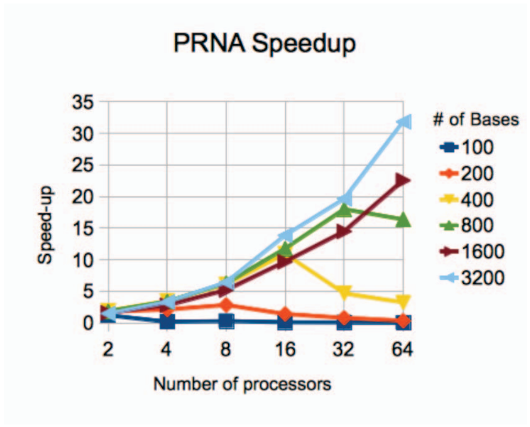


Figure 8. Speedup for PRNA using contrived worst-case data. Up to 32X speedup was achieved using 64 processors and 1600 nested arcs (a sequence containing 3200 bases), and up to 22X speedup was achieved using 64 processors and 800 nested arcs (a sequence containing 1600 bases).

iterative bottom-up strategy for dynamic parallelization. The resulting sequential algorithm is easily parallelized for a distributed memory parallel computer, whereas the original formulation was impractical and difficult to parallelize.

In this case study, the top-down algorithm for finding common RNA secondary structures makes use of one crucial piece of information that controls the manner in which it branches off in its traversal of the dependency graph, and it does so only when an actual matched arc is encountered, rather than trying to account for all possible matched arcs. This observation raised the question as to whether or not the bottom-up algorithm could access this same information and exploit it in order to reduce the amount of overtabulation. The answer is yes, and indeed all of the overtabulation of the bottom-up algorithm was eliminated. With this insight, it turned out the bottom-up algorithm would have a flavour of both the bottom-up and top-down strategies by employing both tabulation and memoization – whenever a matched arc is encountered, recursively spawn off an independent dynamic programming problem to be solved for aligning the intervals beneath the matched arcs, and memoize the optimal value of the tabulated results. Thus, as we iterate over the intervals in a bottom-up manner, we encounter the root nodes of groups of subproblems (subtrees in the dependency graph) that would ordinarily be encountered in the top-down unfolding of the problem, and we can then, by recursion, skip down from the root to the child nodes of the subtree in order to tabulate it in the same bottom-up manner. The end result was that both the bottom-up and top-down algorithms were performing an exact tabulation, with the former accomplishing this without the additional constant amount of overhead. Perhaps most importantly, the child slices could then be tabulated in parallel.

The results suggest that there are problems in which nested structures (or nested sets of subproblems) that emerge in the unfolded dynamic programming problem instance could possess an inherent characteristic in which both tabulation and memoization can co-exist, and better solutions can be devised by more completely taking into consideration the data-driven characteristic. Taken as a whole, the outcome of this research suggests that one ought to explore a dynamic programming formulation with care when devising sequential or parallel algorithms, taking into consideration how the problem structure unfolds in a top-down manner as an exact tabulation of the subproblems in the representative dependency graph, rather than using the generic approach of allocating a table, dividing it up among the processors, and tabulating results using the bottom-up strategy.

Finally, it is the exploration of the problem structure from the point of view of merging both the top-down and bottom-up approaches that led to the development of the improved sequential algorithms that were initially not well-suited to parallelism, but then later were parallelized to good effect. This case study illustrates the benefits of this approach to parallel dynamic programming design, where the parallel design starts with the redesign of the underlying sequential algorithm and makes design choices that will ultimately lead to a better parallelization. Our results suggest that similar problems, where a dynamic programming solution is complicated by recurrence cases that depend on the data, should also be considered using our combined approach to potentially produce more efficient parallel solutions.

References

[1] V. Bafna, S. Muthukrishnan, and R. Ravi. "Computing Similarity between RNA Strings," *DIMACS Technical Report*, Vol. 96, no. 30, 1996.

[2] R. Bellman. *Dynamic Programming*, Princeton University Press, New Jersey, 1957.

[3] P. Evans. "Finding Common RNA Pseudoknot Structures in Polynomial Time," *Proceedings of Combinatorial Pattern Matching '06, Springer-Verlag Lecture Notes in Computer Science (LNCS)*. Vol. 4009, 2006, 223-232.

[4] R. L. Graham. "Bounds for multiprocessing timing anomalies," *SIAM J. Applied Mathematics*, Volume 17, 1969, 263-269.

[5] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*, 2nd ed., Pearson Education Limited, 2003.

[6] J. Kleinberg and E. Tardos. *Algorithm Design*, Pearson Education Inc., 2006.

[7] E. Snow, E. Aubanel, P. Evans. "Dynamic Parallelization for RNA structure comparison," *Proceedings of the Eighth IEEE International Workshop on High Performance Computing Biology (HiCOMB 2009)*, May 2009.

[8] A. Stivala, P. J. Stuckey, M.G. de la Banda, M. Hermenegildo, and A. Wirth. "Lock-free Parallel Dynamic Programming," *Journal of Parallel and Distributed Computing*, Vol. 70, 2010, 839-848.