Visualizing Parallel Dynamic Programming using the Thread Safe Graphics Library

Grey Ballard
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA 27109
ballard@wfu.edu

Sarah Parsons
Department of Computer Science
Wake Forest University
Winston-Salem, NC, USA 27109
parssy18@wfu.edu

***Abstract*—The design and analysis of parallel algorithms are both fundamental to the set of high-performance, parallel, and distributed computing skills required to use modern computing resources efficiently. In this work, we present an approach of teaching parallel computing within an undergraduate algorithms course that combines the paradigms of dynamic programming and multithreaded parallelization. We have developed a visualization tool built with the Thread Safe Graphics Library that enables interactive demonstration of parallelization techniques for two fundamental dynamic programming problems, 0/1 Knapsack and Longest Common Subsequence. We describe the implementation of the tool, the real-time animation it produces, and the results of using it in class. The tool is publicly available to be used directly or as a basis on which to build visualizations of other parallel dynamic programming algorithms.**

I. Introduction

High-performance computing (HPC) and parallel and distributed computing (PDC) topics are being incorporated into undergraduate computer science curricula in various ways, from dedicated topics courses to individual modules distributed across core courses. While HPC and PDC topics are often categorized into the systems side of the curriculum (e.g., the CC2020 curricular report classifies the PDC knowledge area into the *Systems Architecture and Infrastructure* category [1]), teaching parallel algorithmic thinking is just as important on the theory side of the curriculum. In particular, the Knowledge Units of *Parallelism Fundamentals*, *Parallel Algorithms, Analysis, and Programming* and *Parallel Decomposition* as specified within the PDC Knowledge Area of the CS2013 report [2] can all be incorporated into courses where students learn the fundamentals of design and analysis of sequential algorithms. This possibility is already realized in one of the popular textbooks used for algorithms courses [3, Chapter 27].

In this work, we describe an effort to enhance the presentation of PDC concepts in an algorithms course, using a visualization tool to present parallel algorithms for dynamic programming problems. The visualization tool is implemented using the Thread Safe Graphics Library (TSGL) [4], which is designed for educational use, enabling graphics rendering using multithreaded code and real-time animation of parallel algorithms. In Sec. II, we provide background on the dynamic programming problems and parallel computing model we consider, along with a summary of the capabilities and effectiveness of TSGL. We describe our approach of teaching parallel dynamic programming in an algorithms course at Wake Forest University in Sec. III, with a focus on designing and analyzing parallel algorithms for the 0/1 Knapsack and Longest Common Subsequence (LCS) problems. In Sec. IV we present the visualization tool we have implemented with TSGL to illustrate the parallel algorithms for those problems. We provide screenshots of the animation to show how the instructor can display the dynamic programming table and draw table entry values as they are computed within the execution of the parallel algorithm. During the Spring 2020 semester, we used the tool in class (a remote-learning environment due to the COVID-19 interruption of the semester). We report in Sec. V on the student assessments we used that semester in homework assignments and on the final exam, and we suggest alternative possibilities for reinforcement and assessment of the ideas. Finally, we describe our experiences developing and using the visualization tool in Sec. VI. Our source codes for Knapsack and LCS are available as example applications[1] distributed with the TSGL library, and can be used directly or as a basis to extend the ideas to other dynamic programming problems.

The main contributions of our work are

1) an approach for combining elements of dynamic programming and parallel algorithms into a course on the design and analysis of algorithms (see Sec. III),
2) a visualization tool for presenting two parallel dynamic programming algorithms built with TSGL (see Sec. IV),
3) a description of the student assessment instruments we used for the topic of parallel dynamic programming along with suggested alternatives (see Sec. V).

II. Background

A. Dynamic Programming Problems

Dynamic programming is an algorithmic paradigm covered in typical algorithms courses and textbooks [3], [5], [6]. We focus on two example problems in this work, which we briefly review here.

The first problem is the 0/1 Knapsack problem: given a set of n items with values $\{v_i\}$ and weights $\{w_i\}$ and a knapsack

[1] https://github.com/Calvin-CS/TSGL/tree/master/src/examples/

of capacity W, what is the subset of items that maximizes the total value subject to the constraint that the total weight does not exceed W? In this 0/1 variant of the problem, an item is either included or not; it cannot be included multiple times.

The dynamic programming solution to the problem is based on the following recursive rule for the value of the optimal knapsack:

$$K(w,j) = \begin{cases} 0 & \text{if } w = 0 \text{ or } j = 0 \\ \max\{K(w,j-1), v_j + K(w-w_j, j-1)\} & \text{else,} \end{cases} \quad (1)$$

where $K(w,j)$ is defined as the value of the optimal knapsack with capacity w considering the first j items.

The second problem is the Longest Common Subsequence problem: given two strings of characters x and y of lengths m and n, respectively, what is the longest string that appears as a subsequence in both x and y?

The dynamic programming solution is based on the recursive rule for the length of the LCS:

$$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1,j-1) + 1 & \text{if } x[i] = y[j] \\ \max\{L(i-1,j), L(i,j-1)\} & \text{if } x[i] \neq y[j], \end{cases} \quad (2)$$

where $L(i,j)$ is defined as the length of the longest common subsequence of the ith prefix of x and the jth prefix of y.

Other popular problems for which dynamic programming provides a reasonable solution include longest increasing subsequence, sequence alignment/edit distance, subset sum, all-pairs shortest paths, and matrix chain multiplication.

B. Multithreaded Programming Model

The multithreaded programming model [3] is based on the PRAM machine model, which is an abstract shared-memory parallel machine in which each processor can instantaneously access data in the single, unbounded shared memory. In the programming model, a program starts execution with a single thread that is able to spawn or fork other threads (which can themselves spawn new threads) to perform operations in parallel. To maintain correctness, threads are synced or joined before the results of their computations are used for later operations that depend on those results. A nice feature of this parallel programming model is that only two keywords `spawn` and `sync` need to be added to the sequential pseudocode language. For cleaner expression of loop-oriented code, the `parallel for` keyword can also be used in place of a sequential `for` if all loop iterations can be performed independently.

The complexity of a multithreaded algorithm is determined using work-span analysis. The work of an algorithm, denoted $T_1(n)$ for input of size n, is the total number of operations performed, and coincides with the sequential computational complexity: the time it would take to finish with 1 processor. The span of an algorithm, denoted $T_\infty(n)$, is the length of the critical path, which is the longest path in the dependency graph of the algorithm's operations: the time it would take to finish with an infinite number of processors. Given a machine with P processors, the algorithm's time to completion is given by $T_P(n) = \Theta(T_1(n)/P + T_\infty(n))$, assuming an efficient scheduler assigns threads to processors in the PRAM model.

C. Thread Safe Graphics Library (TSGL)

TSGL is a multithreaded graphics library written in C++ that provides various functions and classes for drawing shapes, text, images, and lines in parallel to a canvas screen using multiple threads [4]. Designed as an educational tool for teaching multithreaded programming, TSGL presents a novel approach for visualizing parallel algorithms. Students can observe how multiple threads are called in parallel to draw a visualization in real time and can better understand the benefits of leveraging multiple threads.

TSGL has been used for many applications, such as visualizing parallel loops and the actor parallel design pattern. Specific examples include image processing for color inversion, computing integrals numerically, and drawing the Mandelbrot set [7]. Other applications being explored include parallel merge sort and the task queue pattern.

In the Spring of 2015, an assessment on the effectiveness of using TSGL for visualizing parallel loops was performed in a CS2 course at Calvin College [8]. The assessment demonstrated that students who were provided with an interactive TSGL visualization for learning OpenMP scored higher on an exam question than students who were not. Not only has TSGL proven effective for teaching parallel algorithms, it can also help illuminate coding errors when testing and debugging. When executing code, TSGL enables a programmer to visualize different components of the code and compare the output to their expected result.

The TSGL library is available for download on a Windows, MacOS X, or Linux. A guide for installing the library is included in the TSGL GitHub repository[2], as well as tutorial videos and examples for the various features offered in the library. Integration of these functions into a programmer's C++ program requires a few import statements of the proper classes, installation of certain graphics libraries (if necessary), and the appropriate configuration of a Makefile for compilation.

III. Teaching Parallel Dynamic Programming

The Algorithms course at Wake Forest University is required of BS majors and is typically taken in the 3rd or 4th undergraduate year. The prerequisites ensure that students have already been exposed to a variety of data structures, including heaps, graphs, and hash tables, as well as algorithmic techniques, such as divide and conquer and dynamic programming. Students will also have gained experience writing rigorous proofs and will understand how time and space complexity are computed. The learning outcomes of the Algorithms course are set so that at the end of the course, students should be able to 1) analyze algorithms using asymptotic complexity analysis using Big-Oh and related notation; 2) prove correctness of

[2]https://github.com/Calvin-CS/TSGL

algorithms using rigorous techniques such as mathematical induction and proof by contradiction; 3) design efficient algorithms for combinatorial problems, using approaches that include divide and conquer, dynamic programming, greedy methods, randomization, and parallelization; 4) identify NP-complete problems and devise strategies to deal with them. The focus of the course is on the design and analysis of algorithms, but there are typically programming assignments in addition to theoretical assignments used for reinforcement of the ideas and assessment of the learning outcomes.

A. Course Topics

The semester-long course is typically structured into 7 units, each allocated roughly 2 weeks, that follow mostly from [5]. The units are (in order) 1) integer arithmetic and RSA encryption; 2) divide and conquer; 3) graph algorithms; 4) greedy techniques; 5) dynamic programming; 6) parallel programming; and 7) NP-complete problems. Readings are assigned from the required textbook [5] for all units except for parallel programming. The content of the parallel programming unit largely follows [3, Chapter 27], and reading is assigned using either online access to the chapter through the publisher's website or from a technical report that was the precursor to the chapter being added to the 3rd edition [9].

The course topics help students to achieve the learning outcomes by introducing powerful techniques for designing algorithms and demonstrating analysis of both the correctness and complexity of a multitude of clever algorithms for solving combinatorial problems. The order of the topics also allows for smooth transitions between them, including using divide-and-conquer integer multiplication to transition between the first two topics (integer arithmetic and RSA encryption, divide and conquer) and using Kruskal's minimum spanning tree algorithm to transition between graph algorithms and greedy techniques. Highlighting the overlap among topics helps both to reinforce the old topic from earlier in the course and to scaffold the new topic using context the students are already comfortable with. As described in Sec. III-C, we continue this trend in exploring parallel algorithms for dynamic programming problems.

B. Parallel Programming Unit

The objective of the parallel programming unit is for students to learn how to design and analyze parallel algorithms. We choose to use the multithreaded programming model in the Algorithms course for several reasons. First, it aligns well with the *Parallel Algorithms, Analysis, and Programming* Knowledge Unit within the *Parallel and Distributed Computing* Knowledge Area as specified in the ACM/IEEE computer science curriculum guidelines [2]. The topics in this Knowledge Unit include work-span analysis, which is central to the multithreaded programming model. Second, the PRAM model is the easiest generalization of the (implicit) RAM model used in the complexity analysis of the sequential algorithms covered in the course. For example, using a distributed-memory machine model would force students to reason about data locality, which is not required of the previous sequential analysis, and would add to the cognitive load. Because work analysis is identical to the sequential analysis, students need to learn only how to analyze the span of a parallel algorithm, which helps them focus on dependencies among tasks and the core concepts of thinking in parallel. Third, the model's syntax is a lightweight extension of the pseudocode with which the students are already familiar. Many sequential algorithms can be revisited and parallelized simply by annotating them with `spawn` and `sync` keywords.

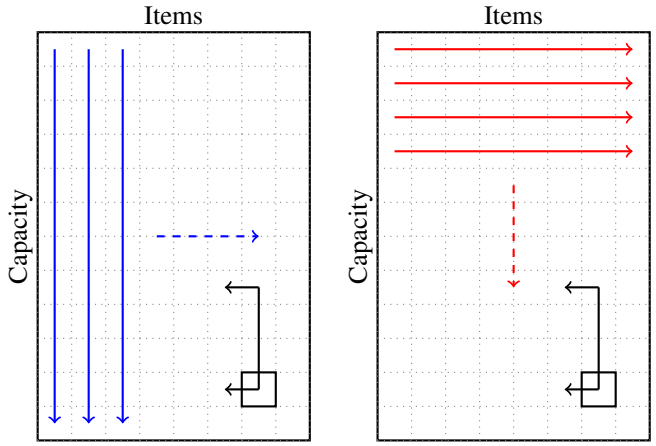
The parallel programming unit is scheduled near the end of the semester, as parallelization is used most effectively in combination with other algorithmic paradigms. Because of the fork-join property of the multithreaded programming model, the ideas of parallelization build most naturally from the divide-and-conquer technique. By analyzing the dependency graph among recursive calls, one can choose a correct parallelization by spawning threads to execute independent calls and syncing to satisfy dependencies among calls. Furthermore, performing span analysis involves deriving and solving recurrences, reinforcing use of the Master theorem and introducing unique recurrences that are not considered in sequential analyses. Divide-and-conquer matrix multiplication and mergesort are the main algorithms studied in [3, Chapter 27]. These are illustrative examples for demonstrating dependency analysis (write conflicts between pairs of memory-efficient matrix multiplication calls) and the effects of Amdahl's law (if the merge is not parallelized in mergesort).

C. Parallel Dynamic Programming

In addition to applying parallelization to divide-and-conquer algorithms in the Algorithms course, we also demonstrate parallel dynamic programming. Sequential dynamic programming solutions typically use nested loops in order to fill in the dynamic programming table. Because the sequential computation of table entries must respect the dependencies of the recursive rule, students have already been exposed to (usually simple) dependency analysis in this context. Also, because the sequential code is typically written with loops, parallel dynamic programming provides compelling examples of the use of `parallel for` instead of the combination of `spawn` and `sync` used for divide-and-conquer algorithms. Thus, dynamic programming is another natural platform on which to build parallel algorithmic thinking.

Many dynamic programming algorithms can work well to illustrate parallelization, but we focus on Knapsack and LCS problems. These are convenient because they both have 2D tables that are amenable to visualization, and more importantly, they have different dependency structures that affect parallelization strategies. This latter distinction forces students to reason about the available task parallelism in each problem.

1) Knapsack: As specified by eq. (1), the Knapsack table is 2D with rows corresponding to capacity values and columns corresponding to items. Fig. 1 shows two valid sequential orders of filling in the table. As shown in the dependencies of a sample entry, each entry has two entries whose values



(a) Column-wise (b) Row-wise

Fig. 1. Valid sequential orders for Knapsack table and sample entry's dependencies. The column-wise ordering is parallelizable but the row-wise ordering is not.

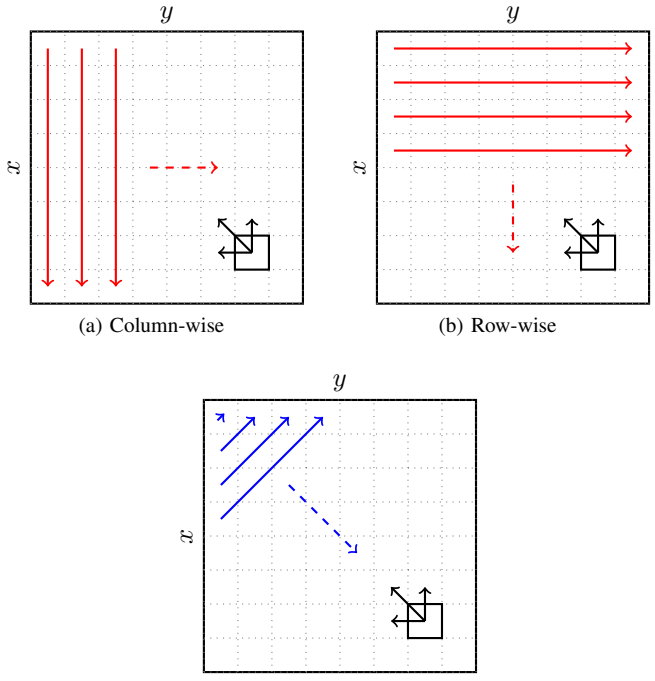
it depends on, both of which are in the previous column. Therefore, the table can be filled in column-by-column, from top to bottom and working left to right, as shown in Fig. 1a, or it can be filled in row-by-row, from left to right and working top to bottom, as shown in Fig. 1b. While both orderings are valid for sequential algorithms, only the column-wise ordering is easily parallelized. This is because all of the entries in a column are independent, while there is a chain of dependence within a row that prevents parallel computation. Contrasting these two sequential approaches demonstrates the dependency analysis required for designing correct parallel algorithms.

Algorithm 1 Parallel Knapsack

```
    # loop over items
 1: for j = 1 to n do
        # parallelize over entries in column
 2:     parallel for w = 1 to W do
 3:         if w_j < w then
 4:             K(w, j) = K(w, j−1)
 5:         else if K(w, j−1) ≥ v_j+K(w−w_j, j−1) then
 6:             K(w, j) = K(w, j−1)
 7:         else
 8:             K(w, j) = v_j + K(w−w_j, j−1)
 9:         end if
10:     end for
11: end for
```

The parallel algorithm implementing the column-wise ordering is given in Alg. 1. Only the main nested loops are shown; the code assumes the base case values have already been set and backtracking is not shown. The sequential outer loop is over columns and the parallel inner loop is over rows within a column. The work of the algorithm is $\Theta(nW)$ and the span is $\Theta(n \log W)$, for n items and capacity W.

2) LCS: The LCS rule is defined in eq. (2) and adds a useful complication compared to Knapsack. The table is again 2D,



(a) Column-wise (b) Row-wise

(c) Anti-diagonal

Fig. 2. Valid sequential orders for LCS table and sample entry's dependencies. The anti-diagonal ordering is parallelizable but the column- and row-wise ordering are not.

with rows corresponding to the characters of the 1st string and columns corresponding to characters of the 2nd string. Fig. 2 shows three different orderings of filling in the table, all of which satisfy the dependencies shown for the example entry. For LCS, each entry depends on either its west and north neighbors or its northwest neighbor, depending on whether the corresponding characters match. This dependency structure implies that column- and row-wise orderings (Figs. 2a and 2b) are not easily parallelized because of the dependencies within columns and rows. Finding the parallelism in anti-diagonals as shown in Fig. 2c is a nice challenge for students after understanding the Knapsack parallelization.

The pseudocode with the main loops of the anti-diagonal approach is given in Alg. 2. Recall that m and n are the lengths of the strings. The sequential outer loop is over anti-diagonals and the parallel inner loop is over entries within an anti-diagonal. For each anti-diagonal, the starting row and the length of the anti-diagonal can vary depending on whether the diagonal is within or after the first m and whether $m \geq n$ or not; the logic of line 2 through line 6 handle these cases. The k variable indexes which entry of the anti-diagonal is being processed, and the row and column indices (i, j), which sum to $d + 1$ for the dth anti-diagonal, are computed in line 8. The work of the algorithm is $\Theta(mn)$ and the span is $\Theta((m + n) \log(\min\{m, n\}))$, because the maximum length of an anti-diagonal is the minimum of the two strings' lengths.

We note that another parallelization of LCS can be done via a divide-and-conquer approach that applies the anti-diagonal

```
// draw horizontal lines of grid and characters of left string
for (int i=0; i<m; i++){
    bgp->drawLine(x1,y2-i*row_wid,0,x2,y2-i*row_wid,0);
    bgp->drawText(x1-30,y2-(i+0.5)*row_wid,0,to_string(x[i]));
}
bgp->drawLine(x1,y1,0,x2,y1,0);

// draw vertical lines of grid and characters of top string
for (int j=0; j<n; j++){
    bgp->drawLine(x1+(j*col_wid),y1,0,x1+(j*col_wid),y2,0);
    bgp->drawText(x1+(j+0.5)*col_wid,y2+30,0,to_string(y[j]));
}
bgp->drawLine(x2,y1,0,x2,y2,0);
```

Fig. 3. Code snippet of drawing LCS grid with TSGL. The `bgp` pointer points to an object that holds a canvas that can be drawn on by multiple threads simultaneously using functions such as `drawLine` and `drawText`.

```
// loop over columns (items)
for(j = 1; j < items+1; j++){
    vj = values[j-1];
    wj = weights[j-1];
    // parallelize over the entries (capacities) within column
    #pragma omp parallel for
    for(w = 0; w <= capacity; w++) {
        if (knap[w][j-1] < vj + knap[w-wj][j-1] && w-wj >= 0) {
            // include the jth item
            knap[w][j] = vj + knap[w-wj][j-1];
        } else {
            // don't include jth item
            knap[w][j] = knap[w][j-1];
        }
        // pause a second and then fill in table entry on canvas
        bgp->sleepFor(1);
        bgp->drawText(x1+(h-0.5)*col_wid,y2-(k+0.5)*row_wid,0,to_string(knap[k][h]);
    }
}
```

Fig. 4. Code snippet of parallel Knapsack code with TSGL visualization. The `sleepFor` function call controls the speed of visualization, and the `drawText` function call displays the value of the table entry in the correct location on the canvas. Because the `drawText` calls are within a parallel for loop, multiple entries will appear on the canvas simultaneously.

ordering to submatrices rather than entries (see [3, Problem 27-5] or [10]). For example, one can divide the table into quadrants, recursively complete the top-left quadrant first, then the top-right and bottom-left quadrants in parallel next, then the bottom-right quadrant last. This approach has more in common with the divide-and-conquer algorithms for matrix multiplication and mergesort, though the dependency structure among recursive calls is more complicated. When $m = n$, the span of this approach is $O(n^{\log_2 3})$, which exceeds that of the loop-based approach.

IV. TSGL Visualization

Using TSGL, we developed visualizations for both Knapsack and LCS algorithms to demonstrate their behaviors while running in parallel. We use library functions to draw and label the dynamic programming table on a canvas, and then within the loops of the algorithms, add function calls to draw the value of each entry as it is computed. Because TSGL is thread safe, we can draw to the canvas within parallel for loops and allow multiple threads to add entries simultaneously to demonstrate execution of the algorithm. Using the `sleepFor` function, we can control how long a thread pauses before drawing, which controls the speed of the visualization and also keeps the threads roughly synchronous.

In Fig. 3, we show a C++ code snippet of (simplified) calls to TSGL to draw the dynamic programming table for LCS and label each row and column by the character of the corresponding string. The `bgp` variable is a pointer to an object that holds the canvas and has member functions for drawing text, lines, and other shapes. Other variables that are defined before the code snippet set parameters such as the corners of the table, the widths of the rows and columns, and the lengths of the strings. Note that there is no parallelism in these loops (though there could be) because the table is drawn and displayed before the parallel algorithm for filling the table is executed.

The C++ code snippet in Fig. 4 shows the main loops of the Knapsack algorithm given in Alg. 1. Here we see the use of OpenMP's `#pragma omp parallel for` directive for the inner loop. The code matches the pseudocode presented earlier, but we added two TSGL function calls within the inner loop: line 16 causes the thread to pause for 1 second, and

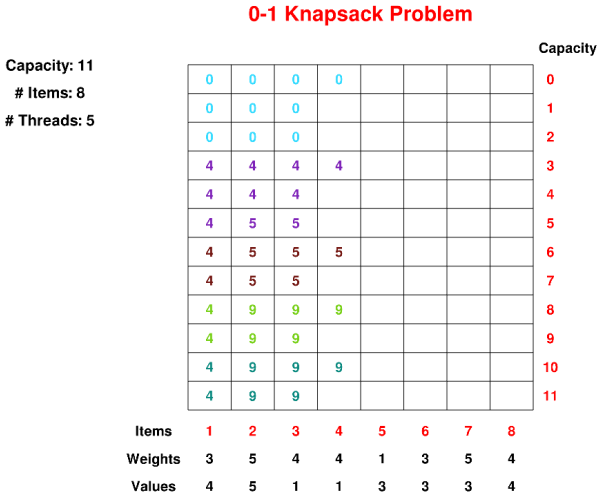


Fig. 5. Visualization of parallel Knapsack with 5 threads filling 4th column

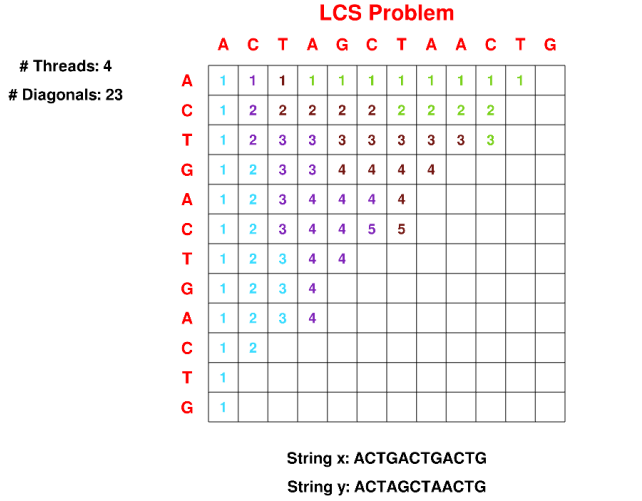


Fig. 6. Visualization of parallel LCS using 4 threads working on 12th diagonal

line 17 writes the computed value to the correct entry location on the canvas's table. Though not shown in the code snippet, our implementation uses a unique color for each thread.

We show a screen capture of the TSGL animation of the Knapsack algorithm in Fig. 5. In this example, the capacity is 11, the number of items is 8, and the number of threads used is 5. In this way, each column of length 12 is unevenly divided over 5 threads, and we can see the default behavior of OpenMP scheduling with one chunk per thread and the first two threads each assigned one more iteration than the others. Fig. 5 shows the moment after all threads complete their first entry of the 4th column; note that each entry is a unique color corresponding to the thread that computed it.

Fig. 6 shows a screen capture of the LCS animation. In this case, each DNA-inspired string is 12 characters long and 4 threads are used. The figure shows the main anti-diagonal of length 12 being filled in by the four threads. Because the lengths of the anti-diagonals vary, the animation demonstrates all possibilities of uneven division of loop sizes over threads.

Algorithm 2 Parallel LCS

```
    # loop over diagonals
 1: for d = 1 to m + n − 1 do
        # set starting row and length of diagonal
 2:     if d ≤ m then
 3:         i_0 = d, ℓ = min{d, n}
 4:     else
 5:         i_0 = m, ℓ = min{m, m + n − d}
 6:     end if
        # parallelize over entries in diagonal
 7:     parallel for k = 1 to ℓ do
            # set row and col indices
 8:         i = i_0 − k + 1, j = d − i_0 + k
 9:         if x_i = y_j then
10:             L(i, j) = L(i−1, j−1) + 1
11:         else
12:             L(i, j) = max{L(i−1, j), L(i, j−1)}
13:         end if
14:     end for
15: end for
```

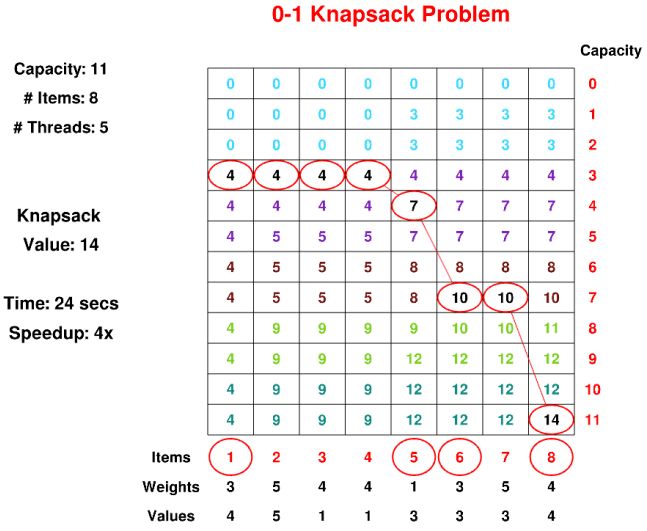


Fig. 7. Visualization of backtracking for the Knapsack example of Fig. 5

While not apparent in the screen captures, the main strength of the TSGL visualization is the real-time animation of the progression of the algorithms. Students can watch the column-by-column or diagonal-by-diagonal progression of the algorithm, they can see the assignment of loop iterations to threads as the number of iterations and threads vary, and they can request demonstrations of certain configurations. It is easy to ask questions, such as "What if we use 100 threads?" or "Do you think 5 threads will finish faster than 4 threads?", have the students develop their hypotheses, and then run the demonstration. When showing the visualizations during the Spring 2020 semester, multiple students asked their own

questions that were answered by demonstration.

Our implementation also demonstrates a sequential backtracking process,[3] as we show for Knapsack in Fig. 7. The visualization of backtracking can be toggled on or off. After the table is filled, we show the measured time and speedup. We considered but did not implement highlighting previously computed entries that an entry depends on in order to show the dependency pattern and why other parallelizations are infeasible.

V. Reinforcements and Assessments

In this section, we discuss how the ideas of parallel dynamic programming can be reinforced and how students' abilities to design and analyze parallel algorithms can be assessed. We report the instruments and results for the Spring 2020 semester, when the TSGL demonstration for Knapsack and LCS was used, as well as propose several alternate strategies and exercises.

A. Spring 2020 Instruments and Results

As a reinforcement exercise in our Spring 2020 Algorithms course, two questions on dynamic programming were presented to students on a homework assignment. The first question asked students to provide a sequential algorithm for the subset sum problem:

> *Suppose we are given a set of positive integers* $\{a_1, ..., a_n\}$ *and a single positive integer* S*. Give an* $O(nS)$ *algorithm to answer the following question: does there exist a subset of the integers that add up to exactly* S*? Justify its correctness and running time, and describe its memory footprint.*

The second question followed from the first, asking students to provide a parallel solution for the subset sum problem and to analyze the algorithm:

> *Give an efficient parallel algorithm for solving the previous problem. Justify its correctness, and derive its work, span, and parallelism.*

Responses to the first question were mostly accurate, with only 4 of 21 students answering incorrectly. Wrong answers were a result of students either choosing methods other than dynamic programming or lacking full understanding of the Knapsack algorithm and its relation to the subset sum problem. Responses to the second question were more varied, with 10 students receiving full credit; 5 students performing incorrect analysis, particularly for the computation of span; 2 students that did not submit; and 4 students submitting an incorrect algorithm, in which they either did not identify the proper data dependency or parallelized both inner and outer loops of the algorithm. These results correspond to 15 of 21 students (71%) correctly designing the parallel algorithm, a success rate we attribute in large part to the use of TSGL during lecture.

To further assess students' ability to design parallel algorithms, we consider one question on the final exam of our Spring 2020 Algorithms course on comparing the parallelization scheme of Knapsack to that of LCS. The question was as follows:

> *The recursive function for computing subproblem values of Longest Common Subsequence is*
>
> $$L(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L(i-1, j-1) + 1 & \text{if } x[i] = y[j] \\ \max\{L(i-1, j), L(i, j-1)\} & \text{if } x[i] \neq y[j] \end{cases}$$
>
> *Explain why the parallelization scheme for this dynamic programming solution must be different than that of 0/1 Knapsack (i.e., items can be chosen at most once).*

In response, we expected students to highlight the distinct dependency structures of the two algorithms and to emphasize the need to parallelize LCS within the diagonals of the table while Knapsack can be parallelized within columns (items). Of the 21 students in the course, 11 (52%) correctly responded. Responses varied in detail, but for incorrect answers, students tended to focus more on the difference in definitions of the algorithms rather than the schemes used to parallelize them.

The correct response rate for this problem was lower than the homework problem, which we attribute in part to it being a timed, cumulative assessment. Because the incorrect responses failed to highlight the differences in dependency structure, we believe emphasizing the dependencies within the visualization would solidify students' understanding.

B. Alternate Strategies

There are many dynamic programming solutions whose dependency structures are variants of Knapsack and LCS that can be used as tools for reinforcement of the parallelization ideas. For example, like subset sum to Knapsack, the edit distance problem has identical structure to LCS. The all-pairs shortest-paths problem extends the ideas of Knapsack to a 3D table, where the two innermost of the three nested loops can be parallelized. Like Knapsack, only two slices of the table need to be maintained in memory during the course of the algorithm. For a more difficult extension, the Knapsack with replacement and longest increasing subsequence problems both involve 1D tables where the available parallelization occurs within the computation of each entry. Designing the parallel algorithm for these problems requires students to apply parallel reduction, a technique that is not used in either Knapsack or LCS. Finally, the matrix chain multiplication problem is an example with a 2D table that, like LCS, allows for the opportunity to parallelize within diagonals, but also enables parallelization within the computation of each entry via parallel reduction.

Another possibility is to ask students to design, debug, and analyze their parallel algorithms using TSGL. Provided a template sequential program with TSGL function calls already included, students could write their own OpenMP code (perhaps only annotating with OpenMP pragmas) and use TSGL visualization as a tool to debug their code as they develop their algorithms or to test their theoretical analysis

[3]Parallelizing the backtracking process using a divide-and-conquer approach can be considered as an advanced topic [11].

of work and span. This possibility requires students to install TSGL and its dependencies, which can present obstacles for some students but enables hands-on interaction with the tool.

VI. Conclusion and Experiences

In this work, we describe a means of incorporating parallel and distributed computing ideas into an undergraduate algorithms course. Using the multithreaded programming model, students can extend their learning of the design and analysis of algorithms into the parallel context. While the divide-and-conquer paradigm is a natural platform on which to build the ideas of task parallelism, we argue here that dynamic programming is another algorithmic technique that combines well with parallelism, in terms of both efficiency and pedagogy. Using the TSGL library as a tool for visualizing parallelization strategies, we propose a novel technique for presenting parallel dynamic programming to students using 0/1 Knapsack and LCS as examples with distinct dependency structures.

The development of this tool progressed over the course of the 2019-2020 academic year. The authors first learned of TSGL when Ballard attended the CSinParallel Piedmont Regional Workshop[4] at Winston-Salem State University in the summer of 2019. During the following semester (Fall 2020), Ballard taught a graduate course on parallel algorithms, and, as a graduate student in that class, Parsons chose to complete a final project focused on PDC pedagogy. Parsons implemented the first versions of the Knapsack and LCS visualizations using TSGL, targeting use in the undergraduate algorithms course, as the two algorithms are studied in both undergraduate and graduate courses. In the Spring 2020 semester, Ballard presented the TSGL tool in the undergraduate course on algorithms to demonstrate parallel dynamic programming. The Spring 2020 semester was interrupted by the COVID-19 pandemic, and both dynamic programming and parallel programming units occurred after the switch to remote learning, so the TSGL visualization was presented virtually. After presenting the tool at the CSinParallel Summer 2021 Virtual Workshop, Ballard and Parsons updated the implementations and submitted them as additional examples available with the TSGL library.

In our experience, the visualizations were well received by students, and we plan to use them regularly in future semesters of the undergraduate algorithms course, as well as in graduate courses where applicable. In particular, TSGL allows for an interactive demonstration that is often more effective than lecturing with presentation slides, even when they are well animated. We believe using TSGL increases student engagement in class by eliciting more discussion in the form of questions and suggestions from students. While the demonstration in the Spring 2020 course seemed to convey the algorithmic design ideas well, we hope in the future to use it more effectively to clarify the work/span analysis and its relation to the time taken by a specified number of threads. Due to the unfortunate circumstances of the Spring 2020 semester, we also learned that TSGL visualizations can be used as easily in remote settings as in-person classes. It is possibly even more valuable in remote settings where student engagement can be more challenging to foster.

We found that developing and running TSGL code requires significant effort, particularly in configuring the environment. Because it involves graphics, there are several dependencies required for both compilation and execution, and these must be installed on the machine used in class to perform the demonstrations. The TSGL library includes installers for each operating system, which are helpful. After the environment is correctly configured, we found the use of the TSGL library to be straightforward, especially given the large set of available example codes. We hope that our implementations will be useful for others as is, but we also think that extending our examples to develop new dynamic programming visualizations is a promising direction.

[4]https://csinparallel.org/csinparallel/workshops

Acknowledgment

The authors would like to thank Joel Adams, Ryan Vreeke, and Samuel Haileselassie from Calvin University for their integration of the Knapsack and LCS examples into the TSGL library. This work is supported by the National Science Foundation under Grant No. CCF-1942892 and OAC-2106920.

References

[1] CC2020 Task Force, *Computing Curricula 2020: Paradigms for Global Computing Education*. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://dl.acm.org/doi/book/10.1145/3467967

[2] Joint Task Force on Computing Curricula, ACM and IEEE CS, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://dl.acm.org/doi/book/10.1145/2534860

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[4] J. C. Adams, P. A. Crain, and M. B. Vander Stel, "TSGL: A thread safe graphics library for visualizing parallelism," *Procedia Computer Science*, vol. 51, pp. 1986–1995, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915012715

[5] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani, *Algorithms*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2006.

[6] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[7] J. C. Adams, P. A. Crain, and C. P. Dilley, "Seeing multithreaded behavior using TSGL," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 972–977. [Online]. Available: https://www.computer.org/csdl/proceedings-article/ipdpsw/2016/3682a972/12OmNyaoDyH

[8] J. C. Adams, P. A. Crain, C. P. Dilley, C. D. Hazlett, E. R. Koning, S. M. Nelesen, J. B. Unger, and M. B. V. Stel, "TSGL: A tool for visualizing multithreaded behavior," *Journal of Parallel and Distributed Computing*, vol. 118, p. 233–246, 2018. [Online]. Available: https://doi.org/10.1016/j.jpdc.2018.02.025

[9] C. E. Leiserson and H. Prokop, "A minicourse on multithreaded programming," MIT, Tech. Rep., 1998. [Online]. Available: http://supertech.csail.mit.edu/papers/minicourse.pdf

[10] R. A. Chowdhury and V. Ramachandran, "Cache-efficient dynamic programming algorithms for multicores," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, 2008, pp. 207–216. [Online]. Available: https://dl.acm.org/doi/10.1145/1378533.1378574

[11] S. Aluru, N. Futamura, and K. Mehrotra, "Parallel biological sequence comparison using prefix computations," *Journal of Parallel and Distributed Computing*, vol. 63, no. 3, pp. 264–272, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731503000108