TSGL: An Exploratory Quest to Visualize the Parallelization of Knapsack and Longest Common Subsequence

Sarah Parsons*
parssy18@wfu.edu
Wake Forest University
Winston-Salem, NC

ABSTRACT

In an effort to visualize the parallelization of the 0-1 Knapsack problem and Longest Common Subsequence for educational purposes, the Thread Safe Graphics Library (TSGL) was employed to develop scripts for viewing these problems in the classroom. Upon building simple examples for each problem in C++ and parallelizing the algorithms using OpenMP, additional code was developed for visualizing the parallelization of each problem on a canvas when given the number of threads and any pertinent parameters for the respective problem.

KEYWORDS

parallel algorithms, 0/1 Knapsack, Longest Common Subsequence, visualization, C++, OpenMP

1 INTRODUCTION

As students studying computer science navigate a curriculum of algorithms, complexity analysis, and various programming models, it is essential that the opportunities for parallel manipulation of sequential algorithms be presented in a clear and consumable manner. Depending on the interests of the student and the focus of the computer science program in which they are enrolled, a parallel algorithms course may not be offered to undergraduates, or may be taught as part of a broader algorithms class. Consequently, a student's first introduction to parallelization and parallel programming interfaces such as OpenMP or MPI may consist of merely a brief overview of the possibilities and purpose of parallelization. Whether learning for the first time or opting to take a course devoted to parallel algorithms, there exists a need to provide students with a learning tool that details the performance and operations occurring throughout the parallelization of an algorithm. An ideal tool could assist students in visualizing the inner workings of a classic algorithm, such as the 0/1 Knapack Problem.

The Thread Safe Graphics Library (TSGL)[1] is a repository of C++ functions and classes used to illustrate the drawing of 2D graphics by a set of threads on a user's local machine. As stated in their mission, TSGL was primarily built for the purpose of teaching parallelization to students through visualizations of thread operations[1]. As outlined in this paper, upon learning and exploring the offerings provided by TSGL, work was completed to build two graphics - one for visualizing the process of solving the 0/1 Knapsack Problem and one for the Longest Common Subsequence. By visualizing the different methods used to parallelize each problem, a learner can better understand the logic used to build each parallel algorithm and its impact on the performance of each model.

2 TSGL OVERVIEW

As noted in the introduction, TSGL[1] is a C++ library that provides a package of various functions and classes for drawing shapes, text, images, and lines to a canvas screen on a local machine. Integration of these functions into a programmer's C++ program is effortless and requires only a few import statements of the proper classes, and the appropriate configuration of a Makefile for compilation.

In addition to drawing objects, programmers can format their drawings with color, text, and texture classes. Further, to leverage the parallel features of the library, a user can animate TSGL drawings using methods such as 'Canvas.sleepFor()' and 'Canvas.wait()', that make up the Canvas class. Last but not least, in order to allow for interaction by an end user, programmers can manipulate the mouse and keyboard interaction feature. The majority of the base functions are comprised in the 'Canvas' class; however, the library is made up of over 20 different classes, each with a range of additional features for more specific goals, such as drawing triangle strips to the canvas, outputting a progress bar as the animation is drawn, or creating a unique font or color palette. For this project, functions used to visualize the filling in of a 2D grid were pulled mainly from the 'Canvas.cpp', 'Line.cpp', and 'Color.cpp' files.

The TSGL library is available for download on a Windows, MacOS X, or Linux. Although there is a guide included in the repository for installation, installing the library may require extensive effort and time. Due to the dependencies of TSGL on GLFW and GLEW, if a user does not already have these libraries installed on their machine, the process for intalling TSGL may require some additional troubleshooting, as an attempt by the package will be made to install them for the user. Also, the installation uses 'Homebrew', which may need to be installed for users currently without it. Unfortunately, there is minimal documentation for installation errors encountered by others, or for common dependencies that users must install on top of TSGL. The process to successfully install the library required more time than necessary and was the main pain point faced during this project. Moreover, efforts were made to replicate the process on another user's machine and the process was unsuccessful due to limited time and failed troubleshooting.

Also included in the repository are tutorial videos and examples for each of the different features offered in the library. Although helpful for newcomers to familiarize themselves with the general features offered, the tutorials are not up-to-date and should not be used for building a simple sample program. The latest code has slightly altered function names and has consolidated several functions. Despite the inconvenience, the tutorials are still helpful when learning what classes are available and in what order certain methods should be called in one's program.

3 PARALLEL ALGORITHMS

To present the best representation of parallel algorithms using the TSGL visualizations, the choice was made to visualize the 0/1 Knapsack Problem and the Longest Common Subsequence (LCS). Both problems are known for their applications in dynamic programming and their ideal setup for parallelism. With similar goals of iteratively solving for a value that is dependent on the answer of a previous iteration, the algorithms used to solve both problems lend themselves well to the use of a 2D grid to store and record answers from each iteration. With a grid, prior solutions only require being solved once and can be stored for future reference. Further, given the grid-like structure used to organize the data, not all iterations are dependent on all other iterations. As such, certain iterations can be computed simultaneously while others are being solved. This notion is essential to the premise for parallelizing these algorithms, all with the goal of enhancing performance time and reducing inefficiency. The Knapsack Problem and Longest Common Subsequence are similar for the reasons listed above, but as described in each subsection below, the order by which each iteration must be solved is not the same, and thus, the algorithm for parallelizing each problem is different. Please note that all parallel code was written using OpenMP pragmas.

3.1 0/1 Knapsack Problem

The Knapsack Problem is a classic computer science problem in which, as its name suggests, one works to maximize the value of weighted items to be placed in a knapsack without exceeding the weight capacity of the sack. Each item is assigned a weight and value, while a capacity is also attributed to the knapsack. Given a variable set of items and capacity, this problem is known to be NP-complete. Nevertheless, by leveraging a 2D grid to represent the capacities (y-axis) and number of items (x-axis), it is possible to iterate through each combination of weighted items to test the possible values that can be added to the knapsack. The final solution will be stored in the last cell of the 2D grid, positioned in the bottom right corner.

The equation for solving the Knapsack Problem is provided below:

$$K(w, j) = max\{K(w, j-1), \quad v_j + K(w - w_j, j-1)\}, \qquad (1)$$

for w capacity, j item, w_j weight of jth item, and v_j value of jth item

As noted in (1), each iteration of the algorithm requires the computation of two different values before selecting the maximum to store in grid cell at index (w,j). The comparison is between the cell to the left of the current cell (which lies in the previous column of the grid) and the cell in the previous column at the row value equal to the weight of the current cell minus the weight of the jth item. Thus, both values to be compared are located in the previous column.

In order to parallelize the problem, given that for each iteration two values from the previous column are evaluated and compared, note that the solutions in each column depend only on the solutions of the previous column. Consequently, if provided the solutions of the previous column, it is possible to solve all solutions in the current column in parallel. The Knapsack Problem will then be solved in parallel by parallelizing the rows so that each processor is responsible for a set of rows for each column.

As stated, the final value of the knapsack will be computed and stored in the final cell of the grid. However, knowledge of the selected items to be placed in the knapsack is not recorded in this algorithm. In order to maintain a record of each item chosen, an indicator can be stored by the processor to denote which of the two items that were compared was chosen.

3.2 Longest Common Subsequence

The Longest Common Subsequence is also classic computer science problem in which, one works to determine the longest consecutive or non-consecutive subsequence of variables or characters that exist in two strings. Given any two strings, this problem is also known to be NP-complete. Using a 2D grid to represent the string composition (characters of string one on y-axis and characters of string two on x-axis), it is possible to iterate through each pair of characters to compare for equality. The final solution will be stored in the last cell of the 2D grid, positioned in the bottom right corner.

The equation for solving the Longest Common Subsequence is provided below:

$$\begin{aligned} c[i,j] = \{\{1 + c[i-1, j-1]\} \quad & if x_i = y_i, \\ max\{c[i-1,j], c[j-1,i]\} \quad & if x_i \neq y_i, \\ \{0\} \quad if \quad i = 0 \quad or \quad & j = 0\} \end{aligned} \qquad (2)$$

for character i in the string on the x-axis, character j in the string on the y-axis

As noted in (2), each iteration of the algorithm requires the computation of three different values before selecting the maximum to store in grid cell at index (i,j). The comparison is between the cell to the left of the current cell, the cell above the current cell, and the cell in previous column and previous row (diagonally left and above). Thus, all values to be compared for one cell are located in the previous antidiagonal.

In order to parallelize the problem, given that for each iteration three values from the antidiagonal are evaluated and compared, note that the solutions in each antidiagonal depend only on the solutions of the previous antidiagonal. Consequently, if provided the solutions of the previous antidiagonal, it is possible to solve all solutions in the current antidiagonal in parallel. The Longest Common Subsequence will then be solved in parallel by parallelizing the antidiagonals so that each processor is responsible for a set of cells along each antidiagonal.

As stated, the final length of the common subsequence will be computed and stored in the last cell of the grid (far right bottom row). However, knowledge of the characters in common is not recorded in this algorithm. In order to maintain a record of each common character, an indicator can be stored by the processor to denote which of the three items that were compared are equal.

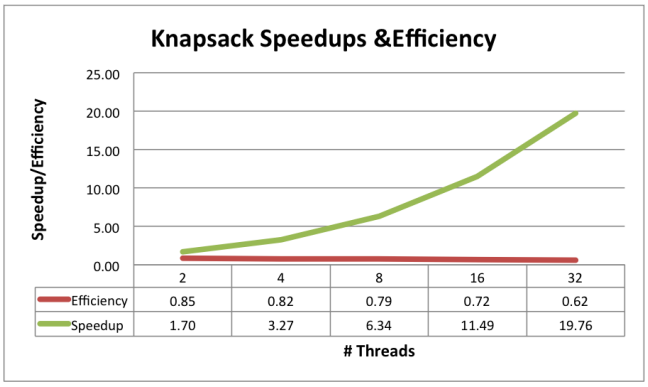
4 PARALLEL IMPLEMENTATION

To parallelize each algorithm, OpenMP was used. For Knapsack, two nested for loops were implemented to traverse the rows of each column of a 2D grid. To improve efficiency, the rows were parallelized for each column by adding the line 'pragma omp parallel for' to the inner for loop. By doing so, the work of the algorithm

improves from $O(Wj)$ to $O(Wj/p)$ for W capacity, j number of items, and p processors. Further, the span for this algorithm is $O(jlog(W))$ due to the spawning of processors for each column in the parallel for loop.

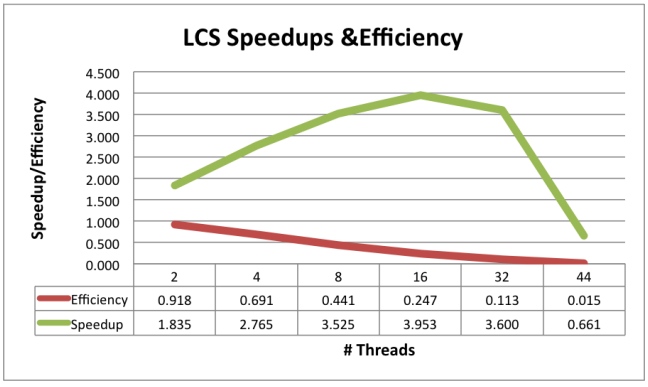
Similarly, for LCS[2], two nested for loops were implemented to traverse each antidiagonal of a 2D grid and within each antidiagonal, the cells were parallelized to be solved by a set of processors. The inner for loop was parallelized by applying a 'pragma omp parallel for'. For the work, the complexity resembles that of the Knapsack implementation, given every subproblem of the 2D grid must be solved. With dimensions of the grid defined by the lengths of each string, the work is improved to $O((string1length * string2length))/p$. The span also contains a log factor due to the spawning of processors for each diagonal. Consequently, the span becomes $O(maxstringlength * log(diagonallength))$ for the task of spawning a diagonal-length number of processors for each column, which is the size of the length of the longest string.

Shown below are the speedups and efficiency calculations for Knapsack and LCS when strong scaling the data with various thread counts: 2,4,8,16,32,44.



	2	4	8	16	32
Efficiency	0.85	0.82	0.79	0.72	0.62
Speedup	1.70	3.27	6.34	11.49	19.76

Figure 1: Speedups and Efficiency for Strong Scaling of Knapsack with 50,000 Capacity and 8,000 Items



	2	4	8	16	32	44
Efficiency	0.918	0.691	0.441	0.247	0.113	0.015
Speedup	1.835	2.765	3.525	3.953	3.600	0.661

Figure 2: Speedups and Efficiency for Strong Scaling of LCS with Strings of length 5,000

Both LCS and Knapsack achieved increasing speedups as the thread count grew; however, the speedup for LCS began to drop when using more than 16 threads. This may be due to the fact that the memory restraints of the parallel LCS implementation were quite expensive and the complexity of the algorithm was high given the number of if statements and conditionals needed to define each antidiagonal. For 2 threads, LCS was only slightly less than linear speedup, while the efficiency of Knapsack was also close to 1.

As a disclaimer, please note that these algorithms were parallelized for the purpose of providing a simple visual tutorial to introduce students to the concept of parallel algorithms. Thus, the implementation of LCS and Knapsack presented in this paper may not be optimal or achieve maximum efficiency and performance.

5 TSGL IMPLEMENTATION

After parallelizing each algorithm, the TSGL library and 'Canvas' class were imported ('include <tsgl.h>' and 'include <Canvas.h>') into the parallel C++ programs, and a canvas was built with a fixed size and background color.

```
Canvas c(0, 0, 1200, 700, "LCS Problem");
c.setBackgroundColor(WHITE);
```

Figure 3: TSGL Canvas function to draw canvas and set size and color

To build the grid, the drawRectangle() function was used with a set of size parameters (top, bottom, left, right) and a color parameter. For each row and column, the row width and column width were defined based on the length of the x-axis and y-axis of the grid, so as to dynamically update as the capacity and number of items, or length of characters for Longest Common Subsequence, changed. Given the row and column widths, drawLine() was used to add grid lines for each column and row.

```
c.start();
int x1 = 300;
int x2 = 800;
int y1 = 650;
int y2 = 150;
int size = x2 - x1;


c.drawText("LCS Problem", 450, 50, 30, RED);
c.drawRectangle(x1,y1,x2,y2,LIME,true);
```

Figure 4: TSGL drawRectangle() function to build 2D grid

Furthermore, for Knapsack, integrated into the nested for loops that traverse the grid are the drawText() and sleepFor(1) functions to draw the value of the knapsack at each iteration in the appropriate column and to add a lag of one second between the drawings by a set of processors so as to observe the processors drawing in each column every second. To observe the trace of each computation, drawLine() and two drawCircle() calls were added at each iteration to draw a line from the cell that was selected by the Knapsack algorithm to the current cell and to add circles on each cell to highlight the cells as the processors were performing that computation.

Algorithm 1 consists of pseudocode that summarizes the logic in (1) used to evaluate each Knapsack cell and illustrates the employment of several TSGL functions to visualize the work performed by each processor. A list of weights, list of values, number of threads, capacity, and final value are also drawn to the canvas (drawText()) for user reference.

Algorithm 1: Knapsack TSGL Algorithm for filling in 2D grid with values, lines, and circles

```
Result: Complete Knapsack grid
while canvas.isOpen() do
    for h from 1 to number of items do
        'pragma omp parallel for';
        for k from 1 to Capacity do
            if knap[k][h-1] < values[h] +
             knap[k-weights[h]][h-1] then
                knap[k][h] = values[h] +
                 knap[k-weights[h]][h-1];
                canvas.sleepFor(1);
                canvas.drawCircle(x,y of
                 knap[k-weights[h]][h-1]);
                canvas.drawCircle(x,y of knap[k][h]);
                canvas.drawLine(x,y of
                 knap[k-weights[h]][h-1], x,y of
                 knap[k][h]);
                canvas.sleepFor(1);
                canvas.drawText(knap[k][h]);
            else
                knap[k][h] = knap[k][h-1];
                canvas.sleepFor(1);
                canvas.drawCircle(x,y of knap[k][h-1]);
                canvas.drawCircle(x,y of knap[k][h]);
                canvas.drawLine(x,y of knap[k][h-1], x,y of
                 knap[k][h]);
                canvas.sleepFor(1);
                canvas.drawText(knap[k][h]);
            end
        end
    end
end
```

For Longest Common Subsequence, the values and trace lines are drawn similar to the Knapsack implementation, but rather than traversing each column, at every iteration the values are drawn in each antidiagonal by a set of processors with a lag of one second between drawings. Algorithm 2 consists of pseudocode for traversing each diagonal and evaluating all cells for their LCS value using (2). Depending on the length of the diagonal in comparison to the maximum and mininum length of the strings passed, the x and y indices for the grid are set accordingly. The evaluation of pairwise characters in each string is then performed and based on this outcome, the value in the corresponding cell in the grid is updated. Based on the location of the cell and the outcome of the evaluation, the value is drawn to that location on the canvas, a circle is placed on the 'winning' cell of the three in contention and on the current cell, then a line is drawn from the 'winning' cell to the current cell. Also provided on the canvas are the strings, number of diagonals, number of threads, and final value for user reference.

Algorithm 2: LCS TSGL Algorithm for filling in 2D grid with values, lines, and circles

```
Result: Complete LCS grid
while canvas.isOpen() do
    for d from 1 to number of diagonals do
        'pragma omp parallel for';
        for p from 1 to length of diagonal do
            if d <= min length of strings then
                set x=d-p+1, y=p;
            end
            if min length of strings < d <= max length of
             strings then
                set x=(min length of string - p + 1), y=(d -
                 min length of string + p);
            else
                set x=(min length of string - p + 1), y=(d -
                 min length of string + p);
            end
            if a[x] = b[x] then
                LCS[x][y] = 1 + LCS[x-1][y-1];
                canvas.sleepFor(1);
                canvas.drawCircle(x,y of LCS[x-1][y-1]);
                canvas.drawCircle(x,y of LCS[x][y]);
                canvas.drawLine(x,y of LCS[x-1][y-1], x,y of
                 LCS[x][y]);
                canvas.sleepFor(1);
                canvas.drawText(LCS[x][y]);
            else
                LCS[x][y] = max(LCS[x-1][y], LCS[x][y-1]);
                canvas.sleepFor(1);
                canvas.drawText(LCS[x][y]);
            end
            if max(LCS[x-1][y], LCS[x][y-1]) = LCS[x][y-1]
             then
                canvas.sleepFor(1);
                canvas.drawCircle(x,y of LCS[x][y-1]);
                canvas.drawCircle(x,y of LCS[x][y]);
                canvas.drawLine(x,y of LCS[x][y-1], x,y of
                 LCS[x][y]);
            else
                canvas.sleepFor(1);
                canvas.drawCircle(x,y of LCS[x-1][y]);
                canvas.drawCircle(x,y of LCS[x][y]);
                canvas.drawLine(x,y of LCS[x-1][y], x,y of
                 LCS[x][y]);
            end
        end
    end
end
```

To improve the user interface for both programs, a keyboard binding was added for drawing the trace of the computations upon initiation by the user after all cells of the grid are filled in. By adding this interactive component of pressing key 'A', users can first run the program to view the columns being filled in with values without the additional distraction of observing the circles and lines being draw on top of the grid. Additionally, command line arguments were added for the user to customize the parameters of each algorithm (e.g. specifying number of threads, strings for Longest Common Subsequence, etc.).

6 VISUALIZATIONS

An example of the LCS visualization while being drawn is shown in Figure 5. As depicted, the tenth antidiagonal is in the process of being filled by a set of 2 processors. Two cells in that antidiagonal were computed in the first time step and a lag of one second separates the next drawing from the previous drawing. In Figure 6, the tracing of each cell is shown. For comparing 'Happy' and 'Holidays!', it can be seen that 'H', 'a', and 'y' contribute to the overall LCS value of 3.

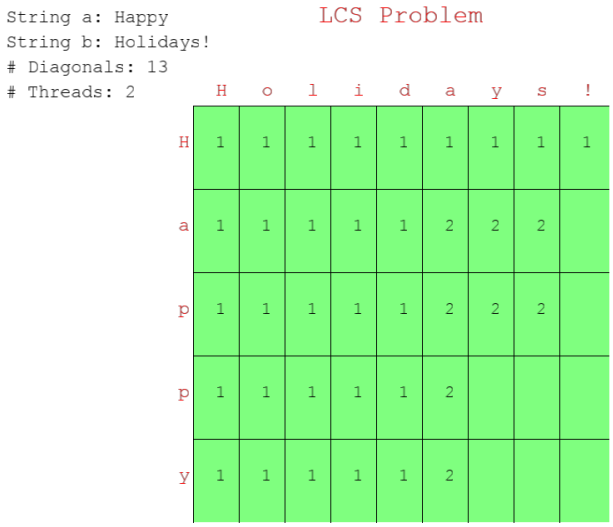


Figure 5: Longest Common Subsequence visualization in progress

In the visualization of the Knapsack in Figure 7, it is shown that the columns are being filled in consecutively and the fourth column is in progress. Upon completion of the grid, by clicking 'A' on the keyboard, a user can opt to draw the trace of the computations to see the cells selected by the algorithm at each step (see Figure 8). It can be seen that items 5,4, and 1 were selected and the total knapsack value was 11.

7 DISCUSSION

By constructing visualizations using TSGL for LCS and the Knapsack Problem, students can observe the effects of parallelizing algorithms and for this case, the process by which multiple processors

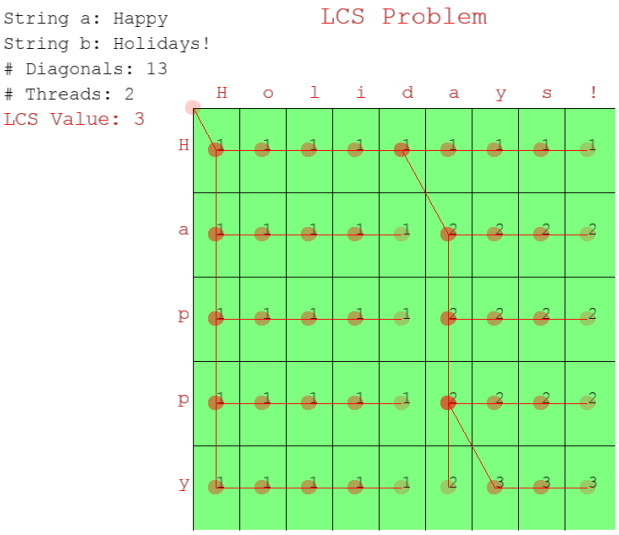


Figure 6: Longest Common Subsequence final visualization with tracing

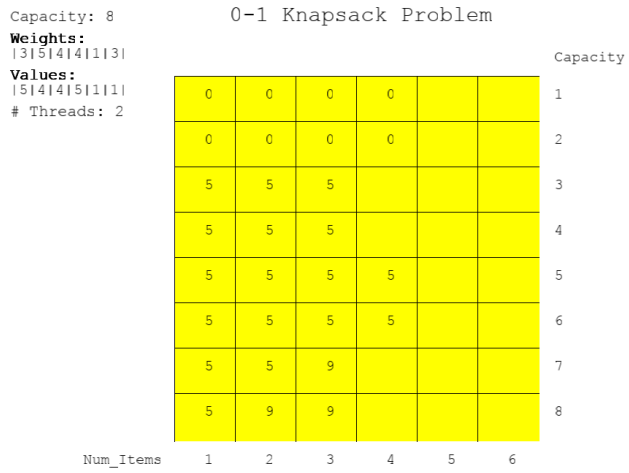


Figure 7: Knapsack visualization in progress

can work simultaneously performing the same computations for individual subproblems. Further, by parallelizing both LCS and Knapsack and building visualizations for each, a student can extend their understanding of parallel algorithms by comparing the need for different logic and methods used for Knapsack and LCS. Students can also compare the work and span of each of the parallel algorithms to determine the changes in performance.

As described throughout this project, due to the data dependencies of the computations for LCS and Knapsack, Knapsack can be parallelized across the rows of each column, whereas LCS cannot. Rather, the parallel implementation chosen for LCS was to parallelize the cells in each antidiagonal. Note that there exist other parallel implementations of each algorithm, particularly a more

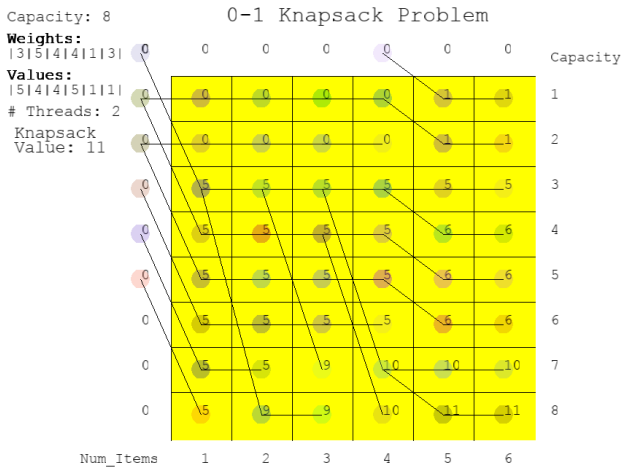


Figure 8: Knapsack final visualization with tracing

cache-efficient solution for LCS in which a 2D grid is divided recursively amongst processors into four quadrants (analagous to a Cartesian coordinate system) and the second and third quadrants are computed in parallel, while the first and fourth must be solved sequentially. For the purposes of a clean and consumable demonstration, it was best to implement each algorithm as discussed in this paper.

Moreover, while the examples in this project focused on relatively small problems, the algorithms can be applied to any size problem (i.e. any number of items and capacity for Knapsack, or any length of strings for LCS). The TSGL implementation was programmed to dynamically adjust to the size of the given problem. Although the canvas size is fixed, the 2D grid is not and will adapt to the specified parameters.

8 CONCLUSION

In conclusion, to enhance the teaching tools available for parallel algorithms, this paper describes a program built to visualize the parallelization of the Knapsack Problem and Longest Common Subsequence. Using the TSGL library and OpenMP, the program consists of draw functions and parallel for loops to iterate over a 2D grid to solve each problem using dynamic programming and to simultaneously and to draw each processor's solutions for their respective subproblems to a canvas. With the creation of this tool, students can have a tangible representation of parallelization and can actually see the processors computing in real time. The potential for this tool is significant, as its impact on computer science education could be immense. Applications in computer science for a visualizer of individual processors, whether in sequential or parallel mode, are countless, as courses such as computer systems, operating systems, and any variation of an algorithms course could benefit from such a tool.

ACKNOWLEDGMENTS

To Cody, for his resilience and continous assistance with the installation of TSGL. To Dr. Ballard for his support and keen eye for debugging my code and for assisting with the successful installation of TSGL.

REFERENCES

[1] Calvin College Computer Science Department. 2007. TSGL library. (7 2007). https://github.com/Calvin-CS/TSGL

[2] Amine Dhraief, Raik Aissaoui, and Abdelfettah Belghith. 2011. Parallel Computing the Longest Common Subsequence (LCS) on GPUs: Efficiency and Language Suitability. (10 2011), 143–148.