parallel_dp: The Parallel Dynamic Programming Design Pattern as an Intel® Threading Building Blocks Algorithm Template

Doug Serfass
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, Arkansas 72204
djserfass@ualr.edu

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, Arkansas 72204
pxtang@ualr.edu

ABSTRACT

Intel Threading Building Blocks (TBB) is an ideal environment for implementation of the parallel dynamic programming design pattern. The task-based parallelism of TBB readily lends itself to the realization of the participants and participant collaboration of this design pattern. We propose the parallel_dp algorithm template, an implementation of the parallel dynamic programming design pattern using TBB. We define the participants and the participant collaboration of this design pattern and describe how the design pattern is implemented by parallel_dp. We analyze the performance of our solution by applying parallel_dp to create four TBB programs that are parallel versions of the four types of dynamic programming algorithms. Our experimental results prove that parallel_dp provides speedup to all of our TBB programs and near linear speedup to one of our TBB programs. We conclude that parallel_dp will improve the performance of many TBB programs that include a dynamic programming algorithm.

Categories and Subject Descriptors

D.1.3 [Concurrent Programming]: Parallel programming; D.3.2 [Language Classification]: Concurrent, Distributed and Parallel Languages; I.2.8 [Problem Solving, Control Methods, and Search]: Dynamic Programming

General Terms

Algorithm, Performance, Design, Theory

Keywords

Multi-core Processor, Threading Building Blocks, Speedup

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACMSE ’13, April 4-6, 2013, Savannah, GA, USA.
Copyright 2013 ACM 978-1-4503-1901-0/13/04 ...$15.00.

2. RELATED WORK

Tan et al. [2] proposed a parallel pipelined algorithm to decompose the computation operators to solve nonsequential dynamic programming algorithms. Dios et al. [3] proposed that the parallelization of wavefront-based dynamic programming algorithms should use logical tasks as opposed to threads. Wu et al. [4] proposed a lightweight synchronization method that can fully exploit fine-grained pipeline parallelism for wavefront computations on multicore platforms.

3. DESIGN PATTERN

Dynamic programming [5] solves problems by combining the solutions to subproblems. It is a bottom-up technique, starting with the smallest, and hence the simplest, subinstances, saving their solutions in an n×n table, combining their solutions through use of the n×n table, obtaining answers to subinstances of increasing size, until arriving at the solution of the original instance.

3.1 Dependency

Dependency in a dynamic programming problem is the solution of a successor smallest subinstance based on the solution of a predecessor smallest subinstance. We say that the successor smallest subinstance is dependent on the predecessor smallest subinstance. Every dynamic programming problem will have at least one dependency. A dependency is a result of the recursive portion of a dynamic programming problem. If a dependency is defined by a constant reference to a previously solved row or column in the n×n table, then we say that it is a uniform dependency. If a dependency is defined by a nonconstant reference to previously solved rows or columns in the n×n table, then we say that it is a nonuniform dependency. Work by [3] and [2] has established a deliberate distinction between parallel solutions to dynamic programming problems with uniform
and nonuniform dependencies, respectively. We relax this distinction by using the four cardinal directions of north, east, south and west (as opposed to magnitude) to define a dependency. We illustrate this concept with a dynamic programming problem example that includes both uniform and nonuniform dependencies.

Our dependency dynamic programming problem example uses a function to calculate each smallest subinstance, $\bullet$, of the $n \times n$ table as shown in Figure 1. This calculation has two dependencies as indicated by the arrows in Figure 1(a). The right arrows represent a uniform dependency in that the solution of each smallest subinstance is dependent on the solution of another adjacent smallest subinstance immediately to its left. The down arrows (dashed and not dashed) represent a nonuniform dependency in that the solution of each smallest subinstance is dependent on the solution of another smallest subinstance somewhere above it. We remove the dashed down arrows and simply state that each smallest subinstance in Figure 1(a) has an east and a south dependency as shown in Figure 1(b).

![Figure 1: Uniform and Nonuniform Dependencies](image)

The dependency flow [3] for our dependency dynamic programming problem example is shown in Figure 3. The integers 0, 1 and 2 indicate the number of dependencies for each smallest subinstance. The upper left smallest subinstance has no dependencies. Every smallest subinstance in row 0 or column 0 (excluding the upper left) has one dependency. All other smallest subinstances have two dependencies. Dependency can be represented as a vector containing pairs of integers (with values of -1, 0 or 1) of the form (row, column), where each pair indicates the cardinal direction of a dependency arrow from a predecessor smallest subinstance to a successor smallest subinstance. Based on Figure 1(b), the vector for our dependency dynamic programming problem example would contain (0,1) (an east dependency) and (1,0) (a south dependency). The two additional allowable vector values are (0,-1) (a west dependency) and (-1,0) (a north dependency).

![Figure 3: Dependency Flow](image)

3.2 Front

A front is included in a dynamic programming problem if only part of the $n \times n$ table is used to calculate a solution. A front is defined as the largest collection of smallest subinstances, where: each smallest subinstance is located on the same diagonal; each smallest subinstance has no dependency; and each smallest subinstance is located on the diagonal that includes the first smallest subinstance solved by the dynamic programming problem. We illustrate a front with a dynamic programming problem example as shown by the $n \times n$ table in Figure 4 and the code in Figure 5. The numbers in Figure 4 indicate the number of dependencies for each smallest subinstance. The - for a smallest subinstance indicates that it is undefined. The first smallest subinstance solved by the dynamic programming problem is located at (0,0), as defined by the looping constructs on lines 4 and 5 in Figure 5. Each smallest subinstance on the main diagonal has no dependency. Therefore, a front is located on the main diagonal.

![Figure 4: Front](image)

3.3 Tile Table and Tile Size

Consider the dynamic programming problem as defined by the code shown in Figure 2, where each smallest subinstance has an east and a south dependency as shown in

```plaintext
for(int i=0;i<n;i++)
a[i] = rand(i);
for(int j=0;j<n;j++)
C[i][j]=rand(i+j);
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
C[i][j]=C[i][j]+f(C[i-g(a[i])][j], C[i][j-1]);
```
Figure 5: Front Code

Figure 6(a). We combine Figure 2 with the model as shown in Figure 6(b) to create a tiled sequential dynamic programming problem as defined by the code shown in Figure 7. Without violating the dependencies in Figure 6(a), we group smallest subinstances into tiles as shown by the rectangles in Figure 6(b). We maintain intratile dependencies and replace intertile dependencies with a single tile dependency. The resulting tiles and tile dependencies are known as a tile table.

Figure 6: Dependency Tables

Tile size, also known as grain size [1], is the number of smallest subinstances in one dimension of a tile. As shown on line 1 of Figure 7, tile size, \( t \), is a parameter to our tiled sequential dynamic programming problem. \( t \) is used on line 6 of Figure 7 to calculate tile dimension, \( d \), which is the number of tiles in one dimension of a tile table. Tile size is used to manipulate tile dimension and, consequently, the number of iterations on lines 13 and 14 of Figure 7. The code on lines 9 through 15 of Figure 7 is called by parallel_dp to execute a tile. As a result, tile size can be used by parallel_dp to specify the number of iterations for a reasonably sized tile of the \( n \times n \) table to deal out to a core.

Tile size amortizes parallel scheduling overhead. A tile size independent of the number of cores will usually maintain parallel scheduling overhead in constant proportion to real work. Packaging and handling overhead will be relatively constant per tile and, therefore, independent of the number of cores. Tile size enables avoidance of excessive parallel overhead. If the iterations on lines 13 and 14 in Figure 7 are too few, the overhead may exceed useful work. Specifying the correct tile size for a dynamic programming problem will limit overhead. Tile size sets a minimum threshold for parallelization.

Figure 7: Tiled Sequential Dynamic Programming Problem Code

3.4 Body Object

The body object (as shown in the code of Figure 8) is a function object that is derived from the tile iterator code (as shown on lines 9 through 15 of Figure 7) in a tiled sequential dynamic programming problem. The body object executes the calculations of a single tile in the tile table. The body object constructor on line 7 of Figure 8 has one argument, the dynamic programming problem-specific local variable \( C \). parallel_dp passes the values of \( i, j, d \) and \( n \) to the body object on line 4 of Figure 8.

3.5 Start and End Coordinates

The start and end coordinates are the row and column locations of the first (if no front) and last tile executed in the tile table. If there is a front, the start coordinate tile is executed in parallel along with all tiles with which it shares a diagonal. These coordinates are derived from the values of the initializer and test expressions of the tile table iterators (lines 7 and 8 of Figure 7) in a tiled sequential dynamic programming problem. They are defined by a pair of integers with values of 0 or \( d - 1 \) of the form (row, column).

3.6 Collaboration

Collaboration between the participants is started after the end coordinate tile executes either the front tiles (in parallel, if there is a front) or the start coordinate tile (if there is not a
The end coordinate tile waits for all tiles to execute. A tile executes the body object. After execution of the body object is complete, a tile communicates this information to successor tiles with which it shares a dependency. A tile then executes successor tiles that have satisfied all of their dependencies. Execution of the tile table is complete after the end coordinate tile is finished executing. At this point, the solution is available in the $n \times n$ table element as specified by the dynamic programming problem.

4. DESIGN PATTERN IMPLEMENTATION

The parallel_dp algorithm is shown in Figure 9. Lines 4 through 22 create and initialize a task graph, $x$, and lines 23 through 30 execute the task graph. In the following sections, we define how the parallel_dp algorithm implements the participants and the participant collaboration of the parallel dynamic programming design pattern.

1. algorithm template<typename Body> parallel_dp(int n,int t, const Body& body,vector<direction> dependency, bool is_front, int start_row,int start_col,int end_row,int end_col);
2. define x : task***; //declare task graph
3. define inc_row, inc_col : bool;
4. for row := 0 to n/t-1 //iterate task graph
5. for col := 0 to n/t-1
6. $x[\text{row}][\text{col}] = \text{new task_dp<Body>(}\text{row,}\text{col,}\text{body}); //populate tasks
7. for row := 0 to n/t-1 //iterate task graph
8. for col := 0 to n/t-1
9. for j := 0 to dependency.size()-1 //iterate dependencies
10. if dependency.at(j).get_row() == 0
11. inc_row := false;
12. else
13. inc_row := true;
14. if dependency.at(j).get_column() == 0
15. inc_col := false;
16. else
17. inc_col := true;
18. if inc_row && inc_col //does task have a successor?
19. do nothing; //no
20. else //yes
21. $x[\text{row}][\text{col}].\text{successor} += 1;
22. $x[\text{row}][\text{col}].\text{increment_ref_count}();
23. if n/t-1 != 0 //more than 1 task?
24. //yes, end task refcount number of dependencies plus 1
25. if is_front //front?
26. //yes, end task spawn front tasks and wait
27. else //no, end task spawn front tasks and wait
28. $x[\text{end_col}][\text{end_col}].\text{spawn_n_and_wait_for_all}(\text{task_list});
29. else
30. destroy(*$x[\text{end_col}][\text{end_col}]); //destroy end task

Figure 9: The parallel_dp Algorithm

4.1 Dependency

Lines 10 through 18 of Figure 9 use the row and column fields of each direction object that define a dependency to determine the successors for each task in a task graph. If a task has a successor, then line 21 increments the successor count for the task and line 22 increments the refcount of the successor task. Line 24 sets the end task refcount to the number of dependencies plus 1. This ensures that the end task will not execute until all other tasks in a task graph have executed.

4.2 Front

Line 25 of Figure 9 uses this value to determine if there is a front. If yes, then, on line 26 of Figure 9, the end task spawns all front tasks in the task graph and the end task waits for all tasks in the task graph to execute. If no, then, on line 28 of Figure 9, the end task spawns the start task in the task graph and the end task waits for all tasks in the task graph to execute.

4.3 Tile Table and Tile Size

The tile table is implemented in Figure 9 as a task graph. Lines 4 through 6 of Figure 9 use tile size, $t$, to calculate tile dimension, $d = n/t$, to iterate over the task graph and populate the task graph with tasks. Lines 7 through 22 of Figure 9 also use tile dimension to iterate over the task graph and all dependencies. Line 23 of Figure 9 uses tile dimension to determine if the number of tasks in the task graph is greater than 1.

4.4 Body Object

The body object participant is created by a user as a class as shown in Figure 8. The user passes a reference to the body object to parallel_dp as shown in Figure 9. This reference is passed to a task_dp object on line 6 of Figure 9 during the instantiation of each task in the task graph.

4.5 Start and End Coordinates

The start and end coordinate participants are used numerous times during execution of the task graph on lines 23 through 30 of Figure 9 to reference the start and end tasks, respectively, of the task graph. We describe this activity in the next section, participant collaboration.

4.6 Collaboration

Collaboration between the participants is started during creation of the task graph after tile size, $t$, is first used on line 4 of Figure 9 to calculate tile dimension, $d = n/t$. Tile dimension is in turn used to iterate over the task graph in order to create tasks and to initialize each task with information based on dependencies. Next, tile dimension is used to determine if the task graph contains more than one task. If the task graph does contain more than one task, then the end coordinate is used to reference the end task of the task graph to set the end task refcount to the number of dependencies plus 1. This ensures that the end task will not execute until all other tasks in the task graph have executed. If there is a front, then the end coordinate is again used to reference the end task of the task graph, which spawns all front tasks in the task graph. If there is not a front, then the end coordinate is used once again to reference the end task of the task graph, which uses the start coordinate to reference the start task of the task graph in order to spawn the start task. In either case (front or no front), the end task of the task graph then waits for all tasks in the task graph to execute. Each task in the task graph uses the body object to execute the calculations of a single tile in the tile table. After all tasks in the task graph (except the end task) have executed, the end coordinate is used to reference the end task of the task graph, which is executed. The end coordinate is used a final time to reference the end task of the task graph in order to destroy the end task.

5. PERFORMANCE ANALYSIS
Grama et al. [7] identified four types of dynamic programming algorithms. We used parallel_dp to create four TBB programs that are parallel versions of each of these four types. We executed each of the TBB programs five times on one through eight cores, \( p \), of an 8 core AMD Opteron CPU @ 2.8GHz and calculated the average execution time. Our problem size was \( n = 2^{11} \). We varied the tile size, \( t \), from \( 2^{11}, 2^{10}, \ldots, 2^{0} \). \( t = 2^{11} \) results in one tile and no parallelism. \( t = 2^{0} \) results in \( 2^{22} \) tiles and maximum parallelism. We converted all dynamic programming algorithms to tiled sequential dynamic programming algorithms and calculated the speedup (with \( p \) cores), \( S_p \), of an algorithm as a result of using parallel_dp as

\[
S_p = \frac{T_{ts}}{T_p},
\]

where \( T_{ts} \) and \( T_p \) are the tiled sequential and the parallel_dp (with \( p \) cores) execution times, respectively.

parallel_dp and \( p = 1 \) in Figures 10, 12, 14 and 16 is parallel code running on one thread, but still creating tasks and enforcing synchronization. The cost of scheduling and synchronization is evident in these Figures where parallel_dp and \( p = 1 \) is greater than sequential. parallel_dp and \( p = 2, 3, \ldots, 8 \) in Figures 10, 12, 14 and 16 show a typical bathtub curve [1] for execution time versus tile size. The upward slope on the right side indicates that, with a tile size of \( t = 2^{2} \), most of the execution time is spent on the overhead of a tile of work. An increase in tile size (moving to the left on a graph) results in a proportional decrease in parallel overhead. Then, the curve flattens out because overhead becomes insignificant for a sufficiently large tile size. At the extreme left, the curve turns up because the tiles are so large that there are fewer tiles than available hardware threads. Notice that, on each graph, any tile size over a wide range works quite well.

The execution time and speedup for the 0-1 Knapsack Problem [5], a serial monadic [7] algorithm, are shown in Figures 10 and 11, respectively. Figure 10 supports the assertion in [4] that a small variation in tile size can have a large impact on performance. As an example, for \( p = 2 \), \( t = 2^{5} \) is 40% faster than \( t = 2^{4} \).

The execution time and speedup for Floyd All Pairs Shortest Paths [5], a serial polyadic [7] algorithm, are shown in Figures 12 and 13, respectively. The time complexity of this algorithm is the largest order of our four algorithms. As a result, for this algorithm, parallel_dp generated the longest execution time.

The execution time and speedup for Longest Common Subsequence [6], a nonserial monadic [7] algorithm, are shown in Figures 14 and 15, respectively. For \( p = 1, 2, \ldots, 8 \) and \( t < 2^{5} \) in Figures 10, 12 and 14, the cost of scheduling and synchronization in parallel_dp is clearly evident as all three of the graphs spike sharply up at this point. Contrast this result with Figure 16, where the cost of scheduling and synchronization in parallel_dp is relatively constant for the entire graph.
The execution time and speedup for Optimal Binary Search Tree [6], a nonserial polyadic [7] algorithm, are shown in Figures 16 and 17, respectively. For $t = 2^3$ in Figure 17, parallel_dp exhibits near linear speedup comparable to the algorithms studied in [2] and [4]. Similar results have also been obtained by [8] for a recursive problem on an 8 core Intel® Xeon CPU.

### 6. CONCLUSIONS

Our experimental results prove that parallel_dp will speedup many TBB programs that include a dynamic programming algorithm. Future work is to tune the parallel_dp algorithm to increase performance.

### 7. ACKNOWLEDGMENTS

We thank Albert Everett, system administrator of our test computer. This work was supported in part by the National Science Foundation under Grant CRI CNS-0855248, Grant EPS-0701890, Grant EPS-0918970, Grant MRI CNS-0619069, and OISE-0729792.

### 8. REFERENCES


