10-31-2018

D2P: Automatically Creating Distributed Dynamic Programming Codes

Nikhil Hegde
Purdue University, hegden@purdue.edu

Qifan Chang
Purdue University, qchang@purdue.edu

Milind Kulkarni
Electrical and Computer Engineering, *Purdue University*, milind@purdue.edu

D2P: Automatically Creating Distributed Dynamic Programming Codes

Nikhil Hegde, Qifan Chang, Milind Kulkarni
School of Electrical and Computer Engineering, Purdue University, West Lafayette, USA
{hegden,qchang,milind}@purdue.edu

Abstract

Dynamic Programming (DP) algorithms are common targets for parallelization, and, as these algorithms are applied to larger inputs, distributed implementations become necessary. However, creating distributed-memory solutions involves the challenges of task creation, program and data partitioning, communication optimization, and task scheduling. In this paper we present D2P, an end-to-end system for automatically transforming a specification of any recursive DP algorithm into distributed-memory implementation of the algorithm. When given a pseudo-code of a recursive DP algorithm, D2P automatically generates the corresponding MPI-based implementation. Our evaluation of the generated distributed implementations shows that they are efficient and scalable. Moreover, D2P-generated implementations are faster than implementations generated by recent general distributed DP frameworks, and are competitive with (and often faster than) hand-written implementations.

Keywords Dynamic Programming, Recursive, Distributed-memory parallel, Autogen, Framework

1 Introduction

Dynamic Programming (DP) [8] algorithms are very efficient in solving problems arising in domains such as bioinformatics, mobile communication, and finance. As some of the problems operate over large data sizes, which can exceed the memory capacity of a single compute node, distributed-memory solutions become necessary to process the entire data. While most parallel implementations of DP algorithms are for shared-memory systems, there exist only a few hand-tuned distributed-memory implementations of DP algorithms [2, 4, 5, 35]. Creating distributed-memory implementations involves several challenges that do not arise in shared-memory systems. First, we need to partition the data and computation into *tasks* of appropriate granularity split among different compute nodes so that different nodes can process portions of the DP problem in parallel. Second, we need to insert communication between nodes to satisfy data dependences between tasks. Finally, we need to schedule tasks efficiently, to balance parallelism with communication overhead.

Because formulating an efficient DP algorithm is a different challenge than performing data partitioning and task creation, communication insertion, and task scheduling, we would like a system where the concerns of *creating* a DP algorithm can be separated from the concerns of *distributing* it. In particular, we would like a system that automates as much of the distribution process as possible, allowing programmers to focus on simply designing DP algorithms.

Recent work has shown that efficient *recursive* formulations of DP algorithms outperform iterative counterparts on shared-memory systems [11, 12]. The primary reason is better cache utilization due to a top-down or depth-first approach in recursive formulations as opposed to a bottom-up or breadth-first approach of iterative formulations. These recursive formulations have the property that each recursive "task" (think: invocation) touches a subset of the data of its parent task (think: caller method). This approach provides a key advantage for distributed execution: the data-dependencies of a recursive method are inclusive (i.e. the collective data-dependencies of all the recursive sub-invocations within the body of the method are a subset of the dependencies captured by method's arguments) and hence coarse-grained parallel tasks, suitable for a task-parallel model of computation, can be easily identified. This is the key insight we exploit in *automatically* creating distributed-memory implementations of recursive DP algorithms.

Automating the creation of distributed-memory implementations requires automatic resolution of the associated challenges. Specifically, we need to (i) automatically partition the data among multiple nodes, (ii) infer data-dependencies, and then (iii) establish communication channels for the purpose of exchanging data to satisfy data-dependencies. While much of prior work has addressed these challenges in different contexts for iterative codes, recursive programs are not explored as extensively. Existing systems [6, 30, 31, 34] either work on iterative codes or adopt a profiling-based approach and assume a regular communication pattern in the case of recursive programs [31]. Efficient recursive DP programs involve irregular communication patterns and hence, previous techniques are not effective. Currently, shared-memory implementations of these recursive DP programs can be generated automatically [22]. However, there do not yet exist tools for automating the creation of these programs on distributed-memory platforms.

Overview

In this paper, we present D2P, an end-to-end system that takes in an iterative code snippet (without any parallelism or partitioning specifications) capturing all the read and

write accesses in a DP program, and produces an MPI based, distributed-memory parallel code with recursive methods.

D2P builds on a tool called Autogen [10], which takes as input an iterative code snippet of the DP program and produces a *specification* of the recursive DP algorithm. The specification is a shared-memory pseudocode, with explicit annotation for parallelizable code regions, consisting entirely of a set of recursive methods calling each other.

D2P takes as input the pseudocode from Autogen and produces a distributed-memory implementation. In order to do this, D2P first unfolds the recursion until a sufficient number of tasks are generated to keep the compute nodes busy. The leaves of the recursion tree are recursive method invocations and represent the tasks to execute. The method parameters capture the data regions read and written within the method invocation (due to the inclusion property mentioned above).

D2P then determines data dependences by inspecting leaf methods for overlapping read and write regions. D2P also merges some leaves to coarsen the task granularity. After performing merging (coalescing), the remaining tasks are partitioned among compute nodes in a deterministic manner known to all nodes (facilitating determination of communication later). The nodes follow an owner-computes rule (single owner for a data region) during computation.

D2P then inserts communication—identifying and establishing communication channels between compute nodes to send/receive data as demanded by the task allocation. By default, intra-node parallelism in D2P is expressed through the use of Cilk [18] parallelism constructs. The result is a "single program multiple data" (SPMD) implementation employing asynchronous communication capable of running on a distributed memory machine.

The contributions of this paper are:

- We build an end-to-end system that takes an iterative code snippet of DP algorithms and outputs MPI-based distributed-memory implementations of these algorithms.
- We provide an interface for application programmers to create distributed-memory implementations for any recursive DP algorithm.
- We evaluate the scalability of our generated MPI-based code implementations and measure speedups achieved w.r.t. baseline implementations, preprocessing overheads, and compare against existing distributed memory implementations.

Our evaluations show that D2P implementations scale well and can admit extremely large problem sizes much beyond the memory capacity of a single node. We also observe that the overheads of task partitioning (recursion unfolding, dependency determination, and task assignment) are negligible compared to the actual computation. Finally, we find that a D2P generated program scales better compared to other distributed-memory implementations. The D2P-generated implementation of Smith-Waterman performs from 33× to 483× faster than an implementation generated by an existing framework for generating distributed implementations of *iterative* DP formulations. Moreover, the D2P-generated implementation even outperforms a *hand-written*, application-specific implementation of Smith-Waterman in most cases, and is only 27% slower in the worst case.

2 Background and Motivation

In this section we discuss some of the essential background needed to understand D2P. First, we briefly discuss dynamic programming with a concrete problem, which serves as a running example throughout the paper. We discuss what it means to design iterative, naïve recursive, and efficient recursive algorithms for this problem. We then discuss the Autogen tool in order to understand how efficient recursive algorithms can be automatically generated. We conclude this section with a discussion on the advantages that recursive formulations offer for distributed implementations.

2.1 Dynamic Programming

Dynamic Programming is based on the principle of divide-and-conquer to find an optimal solution to a variety of problems. Two fundamental properties of problems that admit DP solutions are *optimal substructure* and *overlapping subproblems* [14]. The optimal substructure property is necessary for combining optimal solutions of simpler subproblems in order to find an optimal solution of a problem. The overlapping substructure property ensures that the subproblems repeat. We illustrate these properties with an example, minimum weight triangulation (MWT) of a polygon [23].

MWT is used in applications such as finite element method, and computational geometry, among others. It is an instance of the parenthesis problem [19]. As Figure 1 shows, the goal in MWT is to partition a convex polygon into triangles such that the edges do not intersect while minimizing the sum of the edge lengths of the component triangles. The intuition behind the recurrence equation in Figure 1(a) is to partition the polygon into a simple triangle and sub-polygon(s) to the left and/or right of the simple triangle, compute the solutions for each part separately, then combine the solutions. If the sub-polygons have an optimal solution, then the combined solution is guaranteed to be optimal (optimal substructure property). The sub-polygons can be computed as separate triangulations (overlapping subproblems property). This corresponds to computing $Cost(2, 4)$ in the naïve recursive implementation of Figure 1(b), which also shows the recursion tree with $Cost(0, 4)$ as the root representing the cost of triangulating the entire pentagon. This property of DP problems contrasts with traditional divide-and-conquer solutions, in which the subproblems do not repeat. The naïve recursive pseudocode is inefficient and leads to exponential

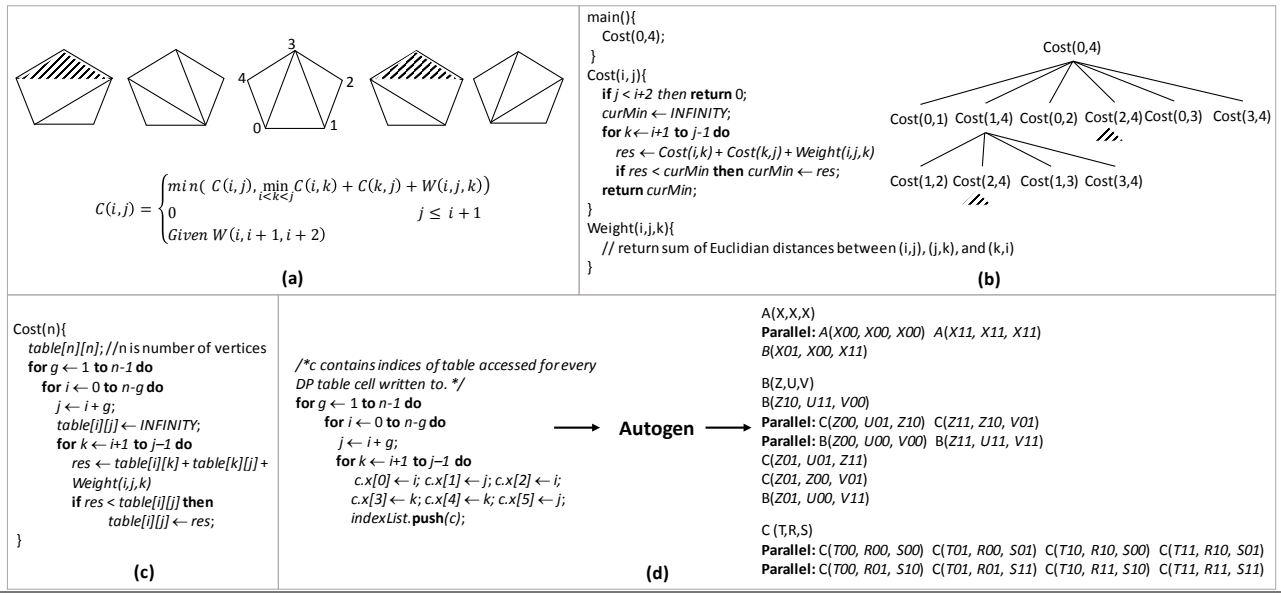

Figure 1. The Minimum Weight Triangulation problem: (a) shows all possible triangulations of a pentagon. Every triangulation has an associated cost that depends on some notion of distance between the vertices of the pentagon. The recurrence equation calculates the least cost. (b) shows the pseudocode of a naïve recursive implementation and the resulting recursion tree. Shaded triangles in the recursion tree highlight repeated subproblems. (c) shows an iterative pseudocode that avoids re-computing solutions for a repeated subproblem. (d) shows an efficient recursive pseudocode generated from Autogen.

time costs because of not reusing the computation already performed due to a repeated subproblem.

The iterative pseudocode of Figure 1(c) overcomes this inefficiency by storing the results of subproblems in *table*, allowing previously-computed results to be reused, but requiring an additional $O(N^2)$ space. Every iteration of the middle `for` loop can be run in parallel in this code. This parallelization strategy reflects *wavefront-parallelism*, which is a popular approach for parallelizing iterative DP codes. However, the order of computation performed in this approach—starting on the main diagonal of a matrix, and then ending at top-right corner—exhibits poor temporal locality. This is because the wavefront-parallel code performs bottom-up computation of finding optimal solutions to all subproblems of smaller sizes first before moving on to a bigger subproblem (e.g. computing all points on the main diagonal before computing points on the smaller adjacent diagonal).

Efficient recursive algorithms take a top-down or depth-first approach to problem decomposition and solution. As a result they have better temporal locality compared to iterative algorithms [12], which adopt a bottom-up or breadth-first approach. Hereafter, we refer to the efficient versions when mentioning recursive algorithms. Designing recursive DP algorithms that have better temporal locality is not straightforward. Autogen [10] is a tool that automates the design of such recursive algorithms.

2.2 Autogen

Autogen [10] is a tool to discover efficient, recursive, parallel, cache-oblivious DP algorithms from iterative specifications of DP recurrences. The output is a pseudocode of a non-trivial, parallel, recursive DP algorithm.

Figure 1(d) shows how Autogen is used to produce a recursive MWT algorithm. Note the similarity between the input to Autogen and the pseudocode of Figure 1(c). This implies that for an existing real-world application, an end-user needs to translate only the embedded loop structure representing the DP table recurrence equation and feed it to Autogen. Autogen stores and analyzes the read and write accesses to the DP table cells within the loop structure of the pseudocode. The dependences among DP table cells are translated to region-wise dependences, where regions are groups of DP table cells. Every unique access pattern corresponds to a distinct recursive method. More details on how Autogen discovers the recursive methods and defines their bodies can be found in the Autogen paper. The output is a pseudocode consisting of a set of recursive methods calling each other. The `main` function is expected to call the top-level recursive method, always named `A`, with arguments `X`, which represent a single tile corresponding to the entire DP table. Also, it is assumed that the first parameter of a recursive method is the tile written and the remaining parameters represent the tiles read before the tile is written. The output pseudocode exhibits better temporal locality because of solving hierarchically decomposed subproblems and combining their

solutions. Furthermore, shared-memory parallel recursive DP algorithms adapt better in the presence of cache sharing, are an order of magnitude faster, and have more predictable runtimes than their tiled iterative counterparts [10].

2.3 Distributed-memory parallelism

Recursive algorithms, in particular, have advantages when it comes to distributed-memory implementations: parallel recursive invocations are naturally easier to adapt to a task-parallel model of computation as opposed to a data-parallel model, which is a good fit for iterative DP algorithms. The iterative codes, as we saw in Figure 1(c), could have multiple layers of `for` loops with regular and irregular data accesses. This poses a challenge for inferring the data partitioning needed for distributed-memory parallel codes. Data dependency analysis is also complicated in iterative codes when compared to recursive codes that exhibit inclusive dependencies: in the case of the recursive DP codes output from Autogen, every method invocation accesses a set of tiles, which is the super-set of the union of the tiles accessed within all recursive invocations done in the method body. In other words, the method arguments specify a bound on the entire data regions read/written. This makes it easier to reason about data-dependencies when we try to automate data-partitioning. We next explain how D2P takes advantage of these properties of recursive DP algorithms in overcoming the challenges associated with automating the generation of distributed-memory parallel codes.

3 Design

Automating the generation of distributed-memory implementations from serial implementations involves multiple challenges: (i) partitioning control and data to create tasks of appropriate granularity, (ii) determining data dependences and identifying parallelism, (iii) identifying communication channels and insert communication code, and (iv) scheduling tasks in parallel to minimize execution time and communication overhead. However, many of these challenges are simplified in the context of D2P. Because the input to D2P is a recursive algorithm, a task in D2P can be a recursive method invocation. However, we do not create a task for every method invocation to avoid creating too many fine-grained tasks. Control partitioning is much simplified due to the absence of branches in the Autogen generated recursive pseudocode. As the recursive method arguments capture inclusive data dependences, reasoning about data dependences and hence data partitioning is also simplified. We also need not identify parallelism as the input to D2P has explicit parallelism specification ('parallel' annotation from Autogen, Figure 1(d)). We let the underlying task-parallel runtime system handle the scheduling of tasks. We next look in detail how these challenges are addressed in D2P.

3.1 Task creation, partitioning, and communication

The preprocessing stage in D2P consists of task creation, dependency inference, and partitioning. Figure 2 summarizes these phases.

Task creation via recursion unrolling Recall from the discussion on Autogen in Section 2.2 that the `main` program invokes the top-level recursive method `A` with a set of arguments representing the entire DP table to update. If each method invocation is considered as an independent task, we have just a single task to begin with. In order to generate more tasks, we expand (unroll) the body of the top-level recursive method. Any order of the parallel method invocations in the algorithm of Figure 1(d) can be considered while unrolling. Unrolling is equivalent to hierarchically decomposing the DP table to compute solutions of smaller subproblems. Unrolling results in a recursion tree, whose leaves (L) represent the tasks computing the smaller subproblems. Deciding the optimal granularity of these tasks—i.e., number of levels of recursion to unroll—is hard. Typically, when executing a recursive parallel program on a single compute node, the tradeoff is between the overhead of creating new tasks and the amount of parallelism exposed. For a distributed-memory implementation, the number of levels of recursion to unroll, D, reflects a tradeoff between communication overhead and the available parallelism: a larger D introduces more tasks, which may increase the available parallelism but also introduces more data dependences, and hence communication. Note that unrolling decomposes the DP table computation into smaller subproblems computing the tiles of the table. Figure 2(a) shows unrolling for the MWT problem when D is two.

Identifying data dependences via effect intersection Figure 2(b) shows the data dependences among tasks resulting from unrolling. Correct dependences can be derived after computing a linearization of the tree in preorder traversal (to ensure that dependences are resolved in program order). The tiles resulting from unrolling are numbered in a deterministic manner known to all processes. The dependency structure, as the arrows indicate, is derived from the tile numbering information, which can be computed using simple set intersection (of the tile numbers). The source of the arrow is a task writing to a DP table tile (producer) and the destination is a task that reads these tiles (consumer). Enforcing these data dependences is all that is necessary to ensure correct execution, due to the inclusion property of the dependences as described earlier.

Note, interestingly, that because D2P performs unrolling of the original recursive algorithm, it is able to expose *more* parallelism than the original formulation. This is because the original formulation uses control dependences (in the form of sequential ordering instead of `parallel` annotations)

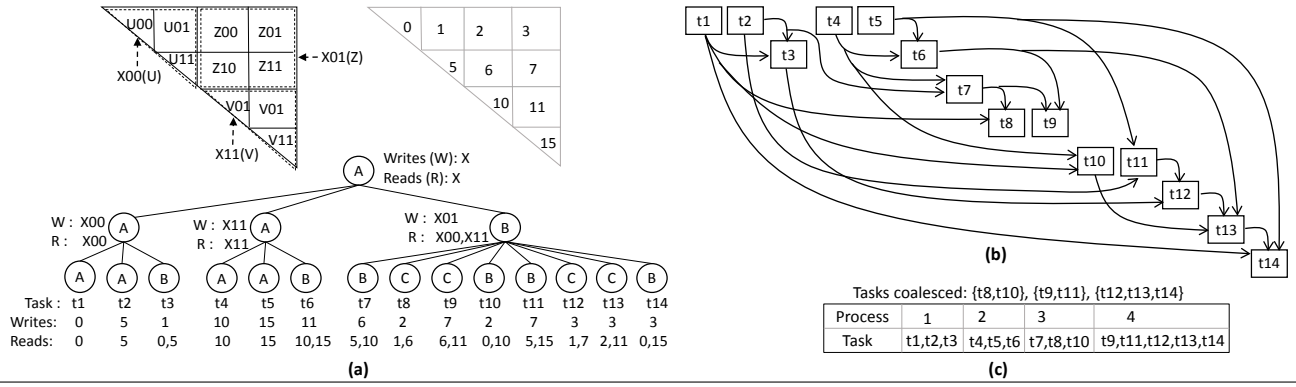

Figure 2. Preprocessing in D2P: (a) shows decomposition of the DP table into subproblems (numbered tiles and corresponding identifiers used in recursive pseudocode of Figure 1(d)) after two levels of unrolling of the MWT recursive algorithm. Unrolling results in the shown recursion tree, whose leaves (L) are identified as tasks. (b) shows dependences among tasks. (c) shows a blocked partitioning scheme applied to the task distribution among 4 processes.

to conservatively enforce data dependences that arise between coarse-grained tasks. Two coarse-grained tasks x and y may have to execute sequentially because some subtask in the y is dependent on some subtask in x, even though *other* subtasks in y can execute in parallel with x. As a concrete example, note that in the original recursive formulation of MWT in Figure 1(d), the first `B` task executes sequentially after the parallel execution of the first two `A` tasks. However, in the unrolled dependence graph of Figure 2(b), we see that this is conservative: t_7 (arising from a `B` task) can execute in parallel with t_3 and t_6 (arising from `A` tasks) despite the control dependence in the original formulation. D2P enforces exactly these data dependences, and hence is able to expose more parallelism than the original parallel recursive implementation, depending on how much unrolling is done. Of course, performing additional unrolling can incur additional runtime overhead and communication, so this tradeoff must be carefully managed.

Task coalescing and partitioning D2P uses an *owner-computes* strategy: DP table tiles are assigned to processes (P), and then *all* tasks that write to that tile are assigned to the same process. Note that this simplifies communication, as we do not need to consider different processes writing to the same tile. Tasks writing to the same tile are coalesced prior to distributing all the tasks among different processes so that task-to-process mapping results in a single owner for a tile. Given owner-computes, all that remains is to assign tiles to processes. D2P provides the programmer with control over this distribution scheme (see Section 4). The complete list of coalesced tasks in MWT resulting in a total of 10 tasks (considering coalesced tasks as a single task) and a blocked partitioning scheme distributing these 10 tasks among 4 processes is shown in Figure 2(c).

Communication insertion In a distributed-memory setting, if the consumed data does not reside on local memory, then the producer and consumer of the data must establish communication channels and perform communication. Due to the SPMD nature of the code, all processes participate in unrolling and determining dependences. However, a process determines dependents (and consumer IDs) of a task only if it is the producer of that task. While unrolling, a process examines the write parameter (first parameter) of a recursive method invocation (associated with the task) to know if it is the producer of the task, since, given a tile number, every process can find out the owner of the tile. If the process is not a producer, it examines the read parameters (remaining parameters) to check if the actual owner can be added as a potential consumer of any tasks for which the process is a producer. This enables a producer to determine the consumer IDs for the purpose of sending data and the receiver to expect a message from the correct sender. Overall, this distributed dependency determination scheme in D2P takes $O(L/P)$ time.

Leaf-task implementation One advantage of D2P is that end users can rely on D2P for orchestrating inter-task parallelism as shown in Figure 2 while continuing to use their highly-optimized, single-process code within a task. Because a task computes a recursive method with inclusive dependences, users can replace the method body with codes offloading computation to multiple threads, vectorizing processors, GPUs etc. By default, D2P uses Autogen's shared-memory, task-parallel, recursive approach (see Section 2)

4 Implementation

Parameters - preprocessing D2P uses a few user-configurable parameters provided at runtime for the purpose of preprocessing and tuning. By default, D2P adopts a simple heuristic of unrolling until there are at least as many tiles in a row or column as the number of processes. Hence, in this scenario, D depends on the number of processes P. However, D2P

also lets users choose a D based on the maximum number of processes that would be used in program execution. D determines the structure of the task graph, which is built at runtime. The input size information is used in inferring the number of DP table cells per row or column, which is required to compute the owner of a tile given the tile number.

The tiles are numbered based on the partitioning scheme, which is specified by the user as a parameter. By default, D2P uses a column-cyclic distribution. In many DP problems, dependences exist along rows and columns of the DP table, and parallelism is along the diagonal (MWT) or anti-diagonal. A column-cyclic distribution of the tiles means that column-wise communication can be avoided, while still leading to tasks along diagonals and anti-diagonals being assigned to different processes. This reduces communication while maintaining parallelism for many DP problems.

Parameters - intra-task parallelism While D2P primarily concerns itself with identifying and exploiting inter-task parallelism, as shown in Figure 2(b), each task itself represents a fairly coarse-grained computation. D2P leverages the resulting *intra*-task parallelism—easily exposed due to the recursive nature of the implementation—by using Cilk `spawn` [13] to implement Autogen's `parallel` sections (though, as described in Section 3, the user can instead replace a leaf recursive implementation with their own optimized single-process code).

During code generation, D2P inserts a Cilk `sync` at the end of parallel sections as the synchronization is implicit in the Autogen produced code. While executing a recursive method body, spawning sub-tasks down till the recursion base case can greatly increase the parallel overhead. The spawn cutoff parameter, C, controls this overhead by executing sequential version of the recursive method body, without `cilk_spawn and cilk_sync`, till the base case is reached. We set C to two levels beyond D, which means that each resultant task (after unrolling) mapped to a process spawns sub-tasks for up to two levels down the recursion. For all the benchmarks and the input sizes considered, this created a sufficient number of sub-tasks. Cilk workers execute the sub-tasks and D2P provides a knob to set the number of Cilk workers per process at execution time. The total number of available Cilk workers for a process on a node is equal to the number of cores divided by the number of processes mapped to a node. By default, we set the number of workers per process to 1 so that it is easier to map one process per core and evaluate a process-level decomposition.

Going deeper into the recursive method body, the recursion may proceed all the way down to computing a single cell or a sub-tile of the DP table. The recursion base case, B, defines when to stop the recursion. The default value of B is 1 in D2P since, by default, Autogen produces a code that computes a single cell in the base case.

API	Description
ReadInput	Captures input-data. For example, reading 2-dimensional coordinates of points from a file in MWT
InitTable	Captures algorithm-specific initialization of the DP table. For example, MWT initializes all the elements on the main diagonal to zero
BaseCase	Recursion base case. Specifies the exact function applied in computing a DP table cell. For example, for MWT, $table_{i,j}$=min($table_{i,j}$,$table_{i,k}$+$table_{k,j}$+$weight_{i,k,j}$).
GetScore	Fetches optimal score. Since the DP table is distributed over multiple processes, aggregate functions MAX, MIN are provided to fetch the maximum/minimum value among DP table cells. Specific cell content can be obtained using GetCell(i,j). For example, in MWT, the optimal score is always in the cell at top right corner. GetCell(0,N-1) fetches the data from the process owning the cell.

Table 1. APIs in D2P.

Programmer interface The application programmer interfaces (APIs) in D2P are designed to capture application-specific properties such as initializing the DP table, defining the base case, fetching the optimal score, and reading inputs. The APIs in D2P and their purpose are described in Table 1. These APIs abstract away the details of distributed-memory programming and let an application programmer focus on the algorithm aspects. The APIs need to be implemented in Autogen and we provide reference implementations for six benchmarks. Note that since the original input to Autogen is an iterative skeleton code (Figure 1(d)), complete automation in the form of source-to-source translation of the input shared-memory implementation of an iterative algorithm to a distributed-memory implementation of a recursive algorithm is not yet available and is future work.

5 Evaluation

We present the evaluation results of D2P with six DP based applications. These applications are drawn from the domains of bio-informatics, computational geometry, and computer science. The applications differ in their data dependency pattern and are spread over the different categories of DP problems mentioned in Galil et al. [19]. We begin with the scalability results and demonstrate that the distributed-memory implementations of these DP applications scale well and can handle problem sizes much beyond the memory capacity of a single node. Finally, we present case studies comparing D2P implementation of a benchmark with other works. We compare D2P with DPX10 [34], a generic framework for implementing distributed-memory DP algorithms. We also compare a D2P generated program with a hand written distributed-memory program [2] implementing a popular algorithm in bio-informatics. Our results show that the performance of D2P is significantly better than DPX10 and that D2P not only outperforms but also scales much better than the hand-written code in general cases.

5.1 Methodology

The distributed-memory (DM) implementations of DP algorithms evaluated are based on the Autogen produced shared-memory pseudocode. Our baselines are shared-memory implementations of recursive DP algorithms, as formulated by Autogen. Being recursive implementations, they are optimized for locality and represent the best single process sequential implementations of each benchmark among the approaches we tested. Section 5.2 shows a comparison of our single process recursive (1_*rec*) and iterative (1_*ite*) implementations for a subset of the benchmarks. We assume that the results hold for remaining benchmarks based on the findings from the Autogen paper.

The overall computation in a DP application can be broadly divided into three steps: 1) table initialization, 2) table computation, and 3) backtracking. Backtracking is an important step constructing the optimal solution based on the optimal scores computed in the second step. As step 2 is computationally dominant in a majority of the DP problems, step 2 alone is timed in all performance measurements unless otherwise noted. The runtimes are measured using wall-clock time and every configuration of a test is run until a steady state is achieved.

Benchmarks

i **Minimum Weight Triangulation (MWT)** [23] is a triangulation algorithm, detailed in Section 2.

ii **Matrix Chain Multiplication (MCM)** [14] finds the optimal way to associate a sequence of matrix multiplications. Like MWT, MCM is an instance of the *parenthesis* problem [19] and has a similar dependency structure in its DP algorithm. Computing a single DP table cell requires reading from $O(N)$ cells in both, and they compute only the upper triangular matrix of the DP table.

iii **Smith-Waterman Local Alignment (SW)** [33] determines the similarity of two DNA (or amino acid) sequences. All pairs of possible *subsequences* from both the sequences are compared and scored rather than considering whole sequences. The algorithm finds local regions within the sequences having an optimal similarity score.

iv **Needleman-Wunsch Global Alignment (NW)** [27] is another sequence alignment algorithm. In contrast to SW, NW does a *global alignment* by considering the entire sequence, rather than subsequences, from the given pair of input sequences. Both NW and SW compute the entire matrix in the DP table in the same order and have the same dependency structure. Hence, the Autogen produced algorithm is the same in both cases, with slightly different base cases.

v **Floyd-Warshall All Pairs Shortest Path (APSP)** [17] finds the shortest path between every pair of vertices in a graph. APSP computes the entire DP table matrix. SW, NW, and APSP are all instances of the *gap* problem [19] with SW and NW being special cases having $O(1)$ dependencies whereas in APSP, computing a cell requires reading $O(N)$ cells.

vi **Nussinov 4D (N4D)** [29] predicts the structure produced as an RNA strand folds onto itself. Our N4D implementation stores similarity values of all possible pair of subsequences, leading to a 4-dimensional DP table. Computing a DP table cell requires reading $O(1)$ cells.

Input	Description	Size	Used with benchmark
Dros	mRNA sequence of Drosophila arizonae	14630 (bases)	N4D
Ecoli1 Ecoli2	Complete genome FASTA sequences of Escherichia coli. Ecoli1=NC_000913.2E, EColi2=BA000007.2	Ecoli1=4.6x10^6 Ecoli2=10x10^6 (bases)	SW, NW
AS	The CAIDA Autonomous Systems Relationships Dataset. A vertex in the graph is an internet service provider (ISP) and the directed edges are relationships.	26,475 vertices. 106,762 edges	APSP
Synth1	Convex hull of randomly generated points in 2-dimensional integer space	65,536	MWT
Synth2	Randomly generated integer weights, each of which is in the range (0,1000). A weight value represents either matrix row or column dimensions	262,145	MCM

Table 2. Data sets used in the evaluation

Table 2 describes the data sets [3, 32] used.

Development and execution environment The distributed memory implementations of all benchmarks use C++11, Intel MPI [1], and Intel CilkPlus [13]. We compile the programs with mpiicpc, a wrapper compiler for ICC 16.0.3. The experiments are run on a cluster of 10 nodes, interconnected by Gigabit ethernet. Each node of the cluster runs RHE Linux workstation release 6.10, contains two 10-core Intel Xeon-E5-2660 processors with 32KB L1 data and instruction cache, and 256KB of L2 cache, a shared 25MB L3 cache, and 64GB main memory. In our multi-process experiments, we accessed up to 8 nodes of the cluster and mapped processes to as many nodes as available on the cluster first before using additional cores on a compute node.

5.2 Scalability

Strong scaling In these experiments, we measure the performance of the benchmarks with increased number of computing resources (processes, Cilk workers) for a fixed input size. The input sizes are chosen such that the single-process DM executions complete in a reasonable amount of time and that the entire DP table can be allotted memory in a contiguous space for comparison with a single process baseline.

Figure 3 shows the strong-scaling results with the default D2P configuration of 1 Cilk worker per process. Since the scalability plots of MCM and NW are similar to that of MWT and SW respectively, we show the scalability plots of only four benchmarks in the figure. The plots show speedup of a DM execution over the sequential recursive baseline (1_*rec*). Table 3 shows baseline runtimes. The recursion base case, B, is 1 by default unless otherwise noted. We observe that

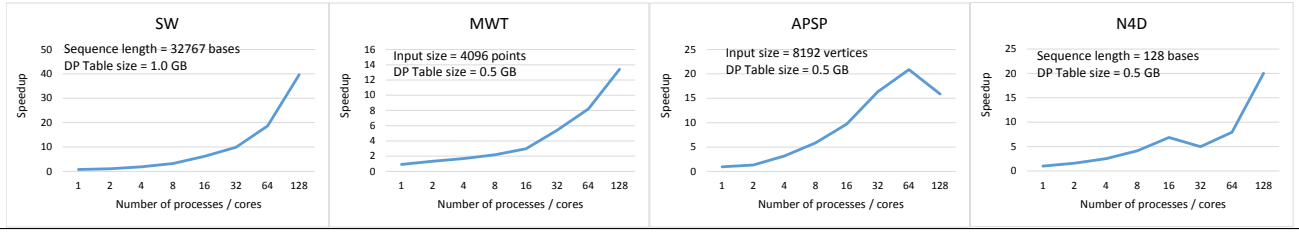

Figure 3. Strong scaling in D2P benchmarks showing speedup of 1 Cilk worker/process configuration over baseline.

Processes	MCM			MWT			SW			APSP			N4D		
1_ite	47.4s			293.4s			14.6s			-			-		
1_rec	43.6s (B=8)			215.8s			31.7s			4570.6s			331.4s		
	Pre	Cilk-workers		Pre	Cilk-workers		Pre	Cilk-workers		Pre	Cilk-workers		Pre	Cilk-workers	
		1	16		1	16		1	16		1	16		1	16
1_DM	0.4s	0.4×	0.7×	0.3s	0.9×	4.8×	1.2s	0.8×	1.9×	2.2s	0.9×	7.5×	0.3s	0.9×	9.3×
2_DM	0.4s	0.7×	3.9×	0.3s	1.3×	15.3×	0.8s	1.1×	2.2×	1.7s	1.3×	2.3×	0.13s	1.6×	6.2×
4_DM	0.2s	1.3×	7.1×	0.2s	1.7×	28.1×	0.4s	1.9×	2.7×	0.9s	3.1×	5.9×	0.1s	2.5×	6.6×
8_DM	0.1s	2.2×	12.5×	0.1s	2.2×	45.9×	0.3s	3.2×	3.2×	0.5s	5.9×	15.9×	0.09s	4.1×	10.4×
16_DM	0.1s	3.1×	12.5×	0.09s	2.9×	52.6×	0.2s	6.2×	1.3×	0.3s	9.7×	35.1×	0.08s	6.9×	11.8×
32_DM	0.09s	3.8×	7.4×	0.08s	5.4×	22.2×	0.1s	9.9×	1.2×	0.2s	16.3×	37.8×	0.07s	5.0×	10.1×
64_DM	0.08s	4.2×	4.8×	0.07s	8.2×	14.5×	0.08s	18.6×	2.1×	0.14s	20.9×	33.6×	0.07s	8.0×	8.5×
128_DM	0.08s	2.9×	0.9×	0.08s	13.4×	3.8×	0.07s	39.6×	2.1×	0.1s	15.9×	6.9×	0.07s	20.1×	9.6×

Table 3. Measurements showing i) iterative (1_*ite*) and recursive (1_*rec*) baseline runtimes in seconds (s). ii) preprocessing overhead (*Pre*), and iii) Comparison of speedups (×) obtained over recursive baselines in 1 and 16 Cilk worker per process configuration for the inputs used in Figure 3.

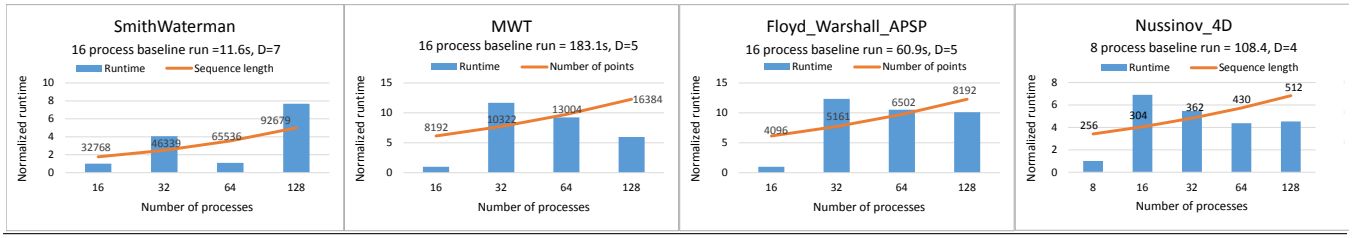

Figure 4. Weak scaling in D2P benchmarks. The y-axis (for the bars) shows normalized runtime w.r.t. the 16-process run (except Nussinov 4D, where we normalize w.r.t. an 8-process run). The lines (with labels) give the input size used for that execution.

in general with $B > 1$, the recursive versions are faster than the iterative versions, 1_*ite* (MWT is faster even with $B = 1$). Overall, in the default D2P configuration, we get a 20.3× geomean speedup, which further increases to 35.5× when we exploit intra-task parallelism.

We investigate exploiting Cilk parallelism within processes by assigning more than one Cilk worker per process. Table 3 shows speedups in 1 and 16 Cilk worker per process configurations. We performed a sweep of 1 to 16 Cilk workers per process at each scale, however, we show here only a part of these results due to lack of space. Since the base case computation in SW and NW differs only in the values used for scoring, the performance numbers of NW are very similar to SW. Hence, we do not show them here. We find that in SW and N4D, multiple Cilk workers do not help improve the performance relative to single Cilk worker configuration: the best speedups seen in SW (39.6×), and N4D (20.1×) are with a single Cilk worker. In contrast, the best speedups seen in MCM (12.5×), MWT (52.6×), APSP (39.9×) are with 16 Cilk workers per process. Our experiments showed that allocating parallelism to additional processes rather than additional Cilk workers consistently gave better performance, even though using more processes requires more communication. We thus believe that there is some performance issue with the single-process Cilk implementation, though we have not yet determined the root cause. The table also shows preprocessing overheads, and the speedups seen earlier are inclusive of the preprocessing overheads. The overhead remains under 1.7% in most cases and decreases with increase in the number of processes due to the $O(L/P)$ time (see Section 3.1, communication insertion).

Overall as the results show, D2P implementations scale well given the inherently strong data dependencies in DP problems. The addition of more Cilk workers improves performance in general, and intra-process parallelism is necessary in order to get the absolute best performance. The results reflect D2P's use of coarse-grained, task-level parallelism at the level of leaves of the tree (see Section 3.1,

recursion unrolling) and Cilk workers to exploit an intra-process, sub-task parallelism available within a task.

Weak Scaling In these experiments, we increase the input size with increasing number of processes while keeping the per-process computation fixed. We also fix the unroll depth, D, to override the default D2P configuration of increasing D as more processes are added. As a result, the runtime sometimes decreases with more processes due to more effective utilization of available parallelism. In weak-scaling experiments in general, we expect the communication overhead to increase proportional to the number of processes. In D2P benchmarks, we have an additional source of communication overhead: as some benchmarks read $O(N)$ (APSP, MWT, MCM) data to compute a cell, the amount of communication *per cell* can increase with scale. Additionally, when the input size is not a perfect power-of-two the recursive decomposition does not evenly divide the table, so partitioning tasks/sub-tasks computing these tiles can result in load imbalance. As a result, we expect the overall runtime to increase with scaling up the inputs and number of processes.

Figure 4 confirms this behavior. The X-axis shows the number of processes and the Y-axis shows runtime normalized to a 16-process baseline. The secondary line plot shows the input sizes for reference. We see that the runtimes show different degrees of increase for different benchmarks. The figure shows that the 32 process runtime is greater compared to the 64 process runtime in SW. This is because of load imbalance in the former run. The 16 process run in SW uses an input sequence length of 32,768, which is perfect-power-of-two. Because SW is $O(N^2)$ algorithm, the next perfect-power-of-two input size is used when the processes are scaled to 64. We consider scaling from 16 to 64 as an indicator of true weak-scaling since these correspond to perfect load-balance. For MWT and APSP, $O(N^3)$ algorithms, 16 and 128 process runs represent the load-balanced configurations. In N4D, which is $O(N^4)$, 8 and 128 process runs are each perfectly balanced.

In a perfectly load-balanced scenario, SW, MWT, APSP, and N4D weak-scale differently. SW weak-scales the best and shows the least increase of 1.08× in runtime due to a $O(1)$ amount of communication per cell. Even though MWT and APSP have both $O(N)$ dependencies in computing a cell, MWT shows 5.9× increase while APSP shows a 10.1× increase in runtime. This is because MWT only computes an upper triangular matrix. Hence, the communication overhead is reduced by half. N4D, despite computing $O(N^4)$ cells, shows a modest 4.5× increase in runtime. This is because, like SW, each cell in N4D depends on $O(1)$ other cells.

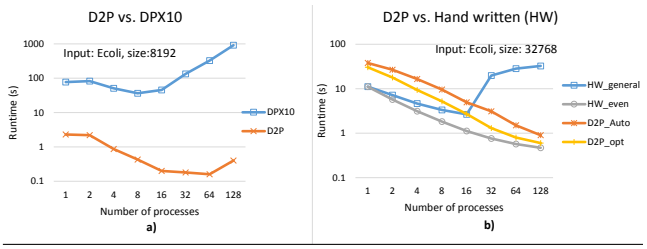

Figure 5. Case studies comparing D2P with other systems.

5.3 Case Studies

D2P vs. DPX10 We compare D2P with DPX10 [34], the best available framework for generating distributed implementations of *iterative* DP programs. DPX10 captures node-level, and thread-level parallelism for iterative DP programs when the computation is partitioned across nodes of a compute cluster. Node-level parallelism is captured through X10 Places and the thread-level parallelism through pthreads.

We compare the performance of SW benchmark only, since, DPX10 currently supports only DP programs with 2-dimensional DP table and $O(1)$ dependency in computing a cell (2D/0D) [34]. DPX10 takes a master-slave approach in executing tasks. First, a master process creates a DAG of tasks based on a user-specified dependency pattern. The tasks are then distributed among slaves, who report back to master with results. Each slave (and master) coordinate task execution with the help of an additional per slave (and master) scheduler processes. DPX10 recommends a single X10 place per compute node and multiple threads to engage the cores on a node.

Figure 5(a) shows the strong scaling results. In this experiment we chose the best performing DPX10 configuration among different choices of X10 threads available per place. We observe that the single process runtime of D2P is 33× faster than DPX10. Since DPX10 stops scaling beyond 16 places (more than two place per node) and D2P continues to scale up to 128 processes, the performance gap widens and we see that D2P achieving a speedup from 33× to 483×. We attribute the reason for superior performance of D2P to its implementation choices and the recursive algorithm advantage [10] over the iterative codes in DPX10. In addition, we believe that the control overheads in DPX10 (e.g. runtime task management, and of a dedicated process for scheduling tasks) make it slower than D2P.

Case Study II: D2P vs. Hand_Written (HW) In this case study we compare D2P with a highly optimized, hand written, distributed memory implementation of iterative SW algorithm [2]. Like DPX10, the HW implementation takes a master slave approach. The master divides the work (row-wise) evenly among slaves and computes any rows remaining due to an uneven division. The row-wise division means that a process depends on only a single row of data owned by a remote process. Note that this is in contrast to D2P, where

a producer sends an entire tile of data even though only a single row (or column) of a tile is required by the consumer.

Figure 5(b) shows the strong scaling results. The HW code is evaluated with input sizes where slaves evenly divide the work (*HW_even*), and where master computes some rows (*HW_general*). D2P is also evaluated with an auto generated implementation (*D2P_auto*) and a modified implementation communicating only a row or column of the tile (*D2P_opt*).

We observe that D2P is competitive. In the general case, *HW_general*, the master process waiting for all the data from slaves in order to compute its rows necessitates an MPI `AllGather` communication besides creating a need to allocate space for the entire DP table. D2P does not have any of these bottlenecks and as a result D2P scales much better than HW in the general case and even outperforms beyond 16 processes. When master does not compute any rows, there is no bottleneck. As a result, *HW_even* scales very well. Even in this ideal scenario, the best *D2P_opt* performance (0.6s) is only 27% slower than HW (0.47s).

6 Related work

Dynamic programming [8] algorithms, with their application in a wide variety of domains, are the target of parallelization efforts [19, 26, 35] as multi-cores and commodity clusters become increasingly accessible.

Frameworks simplifying parallel programming and exploiting multiple levels of parallelism on heterogeneous architectures have been proposed [7, 16, 24, 30, 34]. While D2P is aligned with these works on the end goal, there are some major differences: D2P specializes in recursive DP programs in comparison with OMPD [24], Legion [7], and others [6, 30, 31], which are more general. Both Legion and D2P leverage the inclusion property of recursive formulations. However, Legion relies on region annotations to determine dependencies, which would still need to be generated by something like D2P's unrolling and analysis. Among important prior works that automate the generation of distributed-memory parallel programs, some [6, 24] rely on a convenient starting point of explicit parallelism specification (e.g. `#pragma omp parallel for`) and implicit partitioning (implicit assignment of iterations to threads), while others [30, 31] work well on regular/irregular/mixed iterative codes. D2P instead relies on Autogen produced `parallel` specifications, performs explicit partitioning and works on recursive codes. Sarkar et al. [31] do an extensive study of the serial to parallel program conversion problem. They automate program partitioning in a functional programming setting, and explore compile- and runtime scheduling techniques on multi-processors with different system architectures. We could augment the existing, recursion unrolling influenced partitioning scheme in D2P with sophisticated inspector-executor based techniques [20, 30, 31]. Also, compared to these systems, D2P could be thought of as a semi-automatic framework specializing in recursive DP programs. Semi-automatic because the input is an iterative code snippet rather than an entire program, thus necessitating the introduction of APIs to capture user input. Like D2P, FastFlow [4], EasyHPS [16] and its successor DPX10 [34] focus on simplifying the creation of distributed-memory DP programs through system provided APIs and user-configurable parameters. However, unlike D2P, they work on iterative DP codes.

Galil et al. [19] design parallel iterative DP algorithms and categorize DP problems based on the data-dependency patterns. Chowdhury et al. [10] design efficient recursive DP algorithms and automate the design process through Autogen. They also show that recursive DP algorithms exhibit better temporal locality and outperform their iterative counterparts [11, 12]. Bellmania [22] takes a program synthesis approach in generating parallel shared-memory implementations of recursive pseudocodes output from Autogen. Lifflander et. al. [25] show that cache performance can also be improved for a broader set of divide-and-conquer problems with recursive implementations. They use programmer specified 'effect' annotations for identifying fine-grained parallelism in recursive programs implementing stencil computations. Based on the results, different recursive method invocations are spliced (interleaved) effectively to improve cache locality. The parallelism expressed in D2P is explicit at a coarser level of recursive method invocation. Because the method parameters specify the entire read and write access regions, effect annotations are redundant for the recursive methods in D2P. However, through the use of effect annotations, we could extend our distributed-memory code generation scheme to arbitrary recursive programs.

Majority of the prior work on distributed-memory DP implementations focus on problem-specific customizations [2, 3, 5, 9, 21, 26, 28, 35, 36]. While some focus on algorithmic optimizations [21, 26, 36], others [5, 15, 28] have system-specific optimizations. For example, in optimizing Smith-Waterman algorithm for distributed-memory, a majority [15, 28] exploit the coarse-grained parallelism available in comparing a sequence against a database of sequences. Whereas, D2P's implementation of Smith-Waterman parallelizes the algorithm, which computes the DP table to compare two sequences.

7 Conclusions

In this paper, we presented D2P, an end-to-end system for auto-generating distributed-memory implementations of recursive DP algorithms. We presented how D2P generates those implementations starting from a code snippet of an iterative DP algorithm. In the process, we showed how D2P uses Autogen and transforms its output to produce distributed-memory implementations from shared-memory recursive parallel pseudocodes of DP algorithms. Finally, we presented

an evaluation of D2P that showed that the generated implementations scale well, preprocessing overheads are negligible, and that D2P significantly outperforms the best available system for parallelizing DP programs on distributed-memory systems and is competitive against hand written programs.

8 Acknowledgements

The authors would like to thank Rezaul Chowdhury for providing the source code of Autogen, and Samuel Midkiff for providing access to the Wabash cluster. This work was supported by a DOE Early Career award (DE-SC0010295).

References

[1] [n. d.]. Intel MPI Libraries:. https://software.intel.com/en-us/intel-mpi-library. ([n. d.]).

[2] [n. d.]. SmithWaterman with OpenMPI and OpenMP. https://www.alexjf.net/projects/distributed-systems/smith-waterman-openmp-and-openmpi/. ([n. d.]).

[3] [n. d.]. The CAIDA AS Relationships Dataset, Nov'5, 2007. http://www.caida.org/data/as-relationships/. ([n. d.]).

[4] Marco Aldinucci, Massimiliano Meneghin, and Massimo Torquati. 2010. Efficient Smith-Waterman on multi-core with FastFlow. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 195–199.

[5] Matthew Baker, Aaron Welch, and Manjunath Gorentla Venkata. 2014. Parallelizing the Smith-Waterman algorithm using OpenSHMEM and MPI-3 one-sided interfaces. In *Workshop on OpenSHMEM and Related Technologies*. Springer, 178–191.

[6] Ayon Basumallik and Rudolf Eigenmann. 2006. Optimizing irregular shared-memory applications for distributed-memory systems. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 119–128.

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. IEEE, 1–11.

[8] Richard Bellman. 2013. *Dynamic programming*. Courier Corporation.

[9] Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cave, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. 2013. Integrating asynchronous task parallelism with MPI. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 712–725.

[10] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C Kuszmaul, Charles E Leiserson, Armando Solar-Lezama, and Yuan Tang. 2016. Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 10.

[11] Rezaul Alan Chowdhury, Hai-Son Le, and Vijaya Ramachandran. 2010. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 7, 3 (2010), 495–510.

[12] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 207–216.

[13] CilkPlus [n. d.]. Intel Cilk Plus. `https://www.cilkplus.org`. ([n. d.]).

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[15] Aaron E Darling, Lucas Carey, and W Chun Feng. 2003. *The design, implementation, and evaluation of mpiBLAST*. Technical Report. Los Alamos National Laboratory.

[16] Jun Du, Ce Yu, Jizhou Sun, Chao Sun, Shanjiang Tang, and Yanlong Yin. 2013. EasyHPS: A multilevel hybrid parallel system for dynamic programming. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 630–639.

[17] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962), 345.

[18] Matteo Frigo, Charles E Leiserson, and Keith H Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *ACM Sigplan Notices*, Vol. 33. ACM, 212–223.

[19] Zvi Galil and Kunsoo Park. 1994. Parallel algorithms for dynamic programming recurrences with more than O (1) dependency. *J. Parallel and Distrib. Comput.* 21, 2 (1994), 213–222.

[20] Kang Su Gatlin and Larry Carter. 1999. Architecture-cognizant divide and conquer algorithms. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 25.

[21] Khaled Hamidouche, Fernando Machado Mendonca, Joel Falcou, Alba Cristina Magalhaes Alves de Melo, and Daniel Etiemble. 2013. Parallel Smith-Waterman Comparison on Multicore and Manycore Computing Platforms with BSP++. *International Journal of Parallel Programming* 41, 1 (01 Feb 2013), 111–136. https://doi.org/10.1007/s10766-012-0209-6

[22] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 145–164.

[23] GT Klincsek. 1980. Minimal triangulations of polygonal domains. *Annals of Discrete Mathematics* 9 (1980), 121–123.

[24] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel Midkiff. 2012. A hybrid approach of OpenMP for clusters. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 75–84.

[25] Jonathan Lifflander and Sriram Krishnamoorthy. 2017. Cache locality optimization for recursive programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1–16.

[26] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2014. Parallelizing dynamic programming through rank convergence. *ACM SIGPLAN Notices* 49, 8 (2014), 219–232.

[27] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443 – 453. https://doi.org/10.1016/0022-2836(70)90057-4

[28] Mahdi Noorian, Hamidreza Pooshfam, Zeinab Noorian, and Rosni Abdullah. 2009. Performance enhancement of smith-waterman algorithm using hybrid model: Comparing the mpi and hybrid programming paradigm on smp clusters. In *Systems, Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on*. IEEE, 492–497.

[29] Ruth Nussinov, George Pieczenik, Jerrold R Griggs, and Daniel J Kleitman. 1978. Algorithms for loop matchings. *SIAM Journal on Applied mathematics* 35, 1 (1978), 68–82.

[30] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J Ramanujam, Atanas Rountev, and P Sadayappan. 2015. Distributed memory code generation for mixed irregular/regular computations. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 65–75.

[31] Vivek Sarkar. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA.

[32] Stephen T Sherry, M-H Ward, M Kholodov, J Baker, Lon Phan, Elizabeth M Smigielski, and Karl Sirotkin. 2001. dbSNP: the NCBI database of genetic variation. *Nucleic acids research* 29, 1 (2001), 308–311.

[33] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195 – 197. https://doi.org/10.1016/0022-2836(81)90087-5

[34] Chen Wang, Ce Yu, Shanjiang Tang, Jian Xiao, Jizhou Sun, and Xiangfei Meng. 2016. A general and fast distributed system for large-scale dynamic programming applications. *Parallel Comput.* 60 (2016), 1–21.

[35] Wenduo Zhou and David K Lowenthal. 2006. A parallel, out-of-core algorithm for RNA secondary structure prediction. In *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 74–81.

[36] Michal Ziv-Ukelson, Irit Gat-Viks, Ydo Wexler, and Ron Shamir. 2008. A faster algorithm for RNA co-folding. In *International Workshop on Algorithms in Bioinformatics*. Springer, 174–185.