1 Introduction

The goal of this project is to investigate how to build a domain-specific synthesis system to generate efficient parallel implementations of dynamic-programming (DP) algorithms automatically. DP algorithms build optimal solutions to a problem by combining together optimal solutions to many overlapping subproblems. DP algorithms exploit this overlap to explore otherwise exponential-sized problem spaces in polynomial time, making them central to many important applications ranging from logistics to computational biology.

Recent research by members of our team has shown that it is possible to achieve order-of-magnitude performance improvements over the standard implementation approach for DP by developing *cache-oblivious* implementation strategies that recursively partition the problem into smaller subproblems. These strategies exhibit better temporal locality, and the partitioning can expose more optimization opportunities (see, e.g., [25]). The catch is that such implementations can be incredibly complex—as shown in Section 3, a problem whose naïve implementation can be written in five lines of code can easily turn into a complex collection of mutually recursive routines. Moreover, different platforms often require different implementation choices, which greatly affect the performance in ways that are difficult to predict in advance.

The research vehicle for this project is ***Bellmaniac***,[1] a domain-specific compiler for DP applications. Bellmaniac will automate the process of developing parallel, cache-oblivious implementations of DP algorithms. In its final form, the system will take as input a high-level specification describing a DP problem and will produce an efficient implementation tailored to a specific architecture. The system will combine high-level algorithm exploration driven by a knowledge base of algorithmic tricks with low-level autotuning to ensure that it produces close-to-optimal code for a particular architecture.

Bellmaniac will also serve as a case study for a new approach for building domain-specific synthesizers by relying on general synthesis and verification technology. Previous successful domain-specific code generators for high-performance—such as FFTW [37, 38], ATLAS [110], TCE [11], Spiral [74], and our own work on Pochoir [103]—all rely on a combination of autotuning and symbolic manipulation of high-level program representations to produce implementations that would have been prohibitively difficult to generate by hand. The symbolic manipulation is performed by semantics preserving transformation rules together with heuristics to guide their application. These transformation rules and their associated heuristics capture much of the domain knowledge embedded in the system and allow it to explore the space of possible implementations efficiently and without the risk of introducing bugs. Bellmaniac will replace these transformation rules with parameterized ***tactics*** that describe high-level transformations of the program. Unlike the transformation rules, the tactics are not guaranteed to be semantics preserving; instead, the ***Tactic application engine*** will rely on constraint based synthesis to infer the parameters of each tactic under the constraint that the result must be provably equivalent to the initial program. This means that instead of having to prove that a transformation is correct under all possible scenarios, the system will prove the correctness of transformations on a case-by-case basis, making the system more flexible and easier to develop.

The overall structure of the proposed Bellmaniac compiler is shown in Figure 1. The front-end parser will compile the user-provided program into a ***functional-AST***, or F-AST, describing the problem. This F-AST will be transformed by the ***Tactic application engine***, which will produce a new F-AST that reflects the more sophisticated implementation. This new F-AST will be passed to a ***Base compiler***, which will transform the F-AST into an imperative AST from which the code will finally be generated. The tactics that direct the optimization process will be generated by a ***Tactic synthesis engine***, which will produce sequences of tactics together with their parameters. The synthesis engine will combine simple heuristics about what strategies are applicable to particular problems with feedback from the tactic application engine

[1]Richard Bellman invented dynamic programming.

about whether a tactic application failed or succeeded. In cases where multiple tactics can be applied on a given function, the synthesis engine may decide to apply all of them and produce different versions of the same function. Therefore, at the end of the transformation process, Bellmaniac will have a space of potential algorithms from which to generate code.

The ***code generator*** will generate two versions of the code: ***tuning code***, which will generate a ***plan*** that describes tuning decisions made based on timing measurements, and ***runtime code***, which executes the actual DP according to the plan. The tuning code will be based on a domain-aware "clearbox" optimization strategy, rather than a general-purpose "blackbox" strategy, since Bellmaniac knows the internal structure of the code it generates. We propose to investigate a domain-aware clearbox autotuning framework for Bellmaniac that can achieve better and more portable performance in considerably shorter time than blackbox autotuning frameworks.

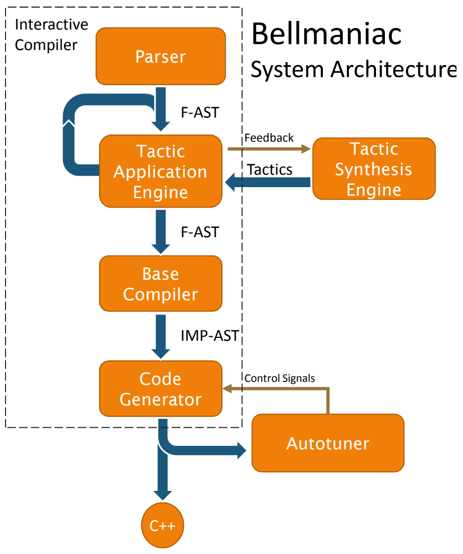


Figure 1: Overall system architecture

This approach to Autotuning will make it possible for us to explore different implementation strategies specialized for individual architectures. Our principal focus will be multicore computers, since they are already in the hands of application experts (and ordinary people) everywhere, and they are the building blocks for heterogeneous and scalable systems. Today's multicore computers contain CPU's with a few complex processing cores, but we also propose to explore how to compile for CPU's with many simple cores, such as the Intel MIC architecture [57] and the Intel Polaris chip [56].

Bellmaniac will be designed with a modular architecture and will be released as an open source project as a way to increase its impact to the community. As part of this effort, we will also assemble a benchmark suite of dynamic programming problems drawn from a variety of disciplines. In addition to helping us develop our system, we hope this benchmark suite will help to catalyze interest in this field by making it possible for other researchers in the community to build upon our work and evaluate their results. The implementations generated from this benchmark suite will also be made available as easy-to-use libraries and will contribute to the broader impact of the project.

We have created an early prototype of Bellmaniac that demonstrates the use of parameterized tactics and the feasibility of automatically verifying the correctness of the transformations by generating verification conditions that are proved using an off-the-shelf theorem prover [75]. Moreover, the project builds on the expertise we have in the area of cache-oblivious algorithms, having first invented the concept [40], as well as pioneered its wide-spread application to dynamic programming [21, 23, 25, 26]. The project also leverages our expertise from previous projects such as the Cilk multithreading technology [16,17,41,55,67], the Sketch synthesizer [96], and the FFTW [37, 38] and Pochoir [103] domain-specific code generation systems.

This experience gives us confidence in our research direction and provides a good starting point for our three-year research effort to address many new research challenges. In particular, we are able to leverage our research on the Pochoir compiler to target a new domain. Pochoir is a domain-specific compiler for ***stencil computations***, where each cell of a d-dimensional grid is repeatedly updated as a function of itself and neighboring cells. On multicore systems with as few as 12-cores, Pochoir achieves $100\times$ performance improvements over naive serial looping code, due to parallelism, cache-efficiency, and vector pipelining, as well as a host of small performance optimizations. Pochoir is limited to nearest-neighbor calculations, however, and it synthesizes an executable from a description of the local calculations it must perform. In contrast, Bellmaniac directly attacks the problem of synthesizing an executable from a high-level description of the recurrence relation governing the dynamic program. By combining the techniques of constraint-based synthesis pioneered in our work on Sketch, in effect by marrying the Pochoir and Sketch technologies, we aim to make Bellmaniac a fully automated synthesis system for DP, the first such system for any domain.

2 Dynamic programming

Dynamic programming (DP) [12, 95] allows certain combinatorial problems over an exponential search space to be solved in polynomial time and space. DP has proved useful in a wide variety of application areas, including stochastic-systems analysis, operations research, combinatorics of discrete structures, biological-sequence analysis, flow problems, and parsing of ambiguous languages [45]. DP is extensively used in biosequence analysis, such as protein-homology search, gene-structure prediction, motif search, analysis of repetitive genomic elements, RNA secondary-structure prediction, and interpretation of mass spectrometry data [8, 34, 50, 109]. A recent textbook [34] on biological sequence analysis lists 11 applications of DP in bioinformatics in its introductory chapter with many more in chapters that follow.

Dynamic programs are usually described through recurrence relations that essentially specify how the cells in a DP table must be filled up using solutions already computed for other cells. In this proposal, we will use the ***parenthesis problem*** [30, 42]—the problem of finding the optimal way of parenthesizing a product of n terms—as a running example. This DP problem encompasses RNA secondary-structure prediction, optimal matrix-chain multiplication, parsing context-free grammars, construction of optimal binary search trees, and optimal polygon triangulation. For all these applications, the cost of a parenthesization is given by the recurrence relation:

$$c[i,j] = \begin{cases} x_j & \text{if } 1 \le i = j-1 < n, \\ \min_{i<k\le j}\{c[i,k] + c[k,j] + w(i,k,j)\} & \text{if } 1 \le i < j-1 < n, \end{cases} \tag{1}$$

where the x_j are given for $j \in [1,n]$. We assume for simplicity that $w(\cdot,\cdot,\cdot)$ is a function that can be computed in constant time without additional memory accesses. Figure 2 illustrates the cell dependencies in the recurrence.

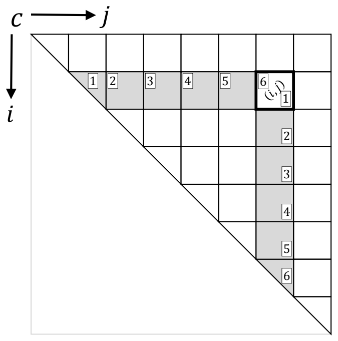


Figure 2: Cell dependencies in the DP table for the parenthesis problem (Recurrence (1)).

```
LOOP-PARENTHESIS( $c$, $n$ )
(Input is an $n \times n$ matrix $c[1:n, 1:n]$ with $c[i,j] = x_j$ for $1 \le i = j-1 < n$ and $c[i,j] = \infty$ otherwise (i.e., $i \ne j-1$). This function updates all $c[i,j]$ with $1 \le i < j-1 < n$ based on Recurrence (1).)
    1. for $t \leftarrow 2$ to $n-1$ do
    2.     parallel for $i \leftarrow 1$ to $n-t$ do
    3.         $j \leftarrow t+i$
    4.         for $k \leftarrow i+1$ to $j$ do
    5.             $c[i,j] \leftarrow \min\{\, c[i,j],\ c[i,k] + c[k,j] + w(i,k,j) \,\}$
```

Figure 3: Loop-based parallel code for Recurrence (1).

Standard implementations of most DP recurrences are based on nested loops. Figure 3 shows loop-based code for recurrence Recurrence (1). Observe that although the outermost loop cannot be parallelized, the loop in line 2 can be parallelized. Additionally, the loop in line 4 can be parallelized using reducers [39], although as a practical matter, parallelizing the loop in line 2 suffices to produce ample parallelism.

Loop-based codes, as that in Figure 3, are straightforward to understand, can be implemented fairly simply, have good spatial locality, and receive good support from hardware prefetchers. But looping codes suffer in performance from poor temporal cache locality. Indeed, when the DP table is too large to fit into cache, scanning the entire table over and over again means that no significant portion of it can be retained in the cache for reuse. In other words, any temporal locality inherent in the computation is almost completely ignored. Low temporal locality leads to increased pressure on memory bandwidth, which may not be observable on a single core, but scaling the number of active cores leads to palpably poor relative performance. Therefore, there is significant room for improvement in the cache usage of these algorithms, and consequently also in their running times, especially on parallel machines. Though several systems for auto-generating serial [46, 81, 86] and parallel [69, 82, 101] DP codes from various types of user-provided specifications (e.g., patterns) exist, none of them produce cache-oblivious code with high temporal locality.

3 Cache-oblivious multithreaded dynamic-programming algorithms

This section describes the kind of efficient cache-oblivious multithreaded algorithms we intend to use as the basis of Bellmaniac-generated codes for user-specified dynamic programming recurrences. For simplicity of exposition, we shall concentrate only on the recurrence for the parenthesis problem from Section 2, and ignore many low-level details in our descriptions. We shall report preliminary performance results showing that these types of algorithms significantly outperform their looping counterparts on real machines.

The principal focus of Bellmaniac's compilation technology is multicore processors. These processors are characterized both by parallelism and memory hierarchy, and thus efficient codes for them must be both multithreaded and cache friendly. Bellmaniac will employ provably efficient cache-oblivious multithreaded algorithms automatically generated from DP specifications provided by the user.

```
A(X)
(X points to a of the input matrix c such that the top-left to bottom-right diagonal of X
lies on the top-left to bottom-right diagonal of c.)
1. if X is a small matrix then A_loop(X)
   else
2.    parallel : A(X_11), A(X_22)
3.    B(X_12, X_11, X_22)
```

```
B(X,U,V)
(X, U and V disjoint submatrices (of the same size) of the input matrix c such that X lies
entirely above the top-left to bottom-right diagonal of c, and the top-left to bottom-right
diagonals of U and V lie on that of c. The submatrix U lies to the left of submatrix X,
and both lie on the same rows of c. The submatrix V lies below submatrix X, and both lie
on the same columns of c.)
1. if X is a small matrix then B_loop(X,U,V)
   else
2.    B(X_21, U_22, V_11)
3.    parallel : C(X_11, U_12, X_21), C(X_22, X_21, V_12)
4.    parallel : B(X_11, U_11, V_11), B(X_22, U_22, V_22)
5.    C(X_12, U_12, X_22)
6.    C(X_12, X_11, V_12)
7.    B(X_12, U_11, V_22)
```

```
C(X,U,V)
(X, U and V are disjoint submatrices (of the same size) of the input matrix c each of
which lies entirely above the top-left to bottom-right diagonal of c. Submatrix U lies to
the left of X, and both lie on the same rows of c. Submatrix V lies below X, and both lie
on the same columns of c.)
1. if X is a small matrix then C_loop(X,U,V)
   else
2.    parallel : C(X_11, U_11, V_11), C(X_12, U_11, V_12), C(X_21, U_21, V_11), C(X_22, U_21, V_12)
3.    parallel : C(X_11, U_12, V_21), C(X_12, U_12, V_22), C(X_21, U_22, V_21), C(X_22, U_22, V_22)
```

Figure 4: Parallel recursive divide-and-conquer algorithm for solving Recurrence (1). Initial call is $A(c)$ for an $n \times n$ input matrix $c[1:n, 1:n]$ with $c[i,j] = x_j$ for $1 \le i = j-1 < n$ and $c[i,j] = \infty$ otherwise (i.e., $i \ne j-1$). This function updates all $c[i,j]$ with $1 \le i < j-1 < n$ based on Recurrence (1). We assume that n is a power of 2.

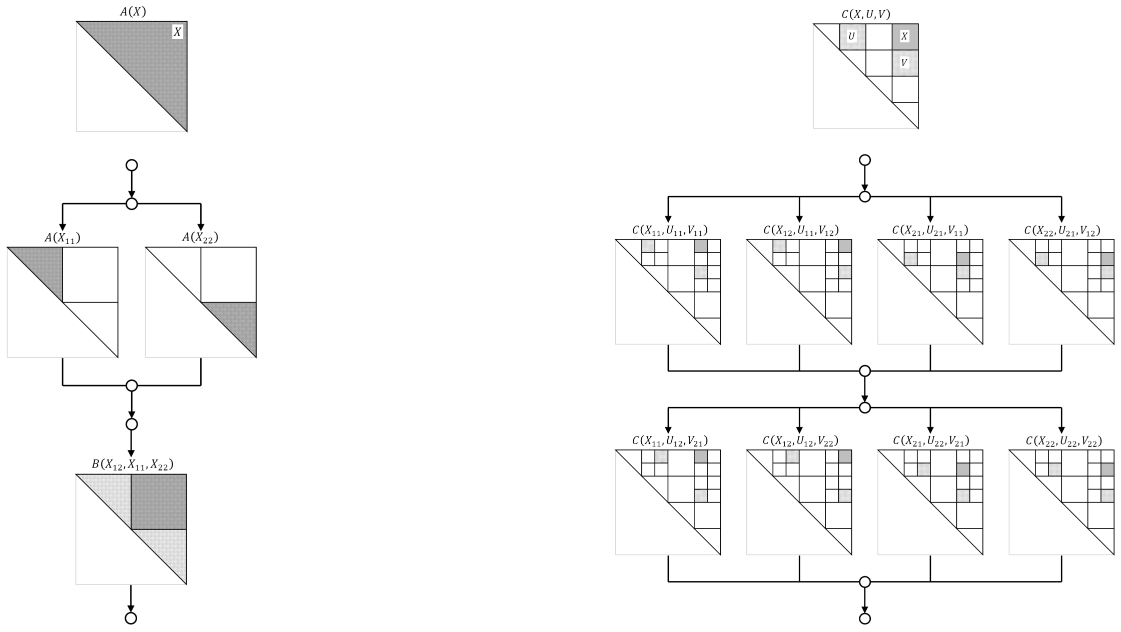
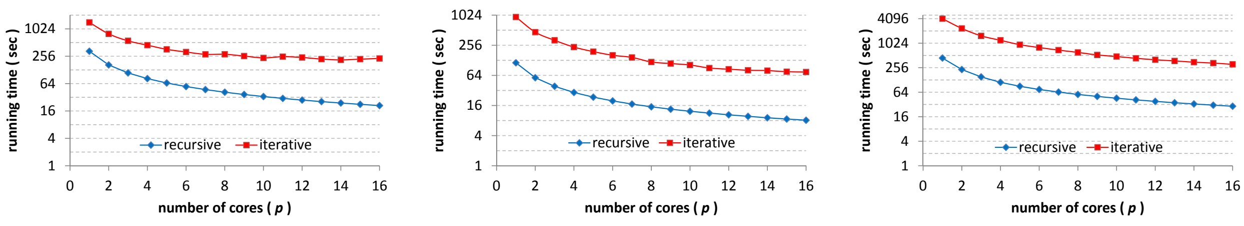

$A(X)$
X
$A(X_{11})$
$A(X_{22})$
$B(X_{12}, X_{11}, X_{22})$
$C(X, U, V)$
U
X
V
$C(X_{11}, U_{11}, V_{11})$
$C(X_{12}, U_{11}, V_{12})$
$C(X_{21}, U_{21}, V_{11})$
$C(X_{22}, U_{21}, V_{12})$
$C(X_{11}, U_{12}, V_{21})$
$C(X_{12}, U_{12}, V_{22})$
$C(X_{21}, U_{22}, V_{21})$
$C(X_{22}, U_{22}, V_{22})$

Figure 5: Visual depiction of the parallel recursive divide-and-conquer algorithm for the parenthesis problem given in Figure 4. Initial call is $A(c)$ for an $n \times n$ input matrix c. We assume that n is a power of 2. Entries in the dark (both dotted and plain) blocks are updated using data from dotted (both light and dark) blocks.

We have argued in Section 2 that though loop-based codes for dynamic programs are often easy to understand and implement, they do not perform well in practice due to poor cache locality. One can easily establish a simple lower bound on the number of cache misses incurred by these algorithms. Let M be the size of the cache, and let B be the size of a cache line. Then if the size of the dynamic programming table is $\gg M$, and the serial running time of the code is T, one can show that LOOPS-PARENTHESIS in Figure 3 incurs $\Omega(T/B)$ cache misses in the ideal-cache model [40].

Efficient cache-oblivious parallel algorithms are known for several DP problems, e.g., see [22] for the pairwise alignment and the gap problems, [25] for Floyd-Warshall's APSP, and [21] for the parenthesis problem. Most of the known cache-oblivious algorithms recursively decompose the DP table into smaller regions so that completely in-cache computations can be performed for sufficiently small subregions. As a result, unlike the looping codes that can exploit only the spatial locality of the data, these algorithms can exploit both spatial and temporal locality. Indeed, one can show that the recursive divide-and-conquer algorithm for the parenthesis problem incurs only $O\left(T/(B\sqrt{M})\right)$ misses.

Figure 4 shows pseudocode for a cache-oblivious multithreaded algorithm for the parenthesis problem (Recurrence (1)), which is the type of parallel cache-efficient algorithm the Bellmaniac compiler aims to generate. Figure 5 gives a clearer visual depiction of the algorithm. For simplicity of exposition, we describe the algorithm only for a square $n \times n$ DP table, where n is a power of 2. The algorithm updates the table through recursive divide-and-conquer of the current 2D grid X into quadrants: X_{11} (top-left), X_{12} (top-right), X_{21} (bottom-left), and X_{22} (bottom-right). Even for this simpler case, the algorithm is considerably more complex than the looping code given in Figure 3 of Section 2. The number and type of recursive functions, their input parameters, order and types of functions called recursively by each of them, and parallelism all vary significantly based on the DP recurrence being evaluated. In order to reduce the overhead of recursion, and to take advantage of vectorization, prefetching, and the processor pipeline, each function stops



(a) Parenthesis problem (Recurrence (1)) **(b)** Floyd-Warshall's APSP [25] **(c)** Gap problem [22]

Figure 6: Comparison of running times of the parallel recursive divide-and-conquer algorithm and the parallel iterative algorithm for (a) the parenthesis problem (Recurrence (1)), (b) Floyd-Warshall's APSP [25], and (c) the Gap problem [22] on a $2^{13} \times 2^{13}$ matrix, as the number of processing cores vary. All algorithms were implemented in Intel Cilk Plus and run on a machine with two 8-core Intel Xeon E5 (Sandy Bridge) processors.

once the input size drops below some threshold and uses a loop-based code for solving the instance. This threshold is often determined empirically, which is a topic we consider in more detail in Section 8. Finally, observe that the type and level of optimization one can apply on the subthreshold looping code of a recursive function can be completely different for each function. For example, while implementation of A_{loop} is basically the same as that of LOOPS-PARENTHESIS in Figure 3, C_{loop} is much more amenable to optimizations (e.g., vectorization and parallelization) than A_{loop}. To see why, observe that in C_{loop} reads and writes are performed on disjoint matrices, and thus eliminating all read/write ordering constraints encountered in A_{loop} and LOOPS-PARENTHESIS. While B_{loop} is not as "optimizable" as C_{loop}, it still can be optimized better than A_{loop}. Assuming that these three loop-based base-case functions are called on matrices of at most a constant size, one can show that for an input $n \times n$ matrix, C_{loop} is called $\Theta\left(n^3\right)$ times over the execution of the entire algorithm, while B_{loop} and A_{loop} are called $\Theta\left(n^2\right)$ times and $\Theta\left(n\right)$ times, respectively. Thus the recursive decomposition exposes many optimization opportunities over the traditional looping code.

The recursive divide-and-conquer algorithms easily outperform loop-based codes on large data sets. For example, we compared the performance of the parallel loop-based and the parallel divide-and-conquer codes for the parenthesis problem, Floyd-Warshall's APSP [25] and the gap problem [25] on 8192×8192 matrices. We compiled using the Intel C++ version 13.1.0 compiler with Intel Cilk Plus [55] and ran on a 16-core (two 8-core) Intel Xeon E5 (Sandy Bridge) machine with a private 32-KB L1-data-cache, a private 256-KB L2-cache, and a shared 20-MB L3-cache. We varied the number of available processing cores from 1 to 16, and in each case the divide-and-conquer code ran significantly faster than the parallel looping code. The speedup factor varied from 4 to 10 for the parenthesis problem, was around 8 for Floyd-Warshall's APSP, and ranged from 9 to 10 for the gap problem. Figure 6 shows the results.

Unfortunately, the performance advantage of the cache-oblivious divide-and-conquer algorithms is inaccessible to most programmers, because the code is simply too hard to write, debug, and tune. Moreover, dependency relationships in DP problems can vary widely, and a slight change in such relationships can change the algorithm significantly. The Bellmaniac compiler, which is the focus of our proposed research, will automatically generate, tune, and compile these complex but efficient algorithms through a simple linguistic interface.

Research Question 1 *How can efficient cache-oblivious multithreaded algorithms for solving general DP problems over general rectangular tables be designed automatically? How should DP's with irregular access patterns, such as the Viterbi algorithm [106], be implemented?*

4 The Bellmania language

We propose to design the ***Bellmania language***, a simple functional programming language that will enable users to concisely define a DP algorithm. For example, consider the parenthesis problem described in Section 2. In Bellmania, this problem might be defined as follows:

```
input n;
input x(i) requires i in [0, n)
input w(i,j,k) requires i, j, k in [0, n)

output C(i,j)
  requires i in [0, n) and j in [i+1, n) :
   case i == j − 1:
     x(j)
   else:
     reduce(min, [C(i,k) + C(k,j) + w(i,k,j) for k in [i+1, j)])
```

Since functions in the Bellmania input language are meant to represent tables, they are limited to only taking integers as parameters. For every integer parameter, the function must declare its allowed range, where the bounds of each range can be constants or linear expression of other parameters. Some functions in a Bellmania program are labeled as **input** functions; these correspond to the tables and scalars that will be fed as input to the generated program. Functions labeled as **output** correspond to the tables that the user expects as output from the program. Finally, temporary functions—defined with the keyword **def**—are used for convenience; the synthesizer may decide to generate a table to implement these functions, but it need not.

For example, suppose that the user only cares about the value of function C at the point (N−1, M−1). Then the user can declare C as temporary and declare an output function **output** OUT() = C(N−1, M−1). The Bellmaniac compiler will then generate an implementation that produces as output a single scalar instead of a table. Section 7 shows how the system can exploit information about what the user expects as output to asymptotically reduce the amount of storage required by a DP algorithm.

All output and temporary functions in Bellmania consist of a set of nonoverlapping cases. For each case, Bellmania restricts the code to simple expressions potentially involving recursive calls and reductions involving min, max, sum, or boolean operations. More formally, functions in Bellmania are represented with a Functional-AST (F-AST) having the following structure:

$$F\,(p, args) = (p \in L_{pre}) \Rightarrow [(L_1, e_1), \ldots (L_n, e_n)]\ .$$

The argument p is an integer vector that corresponds to the integer inputs of each function. The rest of the arguments are used to represent the input functions in the Bellmania program. The constraint $(p \in L_{pre})$ is a precondition which states that p must satisfy a linear constraint L_{pre} corresponding to the **requires** constraints in Bellmania. The cases that make up each function are represented as a pair (L_i, e_i), where L_i is a linear test, and the expression e_i belongs to the grammar below:

$$\begin{array}{rll} e := & var & \mid\ e_1 \circ e_2 \quad \text{where} \ \circ \ \text{is some operator} \\ & & \mid\ F\,(lv, t_1 \ldots t_n) \quad \text{where} \ t_i = \ e_i \quad \mid \quad \lambda x.e_x \\ & & \mid\ reduce_{op}\,(lv_{low}, lv_{high}, \lambda x.e_x) \\ lv := & var & \mid\ linexp(lv) \end{array}$$

Since the L_i must be mutually exclusive, and their disjunction must be logically equivalent to L_{pre}, any $p \in L_{pre}$ will satisfy exactly one L_i, and the result of evaluating F for those arguments is the result of evaluating the corresponding e_i. The notation $F(lv, t_1 \ldots t_n)$ in the grammar is meant to reflect the fact that the vector parameter to a function must be a linear expression of the input parameters, but the rest of the parameters can be more arbitrary expressions. The language supports the creation of anonymous functions, but only in limited contexts, either as parameters to other functions or as arguments to *reduce*.

The F-AST for the user-provided specification in the running example is shown below:

$$C(p) = (p \in L_{pre}) \Rightarrow [(L_1, x(p_1)), (\bar{L_1}, reduce_{min}(p_0, p_1, R)] ,$$

where $R = \lambda k. C(\langle p_0, k\rangle) + C(\langle k, p_1\rangle) + w(\langle p_0, k, p_1\rangle)$. The 2-dimensional vector $p = \langle p_0, p_1\rangle$ corresponds to the `i,j` parameters of `C` in the Bellmania code. The linear constraints are $L_{pre}(p) \equiv 0 \le p_0 < n \wedge p_0 < p_1 < n$ and $L_1(p) \equiv p_0 = p_1 - 1$. The expression $\bar{L_1}$ is the negation of the linear constraint L_1, which is also allowed in the language. The reduction corresponds to the minimization operation in the original program.

The Base compiler implements a simple transformation that can translate a functional representation like the one above into a loop-based implementation. The main idea is that a function like C above can be easily compiled into a recursive implementation that uses uses a table for memoization to avoid redundant recomputation. Moreover, since the indices to the recursive calls are a linear function of the indices to the original call, one can use classical compiler techniques for affine scheduling [36, 61] to discover a loop-nest that will populate the table in an order that guarantees that no recursive call is ever actually needed.

Research Question 2 *What are the trade-offs between expressiveness and analyzability for Bellmania?*

The current language design is simple, but we have found it to be sufficiently expressive to describe many of the DP problems we have studied. As part of this research we want to study the trade-offs between expressiveness and analyzability for this language. For example, there is one benchmark we have considered, the Viterbi algorithm, which cannot be encoded in the above language, but can be encoded with a relatively simple extension. Our goal in the project will be to arrive at a final design that properly trades-off expressiveness with the ability to produce efficient code for any program in the language.

5 Tactic-based lowering process

The Bellmaniac compiler will use a tactic-driven approach to transform the original representation into a representation that corresponds to a recursive cache-oblivious implementation. The approach is inspired by previous work on deductive synthesis based on theorem proving [15, 94], but is simpler by virtue of being domain specific and by relying on an SMT solver. This translation will be performed by the Tactic Application Engine (TAE), which takes as input a F-AST and a series of tactics describing a transformation on the F-AST and produces a transformed F-AST that is proved equivalent to the original. By calling this module with the right sequence of tactics, we can achieve the transformation into a cache-oblivious implementation. We now illustrate the transformation process using the parenthesis problem from Section 2. We show how the F-AST from the original specification can be transformed into a F-AST that describes the recursive partitioning of the problem illustrated in Figure 4. Specifically, we will focus on the recursive decomposition of the triangular part of the problem A_{par} .

As a first step in the decomposition, we capture the global variables n, w, and x and turn them into parameters to the formula in order to make the rest of the manipulations simpler. This conversion is purely mechanical and can be applied automatically as a preprocessing step:

$$C(p, n, x, w) = \left(p \in L_{pre}^{n}\right) \Rightarrow [(L_1, x(p_1)), (\bar{L_1}, reduce_{min}(p_0, p_1, R_1)]$$
$$\text{where } R_1 = \lambda k. C(\langle p_0, k\rangle, n, x, w) + C(\langle k, p_1\rangle, n, x, w) + w(\langle p_0, k, p_1\rangle, n, x, w)$$

The first insight required for the cache-oblivious implementation is knowing that the original grid must be decomposed into quadrants. Our system includes a tactic called split that performs this transformation given a set of parameters describing the details of the decomposition. The result of split is a new function C_1 defined as follows:

$$C_1\,(p,n,x,w) = \left(p \in L^n_{pre}\right) \Rightarrow [(Q^n_{00}, D_{00}), (Q^n_{11}, D_{11}), (Q^n_{01}, D_{01})]$$

where $Q^n_{00} \equiv p_0 < \frac{n}{2} \wedge p_1 < \frac{n}{2}$, $Q^n_{11} \equiv p_0 \geq \frac{n}{2} \wedge p_1 \geq \frac{n}{2}$ and $Q^n_{01} \equiv p_0 < \frac{n}{2} \wedge p_1 \geq \frac{n}{2}$. The case $Q^n_{10} \equiv p_0 \geq \frac{n}{2} \wedge p_1 < \frac{n}{2}$ is not necessary because $Q^n_{10} \wedge L^n_{pre}$ is unsatisfiable. The functions D_{00} and D_{11} are defined in terms of C as follows.

$$\begin{gathered} D_{00}\,(p,n,x,w) = C\,(p, \tfrac{n}{2}, x, w) \\ D_{11}\,(p,n,x,w) = C\,(p - \langle \tfrac{n}{2}, \tfrac{n}{2} \rangle, \tfrac{n}{2}, \lambda a.x\,(a + \tfrac{n}{2}), \lambda q.w\,(q + \langle \tfrac{n}{2}, \tfrac{n}{2}, \tfrac{n}{2} \rangle)) \end{gathered}$$

For D_{01} the definition is more involved, reflecting the fact that the square case is not simply a recursive version of the original problem:

$$\begin{aligned} D_{01} = (p \in Q^n_{01}) \Rightarrow [(L_1, x(p_1)), \left(\bar{L_1}, reduce_{min}\left(p_0, \frac{n}{2}, R_2\right) \oplus reduce_{min}\left(\frac{n}{2}, p_1, R_3\right)\right)] \\ \text{where } R_2 = \lambda k.D_{00}\,(\langle p_0, k\rangle, n, x, w) + D_{01}\,(\langle k, p_1\rangle, n, x, w) + w\,(\langle p_0, k, p_1\rangle, n, x, w) \\ \text{and } R_3 = \lambda k.D_{01}\,(\langle p_0, k\rangle, n, x, w) + D_{11}\,(\langle k, p_1\rangle, n, x, w) + w\,(\langle p_0, k, p_1\rangle, n, x, w) \end{aligned}$$

Given these definitions, one can prove that C_1 is equivalent to C by simple case analysis. Most cases only require checking the satisfiability of sets of linear equalities and inequalities. The case D_{01} additionally requires proving that decomposing a single reduction into two reductions does not alter the result, which follows directly from the associativity of the reduction operator (*min* in the case of the example, but we refer to it generically as $\oplus$).

As a further refinement, each occurrence of C can be replaced with C_1. The replacement is semantics preserving as long as the system can prove that it does not lead to infinite recursion. In this case, proving this is straightforward, because n is divided by half in each recursive call. As a final refinement step, a tactic clear −lambdas eliminates the lambdas in the definition of D_{11}, which is done by adding extra offset parameters o_x and o_w to C_1 and the D_{pq}. The result is the following definition:

$$C_1\,(p,n,x,o_x,w,o_w) = \left(p \in L^n_{pre}\right) \Rightarrow [\ \begin{array}{c} (Q_{00}, C_1\,(p, \frac{n}{2}, x, o_x, w, o_w)), \\ (Q_{11}, C_1\,(p - \langle \frac{n}{2}, \frac{n}{2} \rangle, \frac{n}{2}, x, o_x + \frac{n}{2}, w, o_w + \langle \frac{n}{2}, \frac{n}{2}, \frac{n}{2} \rangle)), \\ (Q_{01}, D_{01}(p,n,x,o_x,w,o_w)) \end{array}\ ]$$

with

$$\begin{aligned} &D_{01}(p,n,x,o_x,w,o_w) = (p \in Q^n_{01}) \Rightarrow [(L_1, x\,(p_1 + o_x)), (\bar{L_1}, reduc\ e_{min}\,(p_0, \tfrac{n}{2}, R_2) \oplus reduce\,(\tfrac{n}{2}, p_1, R_3))] \\ &R_2 = \lambda k.C_1\,(\langle p_0, k\rangle, \tfrac{n}{2}, x, o_x, w, o_w) + D_{01}\,(\langle k, p_1\rangle, n, x, o_x, w, o_w) + w\,(\langle p_0, k, p_1\rangle + o_w, n, x, w) \\ &R_3 = \lambda k.D_{01}\,(\langle p_0, k\rangle, n, x, o_x, w, o_w) + C_1\,(\langle k, p_1\rangle - \langle \tfrac{n}{2}, \tfrac{n}{2} \rangle, \tfrac{n}{2}, x, o_x + \tfrac{n}{2}, w, o_w + \langle \tfrac{n}{2}, \tfrac{n}{2}, \tfrac{n}{2} \rangle) + w\,(\langle p_0, k, p_1\rangle + o_w, n, x, w) \end{aligned}$$

This function corresponds exactly to the initial decomposition in Figure 4 from Section 3. Specifically, function C_1 above will be compiled by the Base compiler into function A_1 in the figure, while function D_{01} will be compiled into a nonrecursive version of B_{par} in the figure. Following a similar sequence of transformations, function D_{01} can also be made cache-oblivious as was illustrated earlier, leading to a fully cache-oblivious algorithm.

The process of compiling C_1 and D_{01} into an imperative implementation is a little more involved than the process described earlier for C. The first important observation is that these functions now take several parameters in addition to p , but the base compiler still need to memoize them into a single table indexed by the input p to the original function. For this example, the Base compiler will solve a set of linear equations to automatically discover that

$$C_1(p,n,x,o_x,w,o_w) = C(\langle p_0+o_x, p_1+o_x\rangle)$$

and similarly for D_{01} . Consequently, the value of $C_1(p,n,x,o_x,w,o_w)$ can be memoized into the output table entry $\langle p_0+o_x, p_1+o_x\rangle$. In general, the mapping from arguments to the transformed function to arguments of the original function will always be an affine transformation thanks to the restrictions in the language.

The second observation is that C_1 and D_{01} are fundamentally different kinds of function. The function D_{01} is structurally similar to the original C; it provides a recursive description of how to compute entries in the table for D_{01} from other entries in the table, and it can be compiled into a nested loop using traditional techniques for affine scheduling. Function C_1, on the other hand must be compiled into a function like those in Figure 4. These functions do not have loops. Instead, they recursively call other functions to do their work. In the case of C_1, we can see that this is such a function because all the recursive calls map to the same table entry; the function simply describes how the original range can be partitioned into three independent domains to be handled by either recursive calls to C_1, or by calls to D_{01}. Determining the dependencies between the three sub-problems requires solving a few additional affine equations which reveal that the computation of D_{01} in quadrant Q_{01} requires the values computed by the recursive calls to C_1 in both quadrants Q_{00} and Q_{11} but that those quadrants do not depend on each other.

We have developed a prototype implementation of the TAE which we have used to generate a cache-oblivious representation of the parenthesis problem and the Floyd-Warshall algorithm. In both cases, the SMT solver Z3 took less than a second to check the validity of all the verification conditions that establish the correctness of the transformation. In addition to the tactics described so far, our system supports the following three tactics: the `refine` tactic replaces function application with one of its refinements; the `unfold` tactic unrolls definition of a function, and the `rewrite` tactic replaces function application $F(x)$ with $G(y)$ where y is derived from x using linear transformations. The proof of validity of such transformation is inductive and it relies on unfolding of the definitions of F and G.

Research Question 3 *Is the system of tactics for generating cache-oblivious computations complete?*

Our current prototype has demonstrated the feasibility of producing provably correct cache-oblivious implementations for nontrivial DP problems, but there are open questions about whether our current set of tactics is sufficiently expressive to derive the optimal implementations for all our target benchmarks or whether the current set of tactics will have to be expanded to support them.

6 Synthesis for tactics

Our current prototype requires a significant degree of expert involvement in order to produce a provably correct implementation. In order to reduce this effort, the Bellmaniac system will introduce automation at two levels. First, in the current prototype, each individual tactic requires considerable information from the user, which makes them hard to write. By using synthesis technology, we can make the tactics smarter and reduce the effort required to write them. Second, the process of generating the sequence of tactics itself can be automated, relieving the programmer from needing to be an expert on cache-oblivious algorithms. The motivation for distinguishing between these two levels of automation is that discovering the details of a given tactic requires fundamentally different technology compared with discovering the sequence of tactics that will lead to a more optimal program.

In terms of the first problem, tactics currently require many details that could be generated automatically. For example in decomposition described earlier, our current implementation of the `split` tactic requires the programmer to specify the details of all the offsets involved in the different recursive calls generated from the transformation in addition to providing the four Q_{ij}^n matrices that define the partitioning. From these details, the Tactic application engine creates the modified code and produces a set of verification conditions whose validity guarantees the correctness of the transformation. The search for parameters for the tactics is very naturally cast as a constraint-based synthesis problem.

Research Question 4 *How can the details of tactics be synthesized automatically? And what is the minimal level of programmer input that can capture the programmer's intent?*

In constraint-based synthesis, the goal is to search for a function f that satisfies a set of constraints that can be solved by a SAT or SMT solver. In the previously mentioned case, all the unknown functions are linear, so the problem reduces to a search for a set of matrices M such that the verification condition is satisfied. Our group has significant expertise in the area of constraint based synthesis [19,20,89,90,97–100]; our Sketch synthesis system [96], is a robust open source infrastructure for constraint based synthesis that has been used to solve problems from a variety of domains ranging from data-structure manipulations [90, 91] to high-performance computations [97] to the automated grading of programming assignments [89]. From our early experiments, we have found that discovering parameters such as the offsets of the `split` tactic are well within the reach of existing synthesis technology.

A second problem with our current set of tactics is that in addition to telling the system how to transform a program, they also have to provide some additional information to explain to the system how to generate the verification conditions. For example, in the case of the parenthesis problem, the transformation required 16 steps of reduction, but an additional 34 commands had to be issued in order to instruct the system on how to generate the verification condition. For the Flowyd-Warshall example, the reduction required 40 steps, but an additional 60 commands had to be specified to generate the verification conditions for the different steps. Most of these steps were required to cope with limitations in the solver's handling of quantifiers. In principle, this problem could be solved by specializing some of the heuristics generally used in theorem proving to work for our problems, but we believe we can do better given our knowledge of the fact that the two functions we are trying to prove equivalent were generated one from the other by a simple transformation.

Compared to searching for details of individual tactics, the search for sequences of tactics requires somewhat different technology for a couple of reasons. Constraint-based synthesis is very good at discovering functions that satisfy a particular correctness constraint; in the case of tactics, constructing a sequence of tactics that is correct is relatively easy; after every step of the transformation, the system can tell us whether the tactic succeeded or failed, and as long as the tactic succeeds, this guarantees that the whole sequence is semantics preserving. The goal is not just to generate a sequence of semantics preserving tactics, however, but to generate a sequence that actually leads to a recursive, cache-oblivious implementation, preferably an efficient one. This goal is difficult to model as a series of constraints, but it is relatively easy to check an implementation to determine whether it has the right structure to produce a recursive cache-oblivious implementation. Moreover, through our recent work in deriving cache-oblivious implementations both by hand and by hand-coding tactics in our prototype, we have observed that very similar sequences of steps tend to be used for very different problems. This suggests that the symbolic search techniques used in constraint-based synthesis are a poor match for this problem, because in addition to requiring a logical formula to describe the goals of the synthesis process, they also treat the search space as a flat space, and are generally unable to take advantage of prior knowledge about the relative likelihood that different solutions will succeed. Instead, a stochastic search procedure such as Metropolis-Hastings as implemented in Church [47] can allow us to encode some of this prior knowledge without requiring a complete symbolic representation of the requirements. Such algorithms have been shown to be effective for some synthesis problems that are difficult to solve symbolically [87].

Research Question 5 *How can statistical synthesis techniques be applied to automate the discovery of tactic sequences?*

7 Complexity analysis, space reduction and traceback

This section outlines how we plan to extend the capabilities of the Bellmaniac compiler beyond simply producing the entire DP table using a multithreaded cache-oblivious algorithm. We will consider three major extensions. First, we will study and implement various space reduction techniques applicable when the entire DP table does not need to be output. Second, we plan to investigate how to find traceback paths through the table cache-efficiently under various scenarios (e.g., batch traceback, space constraints)d extend Bellmaniac to produce efficient traceback extraction algorithms. Finally, we will explore how to incorporate into Bellmaniac the capability to mechanically analyze and output the asymptotic complexities (e.g., parallelism, cache complexity, etc.) of the algorithms it generates.

Space reduction. When one is required to report the values in only a limited number of cells of the DP table, sometimes the entire computation can be performed without allocating space for the whole table. For example, consider the problem of *pairwise sequence alignment with affine gap costs*. This problem takes as input two strings $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$ over a finite alphabet Σ and three cost functions: the gap introduction cost g_i, the gap extension cost g_e, and a mismatch cost $s(a,b)$ for $a,b \in \Sigma$. The minimum cost of aligning X and Y can be found by solving the following recurrences for $i > 0$ and $j > 0$ [48], and computing $\min\{G[m,n], D[m,n], I[m,n]\}$.

$$D[i,j] = \min\{D[i-1,j], G[i-1,j]+g_o\}+g_e, \quad I[i,j] = \min\{I[i,j-1], G[i,j-1]+g_o\}+g_e$$
$$G[i,j] = \min\{D[i,j], I[i,j], G[i-1,j-1]+s(x_i,y_j)\} \tag{2}$$

If one is required to output only the optimal cost of the alignment, i.e., $\min\{G[m,n], D[m,n], I[m,n]\}$, a simple parallel looping code can easily solve the problem using only three arrays of length $m+n$ each. Starting from the top-left corner of the grid, values are computed diagonal by diagonal. All values in the current diagonal are computed in parallel using values from the previous two diagonals. Hence, always retaining the values of the previous two diagonal suffices. In fact, a slightly more careful analysis reveals that only two arrays of length $m+n$ each suffice. Such looping implementations, however, suffer from poor cache performance due to limited temporal locality. For this problem, we have manually derived a cache-oblivious implementation that uses reduced space. Our space-optimized algorithm uses only $\Theta(n)$ space, and has a cache complexity of $O\left(\frac{n^2}{BM}\right)$ where n is the length of both inputs, M is the size of the cache and B is the block transfer size. Our goal for the project will be to automate this process so that users can benefit from such optimizations.

Research Question 6 *From the defining recurrences, how can Bellmaniac mechanically devise an efficient DP computation that operates in reduced space?*

One challenge in devising an implementation that reuses space is that the space-saving implementation no longer fits in our model. This is because our model assumes that every entry in the table is updated only once with its final value, which is why the tables can be treated as functions. In the space-saving implementations, however, the table is reused over and over again to store different values. One way to fit this under our model is to define an affine transformation on the table such that after the transformation, the algorithm derives the values of the table one row at a time, using only the values from the previous row. If this is the case, then the base compiler can identify this situation and create an implementation that only keeps two rows at a time and reuses the space. More formally, suppose that our original goal is to compute the function $D[i,j]$. A space saving implementation is one that computes a table $D'[t,v]$ with the following properties: (a) there exists an affine transformation $T(i,j)$ such that $D[i,j] = D'[T(i,j)]$, (b) entries of $D'[t,v]$ are computed using only entries $D'[t-1,v']$, and (c) the entries that we do care about all map to the same value of t.

We believe we can extend our system to handle these kinds of space reduction techniques by supporting tactics that can infer the kind of affine transformations described above and by extending the base compiler to identify access patterns that allow for space reduction.

Retrieval of traceback path(s). After filling out the DP table, users often seek to extract additional details on the solution(s) by mining the table, and so we propose to extend the Bellmaniac compiler with the ability to generate efficient extraction algorithms. As an example of such DP problems consider Floyd-Warshall's APSP in which one stores additional information in each cell to facilitate the retrieval of an actual shortest path between any two given vertices in addition to the length of the path. For the parenthesis problem, one may want to extract an optimal parenthesization in addition to the cost of such a parenthesization. When the entire DP table is computed and stored, retrieval of such traceback paths is not difficult, and if only a few of them need to be found, the cost of retrieval is often significantly dominated by the cost of computing the original DP table making the efficiency of the retrieval algorithm almost irrelevant. If one needs to find many traceback paths from a single table as in Floyd-Warshall's APSP, however, parallelization and cache optimization of the extraction algorithm becomes important. In any case, when the entire table is available, extraction of traceback paths can be considered a postprocessing step. A more complicated situation arises, however, when one doesn't have enough space to store the entire DP table for postprocessing. For example, the space reduction approach outlined in the preceding paragraphs gives rise to such a scenario. In such cases the retrieval algorithm is no longer straightforward, as it may need to recompute various missing portions of the DP table on-the-fly using the given limited space. Such a traceback path generation algorithm is given in [21, 23, 26] for pairwise sequence alignment. Deriving such an algorithm is nontrivial, but we believe it can be done following the same basic machinery we have outlined so far.

Research Question 7 *How can the Bellmaniac synthesis approach be extended to facilitate efficient traceback in the context of batch extraction and/or space limitations?*

Automating analysis of asymptotic complexity. Since the cache-oblivious algorithms we synthesize are based on a divide-and-conquer strategy involving one or more recursive functions, their asymptotic performance can be analyzed using a set of recurrence relations. We believe it is possible to automate this analysis, allowing Bellmaniac to give the programmer insight about the fundamental properties of her algorithm in addition to generating the algorithm itself.

Research Question 8 *Can we extend Bellmaniac to mechanically analyze and output the various asymptotic complexities of the algorithms it generates?*

To illustrate how these asymptotic bounds are derived by hand, consider the multithreaded cache-oblivious algorithm for the parenthesis problem given in Figure 4. For simplicity, we will assume n to be a power of 2, and the base case functions are called when we reach matrices of size 1×1.

For $f \in \{A, B, C\}$, let $T_f(n)$ be the span of f_{par} on an $n \times n$ matrix. Then $T_f(1) = \Theta(1)$, and for $n > 1$,

$$T_A(n) = T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + \Theta(1)\,, \quad T_B(n) = 3\left(T_B\left(\frac{n}{2}\right) + T_C\left(\frac{n}{2}\right)\right) + \Theta(1)\,, \quad T_C(n) = 2T_C\left(\frac{n}{2}\right) + \Theta(1)\,.$$

Clearly, $T_C(n)$ can be computed directly using the Master theorem [31]. Then we simplify the recurrence for $T_B(n)$ using the solution for $T_C\left(\frac{n}{2}\right)$, and apply the Master theorem on the simplified recurrence. Finally, we simplify the recurrence for $T_A(n)$, and solve it to find $T_A(n) = \Theta\left(n^{\log_2 3}\right)$. Since the algorithm performs $\Theta\left(n^3\right)$ work, its parallelism = work / span = $\Theta\left(n^3/n^{\log_2 3}\right) = \Omega\left(n^{1.41}\right)$.

The cache complexity of the algorithm on a serial machine can be described using similar recurrences. Once we have both the span and the serial cache complexity of the algorithm we can predict its cache performance on a parallel machine with private caches under Cilk's work-stealing scheduler [7]. This complexity will help users better understand the performance characteristics they can expect of their synthesized code.

8 Code generation and tuning

Bellmaniac will generate near-optimal and automatically tuned code for the user's DP problem. Code generation and tuning go hand in hand. The code generator can produce code that is hard to tune or easy to tune, and the ease of tuning affects which strategy the code generator should use. Some design decisions should be bound early and embodied in the code generator, whereas others should be bound later, either autotuning ***online***, where tuning occurs at runtime, or ***offline*** as a separate autotuning step in advance of the runtime execution. This section discusses some of the key design decisions for Bellmaniac and how we plan to investigate the various design decisions.

Bellmaniac's code generator can be viewed as producing two outputs:

- tuning code that creates an optimized plan for a DP computation, and
- runtime code that executes according to the plan.

The autotuning code will be employ a strategy similar to FFTW [37, 38] (originally developed by Matteo Frigo when he was student in our MIT research group and Steve Johnson, our collaborator on the NSF-sponsored Pochoir project). FFTW exhaustively enumerates the search space during the tuning process using DP to avoid timing similar subproblems. FFTW manages to perform its tuning online, because it adopts what we call a ***clearbox*** strategy, where the internal structure of the code is known to the tuning code, as opposed to ***blackbox*** tuning, where the internal structure of the code being tuned is opaque to the tuning code. clearbox tuning is particularly effective for recursive programs, because the work of timing a subproblem that is similar to a previously timed subproblem can be avoided by simply using the earlier timing. The clearbox tuner knows that the two subproblems are equivalent from a timing point of view, whereas a blackbox tuner would have no such knowledge. The big difference between Bellmaniac's clearbox approach and FFTW is that FFTW's code generator knows that the generated code is performing an FFT, whereas Bellmaniac's code generator does not know what DP problem the code is performing. As a consequence, Bellmaniac's search space is both larger and more complex.

Research Question 9 *Can clearbox tuning reduce the tuning overhead and improve search coverage compared to blackbox tuning? Can clearbox tuning make online tuning practical for DP problems? Is there a way of combining online and offline tuning?*

Various empirical autotuning frameworks have been developed in domain-specific libraries. ATLAS [110] and OSKI [107], which are dense and sparse linear algebra libraries, where they mathematically model the search space. Others such as PATUS [27], PetaBricks [4], Sepya [60], SPIRAL [74] and Active harmony [105] use evolutionary and learning methods to prune the search space. Many other heuristic and exhaustive autotuning frameworks have been developed in recent literature [1,3,9,10,14,43,49,54,59,70,73,76,80,102,112].

```
$A(X)$
1. if $plan[``A(X)"] \neq$ NIL, then return
2. $t_{loop}$ = TIME: {A(X)}
3. $t_{rec}$ = TIME: {$A(X_{11})$
          $A(X_{22})$
          $B(X_{12}, X_{11}, X_{22})$}
4. if $t_{loop} < t_{rec}$
5. then $plan[``A(X)"].status$ = LOOP
          $plan[``A(X)"].cost = t_{loop}$
6. else $plan[``A(X)"].status$ = REC
          $plan[``A(X)"].cost = t_{rec}$
```

(a)

```
$A(X)$
1. if $plan[``A(X)"].status$ = LOOP then $A_{loop}(X)$
   else
2.    parallel : $A(X_{11}), A(X_{22})$
3.    $B(X_{12}, X_{11}, X_{22})$
```

(b)

Figure 7: The output of the Bellmaniac code generator for the parenthesis problem from Figure 4. Only the code for A is shown. **(a)** A tuning program for generating an optimized plan. **(b)** The runtime code.

To illustrate how code generation and tuning will operate in Bellmaniac, consider the parenthesis problem, and suppose that the only design parameter being optimized is the coarsening of recursion. Figure 7 shows pseudocode for both the tuning code and the runtime code. The tuning code in Figure 7(a) recursively divides the problem to generate a ***plan***. For the generated subproblems during the recursion, the times[2] for both looping and recursion are computed, and the less costly decision is stored in the plan. The generated plan from the tuning stage is used by the runtime code shown in Figure 7(b).

The tuning code performs exhaustive enumeration of the subproblems, timing both the divide-and-conquer version and the looping version for each subproblem size. We analyzed the runtime code in Section 7 and showed that the looping code and the pure divide-and-conquer code both run in $\Theta(n^3)$, where n the problem size. Although the recurrence describing the work of the divide-and-conquer runtime code is somewhat more complicated due to the mutual recursion among A, B, and C, intuition will be enhanced by considering it to be $T(n) = 8T(n/2) + O(1)$.

How much more expensive is the tuning code? The tuning code performs both the divide-and-conquer and looping. Consequently, its performance can be modeled intuitively by the recurrence $T(n) = 8T(n/2) + O(n^3)$, which has solution $T(n) = \Theta(n^3 \lg n)$. That is, there is only a logarithmic overhead from the runtime code to the tuning code. Compared with blackbox techniques, which often must run the tuning code hundreds of times, the overhead of our clearbox approach is only about 20–30, even for large problem sizes.

In fact, we expect to do even better for this kind of optimization. The *plan* data structure saves the decisions and costs made by the tuning code for the optimized runtime code. But *plan* can be used to "memoize" [31] the code for the benefit of the tuning code itself. In particular, two subproblems that are computed in the same way can be considered as the same subproblem from the point of view of timing if the only difference (aside from the actual table data) is that one is a translation of the other in time. That

[2]The pseudocode shows which code should be instrumented for time, but it does not show how. Instead, it simply uses the nomenclature **TIME** to indicate that a piece of code should be timed. In fact, timing can be a tricky business to do correctly with minimum overhead, but our team has the experience to do it accurately and robustly.

is, they have the same shape and size. For example, in Figure 7(a), if X_{11} and X_{22} are the same size, as they would be if all problem sizes are a power of 2, then only one needs to be calculated. The effect of this optimization to the tuning code is that the recurrence for tuning becomes $T(n) = T(n/2) + \Theta(n^3)$, which has solution $T(n) = \Theta(n^3)$, just like the runtime code! In other words, the tuning code may be insignificantly slower than a single execution of the runtime code. The *plan* data structure can be made to recognize when two subproblems are equivalent, and then only time the first of them, returning the computed time for any equivalent subproblem, not just for any "equal" subproblem. Thus, the search space of subproblems can really be much smaller than the total number of subproblems.

What happens when we relax the constraint that the problem size is a power of 2? As it turns out, the tuning time remains stable. A good analogy is to ask for an arbitrary number n, if we divide it into $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ and recursively divide those two numbers, how many different numbers do we obtain by the time we get down to $n = 1$? If n is a power of 2, the answer is $\lg n$. If n is an arbitrary number, it turns out that the answer is only at most $2\lg n$. Dealing with subproblem sizes that are not powers of 2 poses no real difficulty, although in more than one dimension, the problem becomes more complex.

	1D	*2D*
Autotuning	21,378	572.2
Predicted serial	6,219	14.7
Autotuned serial	6,186	14.2
Hand-tuned serial	6,644	16.0
Autotuned parallel	1,047	2.5
Hand-tuned parallel	815	2.0

Figure 8: Performance results (in seconds) for the proposed autotuning strategy operating on 1*D* and 2*D* stencil codes for the heat equation. The codes ran on a 12-core (two 6-core) Intel Xeon E5 (Sandy Bridge) machine with a private 384-KB L1-data-cache, a private 1.5-MB L2-cache, and a shared 15-MB L3-cache.

To get a handle on the empirical implications of this tuning strategy, we tested it using our Pochoir stencil compiler. As with Bellmaniac, the generated code from Pochoir is recursive, although considerably simpler, since it only deals with nearest-neighbor dependencies. Figure 8 shows how the tuning strategy performs on two stencil codes: a 1D heat equation with a space-time size of $100\text{K} \times 10\text{M}$, and a 2D heat equation with space-time size $2\text{K} \times 2\text{K} \times 100\text{K}$. The predicted runtime was computed by accumulating looping or recursion costs of the generated plan from the tuning phase. The hand-tuned code was the original hand-optimized Pochoir code.

The figure shows that autotuning takes less than 4 times the runtime of the optimized code. The predicted serial runtime is within a few percent of the actual autotuned serial runtime, which outperforms the hand-tuned code by about 20%.

The results for the parallel code are decent, but not as good. An investigation into the causes revealed that the autotuned parallel runtime suffered from poor parallelism. We modified the tuning code to favor the divide-and-conquer code over the base case serial looping code if the divide-and-conquer code, which has parallelism, was only a few percent slower. This change improved the quality of the parallel code with only a small overhead to the serial code, but the autotuned parallel code still did not outperform the serial code.

The structure of Bellmaniac's tuning code admits several other optimizations that we plan to explore. For two-dimensional problems, for example, the tuning code could be given a choice as to whether to partition in x, partition in y, or invoke the base case looping code. It turns out that this additional choice does not increase the size of the search space unduly. Other tuning opportunities that we shall investigate include padding arrays for low-associativity caches, dealing with data alignment, and determining exactly how and where to parallelize.

Because Bellmaniac's clearbox strategy yields fast tuning code, we shall investigate options for running it online, as well as offline. In particular, an online tuner could exploit *a priori* knowledge that the leaves of the recursion cannot be smaller than a known threshold and that there is similarly no point in considering base cases far up in the tree. Consequently, the tuner may actually be able to operate in a fraction of the time it takes to actually run the code, which would make it suitable for online tuning.

9 Beyond multicore

Programmers should be able to write a single Bellmania source program and rely on the Bellmaniac compiler to target multiple platforms. Although the principal focus of this research project targets standard multicore systems, we propose to explore some research ideas on how to map to other architecture platforms.

Clusters. Distributed-memory clusters require that programs operate on a collection processing nodes, each of which has an independent address space. Instead of communicating through shared memory, clusters employ message-passing using, for example, MPI [32, 72, 79] or Erlang [5, 6]. Since the nodes of today's clusters are multicore processors, obtaining best-possible performance involves parallelism at two levels: intranode (shared memory) and internode (distributed memory), creating a level of complexity that most programmers are not willing or able to address.

We propose to investigate a strategy by which the multicore Bellmaniac implementation can be employed for intranode parallelism, while extending it with message passing for internode communication. Conceptually, the message-passing Bellmaniac could generate multicore Bellmania code with embedded MPI or Erlang library calls, which would then be run through the multicore Bellmania compiler. The central research question is how Bellmaniac can minimize the overhead of internode data movement while retaining intranode cache-efficiency.

GPU's and vector units. While the emergence of GPGPU languages like CUDA [77] and OpenCL [64] have substantially eased the job of GPU programming, writing high-performance GPU codes still requires significant effort. Moreover, the vector units in most multicore processors are underutilized. We believe Bellmania can produce specialized recursion base cases to exploit these architectural features.

Exploiting GPU's and vector units poses several difficult research challenges. First, to keep the Bellmania specification simple and universal across the range of targeted platforms, it will be necessary to produce vectorized code from a scalar description of the kernel. While this may not be hard for simple dynamic programs, such as the longest common subsequence, more complex kernels, such as LBM, may resist efficient vectorization. Second, vectorized loops consume significantly more memory bandwidth than divide-and-conquer recursion. Thus, although the code may go faster using vector technologies on one processing core, memory bottlenecks may cause it to run slower on multiple cores. An intriguing possibility is to adaptively enable vectorization depending on the number of cores.

MIC Coprocessors. Recently, Intel launched its highly threaded ***Many Integrated Core*** (MIC) coprocessors [58] with wide SIMD capability for fast data-parallel computations. An Intel MIC coprocessor is an x86-based SMP-on-a-chip with many cores. Each core supports multiple threads and features 512-bit SIMD registers. The current version, Intel Xeon Phi, has 61 cores and supports 4 threads per core. Each core has a private L1-cache as well as its own L2-cache. The L2-caches are connected through an interprocessor network. Each processor has fast access to its own L2-cache, and a bit slower access to all other L2-caches. Caches are coherent across the entire coprocessor.

Intel MIC has much better support for recursion than GPU's. If the DP table is small enough to fit into MIC's device memory, optimizing the base case functions to take advantage of MIC's SIMD capability should lead to significant improvement in the performance of the cache-oblivious DP code. We also plan to investigate how to run the code cache-efficiently on MIC when the entire table doesn't fit into its memory. One approach would be to generate efficient code for paging between device and host memories in addition to the cache-oblivious DP code. Whenever a base-case function accesses a cell from the DP table, the entire page containing the cell is brought into device memory, and because of the recursive divide-and-conquer nature of the cache-oblivious algorithm, the page will often contain data to be accessed by the core(s) in near future. The paging policy and the page size should to be chosen carefully so that a page being accessed by one core does not need to evict a page currently being accessed by another core in order to make space for itself. LRU should work nicely as the paging policy, and the page size can be computed easily based on the number of cores, the size of the device memory, and the maximum number of different blocks accessed by a function. Bellmaniac should also align the DP table carefully in order to avoid false sharing, which can become an even bigger issue when multiple MIC coprocessors are running in parallel.

FPGA systems. Field-Programmable Gate Arrays (FPGA's), which can provide a massive level of hardware parallelism with fast on-chip memories, seem eminently suitable for dynamic-programming computations. Indeed, companies like Maxeler [71] offer FPGA dynamic-programming solvers. The main research issue is that FPGA's are much more difficult to program efficiently than pure-software approaches. As with GPU's and vector units, a key challenge will be whether the Bellmania language can be kept simple while fully exploiting the hardware.

10 Benchmarking

We propose to assemble a benchmark suite of DP problems drawn from a wide range of application areas in order to track improvements during the engineering of Bellmaniac. The benchmark suite will include representative applications from a variety of disciplines including bioinformatics (e.g., RNA secondary structure prediction [2,26], pairwise sequence alignment [48], structural alignment [78]), games and sports (e.g., various board games [93], football [84], the Duckworth-Lewis method for resetting the target in interrupted cricket matches [33]), the Viterbi algorithm used in many applications based on the Hidden Markov Model [106], relational database management (e.g., access path selection [88]), text processing (e.g., Knuth's word-warping algorithm for paragraph formatting in the TEX typesetting system [65], the Wagner-Fischer algorithm for computing the Levenshtein distance between two strings of characters [108]), parsing (e.g., CYK algorithm for parsing context-free languages [29, 62, 111], Earley's algorithm for chart parsing [35]), economics and finance (e.g., bond refinancing [83], modeling Markov Decision Processes [85]), and agriculture (e.g., predicting biomass of a crop [63]).

Research Question 10 *Can Bellmaniac consistently deliver performance over such a variety of DP problems?*

The benchmark suite, together with the synthesized implementations for a variety of platforms, will be publicly released and will constitute an important deliverable for our project. We believe that for many problems in our suite, the Bellmaniac-generated implementations will become the *de facto* standard on many systems, in the same way that FFTW has become the *de facto* standard for solving FFT's on many systems.

11 Intellectual merit

Traditional loop-based implementations of dynamic programming may suffer by well over an order of magnitude in performance compared to the optimal algorithms when run on modern multicore architectures. Exploiting the parallel processing capability of multicore architectures and their steep cache hierarchies, however, involves complicated programming. By attaining high performance without sacrificing programming simplicity, the Bellmania project will broadly advance knowledge across diverse fields. Bellmania will enable a dramatic increase in both computational and programmer efficiency. Even within computer science, the generalization of cache-oblivious algorithms to DP and the other innovations of this proposal, as well as their crystallization into a computer language, represents a significant advance in the understanding of DP problems. Previous work on cache optimization and load balancing of DP algorithms, aside from straightforward loop-based techniques, has been dominated by simplified examples. This proposal represents an original as well as a practical direction of research, with a creative new approach in the form of an expert system to generate efficient multicore code.

The project will also explore a new approach to develop domain specific synthesis systems by relying on general technology for constraint solving and constraint-based synthesis. We believe that by demonstrating how this technology can simplify the process of developing domain-specific programming systems, the project will give further impetus to the emerging interest in synthesis within the programming-language and formal-methods communities.

12 Broad impact

This research will allow users of dynamic programming to exploit the benefits of cache-efficient parallel implementations without having to incur the enormous effort to develop them. This access will come both in the form of a domain-specific language that will be easy to learn and use by practitioners in the field, as well as in the form of prebuilt libraries that can be linked and accessed directly from an application. These algorithms can allow for order-of-magnitude performance improvements, which could be transformative for a range of disciplines, from computational biology to operations research.

As we have done in the past with our research projects, including Cilk, FFTW, and Sketch, we will make our software freely available in open-source form. It is our experience that doing this can help maximize the impact of a project, allowing it to become not only an essential tool for practitioners, but a research platform that can be used by other researchers to further advance the state of the art.

Finally, the project will include 5 graduate students at MIT and 2 at Stony Brook. The project will engage undergraduate students as part of MIT's Undergraduate Research Opportunities Program. The proposing faculty have a history of involvement in outreach to underrepresented minorities, including serving as faculty supervisor for the Undergraduate Society for Women in Mathematics at MIT, serving as director of the Undergraduate Practices Opportunity Program, and by supervising numerous student researchers who are women or underrepresented minorities (four in the past two years).

13 Curriculum Development Activities

Our team has a strong track record of incorporating research developments into the classroom. For example, the Sketch system is used as part of a class on Foundations of Program Analysis (6.820) to introduce students to synthesis. The AutoGrader system [89], which our group developed to automatically derive feedback for programming assignments is currently being used to help teaching assistants to provide better feedback to students in the course Introduction to Computer Science and Programming (6.00). Cilk technology is currently used to teach students about the principles of parallel programming as part of the Performance Engineering course at MIT (6.172) and the Parallel Programming (CSE613) and Advanced Algorithms (CSE638) courses at Stony Brook. We plan to incorporate Bellmaniac into all three of these courses. One particular feature of our system that we believe will be helpful from a pedagogical standpoint will be the ability to automatically generate diagrams like the one in Figure 5, which explain the recursive decomposition of the original problem into a cache-oblivious one, as well as the ability to describe to students the algorithmic complexity of the synthesized algorithms. These capabilities will give students a better understanding of how cache-oblivious algorithms work. Materials will additionally be made freely available to the public via the MIT OpenCourseWare initiative (`http://ocw.mit.edu`).

14 Results under prior NSF support

Prof. Solar-Lezama has three ongoing grants from NSF. The project 1116362, "SHF: Small: Human-Centered Software Synthesis" ($404,779, 09/11-08/14) is exploring new approaches for interaction between a synthesizer and a user. This project was born from an earlier short-term EAGER grant 1049406, "EAGER: Human-Centered Software Synthesis" ($90,000, 08/10-07/11). **Intellectual Merit:** This project pioneered the idea of using constraint based-synthesis to enable *storyboard programming*, allowing a user synthesize data-structure manipulations from high-level diagrams [90,91]. As part of this work we have also developed some basic synthesis technology [92]. **Broader Impact:** The systems developed as part of this project are fully open source and available for download from the PI's website.

Prof. Solar-Lezama is also being supported by project 1139056 , "Collaborative Research: Expeditions in Computer Augmented Program Engineering (ExCAPE): Harnessing Synthesis for Software Design" ($500,000, 04/12-04/17) is a large multiinstitution project that aims to develop new programming models based on software synthesis and apply them in a variety of domains. **Intellectual Merit:** As part of this project, we have developed a new synthesis-based approach to optimize database-backed programs [20] and a new way of automatically providing feedback for programming assignments by that identifies minimal corrections for a student's program to work correctly [89]. **Broader Impact:** The autograding work is already being used to support TAs teaching 6.00.

Prof. Solar-Lezama is also supported by project 1161775, "SHF: Medium: Collaborative Research: Marrying program analysis and numerical search" ($600,000, 09/12-08/16) is exploring the use of numerical search techniques for synthesis and verification of programs. **Intellectual Merit:** As part of this project, we have shown that by symbolically *smoothing* a program, we can successfully use numerical optimization techniques like Nelder-Mead to find optimal parameters for it [18]. More recently, we have shown that the same idea can be used to find provably correct parameters for probabilistic programs [28]. **Broader Impact:** As part of this project, we have also developed the Euler system, which is available in open source form the PI's website.

Prof. Chowdhury and Prof. Leiserson are currently supported by NSF Grant CCF-1162196, "SHF: AF: Medium: Collaborative Research: The Pochoir Stencil Compiler," ($983,017, 04/12- 03/2014) a two-year collaborative grant with Steven Johnson (Applied Math) of MIT. This research project investigates how high-efficiency cache-oblivious codes for stencil computations can be generated from simple-to-write linguistic specifications. **Intellectual Merit:** Thus far, this award has provided funding in part for six research papers [13, 44, 66, 68, 103, 104]. **Broader Impact:** In addition, the Pochoir compiler itself is available as open-source software.

Prof. Leiserson has recently received support for NSF Grant CCF-1314547, "SHF: AF: Large: Collaborative Research: Parallelism without Concurrency," ($1,000,000, 07/13-06/17) a four-year collaborative grant with Guy E. Blelloch of Carnegie Mellon and Jeremy Fineman of Georgetown. This research project is studying three key strategies for dealing with concurrency: encapsulating concurrency so that it is hidden by layered abstractions at appropriate abstraction levels, avoiding concurrency by restructuring programs to employ deterministic approaches, and managing concurrency when it is impractical to either encapsulate or avoid concurrency completely. **Intellectual Merit:** The project has already produced three papers [51–53] in the final phases of preparation.

15 Collaboration plan

The research project will be led by Prof. Armando Solar-Lezama of MIT CSAIL. The project will involve Prof. Charles E. Leiserson of MIT CSAIL, and Prof. Rezaul Chowdhury of Stony Brook University Department of Computer Science, who is also affiliated with Stony Brook Institute for Advanced Computational Sciences. Prof. Yuan Tang from Fudan University, Shanghai, P.R.China will contribute to the project as an unpaid collaborator and seek his own funds elsewhere. Dr. Maryam Mehri Dehnavi will work on the project as a postdoctoral researcher. The team has been meeting regularly using phone and video conferencing as well as in person, and will continue to do so to ensure the success of the project.

The project comprises six subtasks:

1. ***Basic*** — Using the existing Bellmaniac prototype as a model, producing a robust implementation of the interactive component of the system.
2. ***Tactic Synthesis Engine*** — Explore the combination of constraint-based synthesis and stochastic search to infer sequences of tactics that lead to efficient cache-oblivious algorithms.
3. ***Algorithms*** — Designing provably optimal parallel cache-efficient algorithms and data structures for DP problems.
4. ***Code Generation and Autotuning*** — Performance-engineering the DP algorithms to produce efficient code. Developing clearbox and hybrid autotuning frameworks to optimize tuning parameters in the code.
5. ***Platform*** — Tailoring the system to support platforms beyond today's mainstream multicore architecture. The targeting platforms under consideration include MIC coprocessors, GPGPU's, distributed-memory clusters, and FPGA's. Investigating effective and efficient algorithms and data structures on different platforms.
6. ***Benchmarking*** — Benchmarking typical applications. Assembling a comprehensive benchmark suite of dynamic programs.

As the research team is small and its members familiar with working with one another, management of tasks can largely be done informally. In addition to leading the overall research effort, Prof. Solar-Lezama will lead the design effort for Tasks 1 and 2. Prof. Chowdhury, and Prof. Tang will jointly lead the development of Tasks 3 and 6. Prof. Leiserson will mentor the postdoctoral researcher and jointly lead Task 4 with Prof. Tang. Dr. Mehri Dehnavi will engage in Tasks 3–4 under the supervision of Prof. Leiserson. All PI's and collaborators will work on Task 5 and otherwise contribute to all tasks, including helping designing linguistics, developing foundational algorithmic theory, and implementations. We plan to make a major release of Bellmaniac annually, as well as minor and beta releases throughout the year. The software will be freely available on the World Wide Web under an open-source license.

References

[1] Felix V. Agakov, Edwin V. Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael F. P. O Boyle, John Thomson, Marc Toussaint, and Christopher K. I. Williams. Using machine learning to focus iterative optimization. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pages 295–305. IEEE Computer Society, 2006.

[2] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudo-knots. *Discrete Applied Mathematics*, 104:45–62, 2000.

[3] L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. *SIGPLAN Not.*, 39(7):231–239, June 2004.

[4] Jason Ansel and Cy Chan. Petabricks: Building adaptable and more efficient programs for the multi-core era. *Crossroads, The ACM Magazine for Students (XRDS)*, 17(1):32–37, Sep 2010.

[5] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98*, pages 119–129, June 1998.

[8] V. Bafna and N. Edwards. On de novo interpretation of tandem mass spectra for peptide identification. In *Proceedings of the 7th Annual International Conference on Research in Computational Molecular Biology*, pages 9–18, Berlin, Germany, 2003.

[9] Prasanna Balaprakash, Stefan M Wild, and Paul D Hovland. An experimental study of global and local search algorithms in empirical performance tuning. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 261–269. Springer, 2013.

[10] Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. Towards making autotuning mainstream. *International Journal of High Performance Computing Applications*, 27(4):379–393, 2013.

[11] Gerald Baumgartner, David E. Bernholdt, Daniel Cociorva, Robert J. Harrison, So Hirata, Chi-Chung Lam, Marcel Nooijen, Russell M. Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *SC*, pages 1–10, 2002.

[12] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[13] Michael Bender, Rezaul Chowdhury, Pramod Ganapathi, Sam McCauley, and Yuan Tang. The Range 1 Query (R1Q) problem. Under review, 2013.

[14] James Bergstra, Nicolas Pinto, and David Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9. IEEE, 2012.

[15] Mark Bickford, Christoph Kreitz, Robbert Van Renesse, and Robert Constable. An experiment in formal design using meta-properties. 2001.

[16] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[17] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *JPDC*, 37(1):55–69, August 1996.

[18] Swarat Chaudhuri and Armando Solar-Lezama. Euler: A system for numerical optimization of programs. In *CAV*, pages 732–737, 2012.

[19] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In *CIKM*, pages 1732–1736, 2012.

[20] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[21] R. Chowdhury and V. Ramachandran. Cache-efficient Dynamic Programming Algorithms for Multicores. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 207–216, 2008.

[22] Rezaul Chowdhury. *Cache-efficient Algorithms and Data Structures: Theory and Experimental Evaluation*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, 2007.

[23] Rezaul Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, 2006.

[24] Rezaul Chowdhury and Vijaya Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 71–80, 2007.

[25] Rezaul Chowdhury and Vijaya Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4):878–919, 2010. A preliminary version appeared as [24].

[26] Rezaul A. Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *TCBB*, 7(3):495–510, July-September 2010.

[27] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[28] Martin Clochard, Swarat Chaudhuri, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *POPL 2014 (to appear)*, 2014.

[29] John Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes, 1970.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[32] Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors. *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*. Springer, 2011.

[33] F. C. Duckworth and A. J. Lewis. A fair method for resetting the target in interrupted one-day cricket matches. *The Journal of the Operational Research Society*, 49(3):220–227, 1998.

[34] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[35] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[36] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[37] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[38] Matteo Frigo. A fast Fourier transform compiler. *ACM SIGPLAN Notices*, 34(5):169–180, May 1999.

[39] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, pages 79–90, Calgary, Canada, August 2009. ACM.

[40] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, New York, NY, October 17–19 1999.

[41] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[42] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[43] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[44] Pramod Ganapathi, Rezaul Chowdhury, and Yuan Tang. The R1Q problem. In *22nd Annual Fall Workshop on Computational Geometry*, 2012.

[45] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[46] Robert Giegerich and Georg Sauthoff. Yield grammar analysis in the Bellman's GAP compiler. In *Proceedings of the 11th Workshop on Language Descriptions, Tools and Applications*, page 7. ACM, 2011.

[47] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *In UAI*, pages 220–229, 2008.

[48] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[49] Max Grossman and Sameer Kulkarni. Applying genetic algorithms to tune heterogeneous platform configurations. In *The International Conference on Parallel, Distributed, and rid Computing (PDGC)*, 2012.

[50] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, New York, 1997.

[51] Will Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Chromatic scheduling of dynamic data-graph computations. 2013.

[52] Will Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. 2013.

[53] Will Hasenplaugh and Julian Shun. Tighter bounds for parallel greedy maximal independent set and matching. Unpublished manuscript, 2013.

[54] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *SIGARCH Comput. Archit. News*, 39(1):199–212, March 2011.

[55] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[56] Intel Corporation. Tera-scale Computing-A Parallel Path to the Future. `http://software.intel.com/en-us/articles/tera-scale-computing-a-parallel-path-to-the-future/`, 2011. Retrieved 9/19/2011.

[57] Intel Corporation. The Intel Many Integrated Core Architecture. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2011. Retrieved 9/19/2011.

[58] Intel Corporation. The Intel Many Integrated Core Architecture. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2011.

[59] Herbert Jordan, Peter Thoman, Juan J Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12. IEEE, 2012.

[60] Shoaib Ashraf Kamil. *Productive High Performance Parallel Programming with Auto-tuned Domain-Specific Embedded Languages*. PhD thesis, University of California, Berkeley, 2012.

[61] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[62] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

[63] John O. S. Kennedy. Applications of dynamic programming to agriculture, forestry and fisheries: Review and prognosis. *Review of Marketing and Agricultural Economics*, 49(03), 1981.

[64] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1*, June 2011. Retrieved 9/19/2011.

[65] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

[66] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly pipeline parallelism. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 140–151, July 2013.

[67] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[68] Charles E. Leiserson, Tao Benjamin Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*. ACM, 2012. to appear.

[69] Weiguo Liu and Bertil Schmidt. A generic parallel pattern-based system for bioinformatics. In *Euro-Par 2004 Parallel Processing*, pages 989–996. Springer, 2004.

[70] Thibaut Lutz, Christian Fensch, and Murray Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59, 2013.

[71] Maxeler. The MaxGenFD Whitepaper. `http://www.maxeler.com/content/briefings/MaxelerWhitePaperMaxGenFD.pdf`.

[72] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009.

[73] Jeffrey Morlan, Shoaib Kamil, and Armando Fox. Auto-tuning the matrix powers kernel with sejits. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 391–403. Springer, 2013.

[74] José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Appplications*, 18(1):21–45, February 2004.

[75] Leonardo De Moura and Nikolaj Björner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[76] Svetlana Nogina, Kristof Unterweger, and Tobias Weinzierl. Autotuning of adaptive mesh refinement pde solvers on shared memory architectures. In *Parallel Processing and Applied Mathematics*, pages 671–680. Springer, 2012.

[77] Nvidia. *CUDA C Programming Guide v 3.2*, October 2010.

[78] Christine A. Orengo and William R. Taylor. SSAP: Sequential structure alignment program for protein structure comparison. In Russell F. Doolittle, editor, *Computer Methods for Macromolecular Sequence Analysis*, volume 266 of *Methods in Enzymology*, pages 617–635. Academic Press, 1996.

[79] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997.

[80] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 119–129, Washington, DC, USA, 2011. IEEE Computer Society.

[81] Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 83–98. ACM, 2011.

[82] Raphael Reitzig. Automated parallelisation of dynamic programming recursions. Master's thesis, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2012.

[83] Alexander A. Robichek, Edwin J. Elton, and Martin J. Gruber. Dynamic programming applications in finance. *The Journal of Finance*, 26(2):473–506, 1971.

[84] David Romer. It's fourth down and what does the bellman equation say? a dynamic programming analysis of football strategy. Technical report, National Bureau of Economic Research, 2002.

[85] John Rust. Numerical dynamic programming in economics. *Handbook of computational economics*, 1:619–729, 1996.

[86] Georg Sauthoff, Stefan Janssen, and Robert Giegerich. Bellman's GAP: a declarative language for dynamic programming. In *Proceedings of the 13th international ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, pages 29–40. ACM, 2011.

[87] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM.

[88] P. Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[89] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[90] Rishabh Singh and Armando Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, pages 289–299, 2011.

[91] Rishabh Singh and Armando Solar-Lezama. Spt: Storyboard programming tool. In *CAV*, pages 738–743, 2012.

[92] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. Modular synthesis of sketches using models. In *VMCAI 2014 (to appear)*, 2014.

[93] David K. Smith. Dynamic programming and board games: A survey. *European Journal of Operational Research*, 176(3):1299–1318, 2007.

[94] Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[95] M. Sniedovich. *Dynamic Programming*. The Marcel Dekker, Inc., New York, NY, 1992.

[96] Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[97] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *PLDI*, 2007.

[98] Armando Solar-Lezama, Chris Jones, Gilad Arnold, and Rastislav Bodík. Sketching concurrent datastructures. In *PLDI 08*, 2008.

[99] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[100] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[101] Shanjiang Tang, Ce Yu, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zhen Xu, and Huabei Wu. EasyPDP: An efficient parallel dynamic programming runtime system for computational biology. *IEEE Transactions on Parallel and Distributed Systems*, 23(5):862–872, 2012.

[102] Wai Teng Tang, Wen Jun Tan, Ratna Krishnamoorthy, Yi Wen Wong, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. Optimizing and auto-tuning iterative stencil loops for gpus with the in-plane method. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 452–462. IEEE, 2013.

[103] Yuan Tang, Rezaul A. Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir stencil compiler. In *SPAA*, San Jose, CA, USA, 2011.

[104] Yuan Tang, Rezaul A. Chowdhury, Chi-Keung Luk, and Charles E. Leiserson. Coding stencil computation using the Pochoir stencil-specification language. In *HotPar'11*, Berkeley, CA, USA, May 2011.

[105] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *In Proceedings from the Conference on High Performance Networking and Computing*, pages 1–11, 2003.

[106] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[107] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005.

[108] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[109] M. S. Waterman. *Introduction to Computational Biology*. Chapman & Hall, London, UK, 1995.

[110] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *Conference on High Performance Networking and Computing. Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27, 1998.

[111] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967.

[112] Yongpeng Zhang and Frank Mueller. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 155–164. ACM, 2012.