2015 IEEE 29th International Parallel and Distributed Processing Symposium

High-performance Energy-efficient Recursive Dynamic Programming with Matrix-multiplication-like Flexible Kernels

Jesmin Jahan Tithi Pramod Ganapathi Aakrati Talati Sonal Aggarwal Rezaul Chowdhury
Department of Computer Science, Stony Brook University, New York, 11790-4400, USA
E-mail:{jtithi, pganapathi, atalati, soaggarwal, rezaul}@cs.stonybrook.edu

***Abstract*—Dynamic Programming (DP) problems arise in a wide range of application areas spanning from logistics to computational biology. In this paper, we show how to obtain high-performing parallel implementations for a class of DP problems by reducing them to highly optimizable flexible kernels through *cache-oblivious recursive divide-and-conquer* (CORDAC). We implement parallel CORDAC algorithms for four non-trivial DP problems, namely the parenthesization problem, Floyd-Warshall's all-pairs shortest path (FW-APSP), sequence alignment with general gap penalty (gap problem) and protein accordion folding. To the best of our knowledge our algorithms for protein accordion folding and the gap problem are novel. All four algorithms have asymptotically optimal cache performance, and all but FW-APSP have asymptotically more parallelism than their looping counterparts.**

We show that the base cases of our CORDAC algorithms are predominantly matrix-multiplication-like (MM-like) flexible kernels that expose many optimization opportunities not offered by traditional looping DP codes. As a result, one can obtain highly efficient DP implementations by optimizing those flexible kernels only. Our implementations achieve $5-150\times$ speedup over their standard loop based DP counterparts while consuming order-of-magnitude less energy on modern multicore machines with $16-32$ cores. We also compare our implementations with parallel tiled codes generated by existing polyhedral compilers: Polly, PoCC and PLuTo, and show that our implementations run significantly faster. Finally, we present results on manycores (Intel Xeon Phi) and clusters of multicores obtained using simple extensions for SIMD and shared-distributed-shared-memory architectures, respectively, demonstrating the versatility of our approach. Our optimization approach is highly systematic and suitable for automation.

***Keywords*-cache-oblivious, recursive, divide-and-conquer, flexible kernel, polyhedral compiler, dynamic programming**

I. Introduction

Dynamic programming (DP) [2], [19], [25] is a popular algorithm design technique for finding optimal solutions to a problem by combining optimal solutions to many overlapping subproblems. DP is used in a wide variety of application areas [20] including operations research, compilers, sports and games, economics, finance, and agriculture. DP is extensively used in computational biology, such as in protein-homology search, gene-structure prediction, motif search, phylogeny analysis, analysis of repetitive genomic elements, RNA secondary-structure prediction, and interpretation of mass spectrometry data [1], [9], [17], [31].

This work is partially supported by NSF grant OCI-1053575 and uses Extreme Science and Engineering Discovery Environment (XSEDE). Rezaul Chowdhury and Pramod Ganapathi are supported in part by NSF grants CCF-1162196 and CCF-1439084.

Traditional Loop-based DP Algorithms. Dynamic programs are traditionally implemented using simple loop-based algorithms. For example, Fig. 2 shows looping code snippets for four DP problems. Such loop-based algorithms are straightforward to implement, sometimes have good *spatial locality*[1], and benefit from hardware prefetchers. But looping codes suffer in performance due to poor *temporal cache locality*[2]. Low temporal locality leads to increased pressure on memory bandwidth which increases with the number of active cores. More cache misses also results in more energy consumption. Therefore, there is a significant room for improvement in the cache usage of these algorithms, and consequently also in their running times and energy usage, especially on parallel machines.

LOOP-PARENTHESIS(c, n) {*Inflexible Code*}
(Input is an $n \times n$ matrix $c[1:n, 1:n]$ with $c[i,j] = v_j$ for $1 \le i = j-1 < n$ and $c[i,j] = \infty$ otherwise (i.e., $i \neq j-1$).)
1) **for** $i \leftarrow n-2$ **downto** 1 **do**
2) **for** $j \leftarrow i+2$ **to** n **do**
3) **for** $k \leftarrow i+1$ **to** j **do**
4) $c[i,j] \leftarrow \min\{\, c[i,j],\ c[i,k] + c[k,j] + w(i,k,j) \,\}$

LOOP-MM(d, a, b, n) {*Flexible Code*}
(Inputs are disjoint $n \times n$ matrices a, b and d. This function computes the product of a and b in d.)
1) **for** $i \leftarrow 1$ **to** n **do**
2) **for** $j \leftarrow 1$ **to** n **do**
3) **for** $k \leftarrow 1$ **to** n **do**
4) $d[i,j] \leftarrow d[i,j] + a[i,k] \times b[k,j]$

Figure 1. Inflexible looping code for the parenthesization problem vs. the flexible looping code for matrix multiplication.

Flexible vs. Inflexible Kernels. Iterative DP implementations are often inflexible in the sense that the loops and the data in the DP table cannot be suitably reordered in order to optimize for better spatial locality, parallelization and/or vectorization. Such inflexibility arises from the strict read-write ordering of the DP table cells imposed by the code that reads from and writes to the same table. For example, irrespective of whether matrix c is stored in row-major order or column-major order, the given i-j-k ordering of the loops in LOOP-PARENTHESIS of Fig. 1 incurs $\Theta\left(n^3\right)$ cache misses under the *ideal cache model* [13]. Observe that i-k-j ordering of the loops will incur only $\mathcal{O}\left(n^3/B + n^2\right)$ cache misses, where B is the cache line size, and will also lead to better stride lengths for efficient vectorization. However, the i-k-j ordering will make the algorithm incorrect. Compare this DP implementation with the iterative matrix multiplication (MM) code LOOP-MM shown in Fig. 1. Both code snippets look similar except that LOOP-MM reads from and writes to two disjoint matrices making all 6 orderings of the loops valid, and thus making the code much easier to optimize.

[1]Spatial locality — whenever a cache block is brought into the cache, it contains as much useful data as possible.

[2]Temporal locality — whenever a cache block is brought into the cache, as much useful work as possible is performed on it before removing the block from the cache.

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.107

Though the given i-j-k ordering will incur $\Theta\left(n^3\right)$ cache misses, one can easily reduce that to $\mathcal{O}\left(n^3/B + n^2\right)$ either by reordering the loops to i-k-j or by storing matrix b in column-major order and a in row-major order. Also since no cell in d depends on any other cell of d, one can correctly update all its n^2 cells in parallel by parallelizing both i- and j-loops. One cannot extract that much parallelism from LOOP-PARENTHESIS because almost every cell in c depends on many other cells of c, and thus imposes an order in which the cells must be updated. We refer to kernels, such as LOOP-MM, that perform reads and writes on disjoint matrices as *flexible* kernels.

DP using Recursive Divide-and-Conquer. DP algorithms based on the cache-oblivious recursive divide-and-conquer (CORDAC) technique can often overcome many limitations of their iterative counterparts. Because of their recursive nature such algorithms are known to achieve excellent (and often optimal) temporal locality. Efficient implementations of these recursive algorithms use iterative kernels when the problem size becomes reasonably small [32]. In this paper, we show that for several DP problems the recursive decomposition reduces the original inflexible looping code into recursive functions and iterative kernels that are predominantly flexible (i.e., reading from and writing to disjoint submatrices). Such flexibility does not only lead to *highly optimizable codes*, but often to algorithms with *asymptotically better parallelism* than the original looping code. The size of the iterative kernel can often be kept independent of the cache parameters[3] without paying a significant performance penalty, and thus keeping the algorithms both cache-efficient and cache-oblivious[4] [13].

Tiled Loops vs. Recursive Divide-and-Conquer. Though one can achieve optimal cache performance by tiling the looping code, unlike CORDAC tiling remains a cache-aware approach. Moreover, simply tiling a parallel loop nest does not improve its asymptotic parallelism. While tiling can produce flexible iterative kernels too, CORDAC's strength lies in its ability to utilize flexible recursive functions. High level of parallelism achieved by these functions often leads to a CORDAC-based DP algorithm with asymptotically more parallelism than its parallel looping counterpart.

Our Contributions. We consider four DP problems. Among them, the *parenthesization problem* (also known as the *parenthesis problem* [15]) arises in sequence analysis and in RNA secondary structure prediction [21], [26] as well as in optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polygon triangulation. The *gap problem* occurs in sequence alignment with gaps, *Floyd-Warshall's all-pairs shortest path (FW-APSP)* has applications in computing transitive closure and phylogeny analysis [22], and the *protein accordion folding (PAF) problem* has its roots in protein structure prediction.

Our major contributions are as follows.

▷ **[*Reduction to Flexible Computations for Better Parallelism and Optimizations*]** We show that for our benchmark problems the CORDAC approach basically reduces the computations to flexible recursive functions and highly optimizable flexible kernels which asymptotically dominate the total computation cost. The flexible recursive functions often lead to asymptotic improvements in parallelism over the corresponding parallel looping codes (with/without tiling).

▷ **[*Novel CORDAC Algorithms*]** We present the first efficient parallel CORDAC algorithms for protein accordion folding and sequence alignment with general gap penalty. We analyze their theoretical time and cache complexities.

▷ **[*Optimizations and Experimental Analyses on Shared-Memory Machines*]** We describe general optimization strategies for our CORDAC implementations that can lead up to $5 - 150\times$ speedup w.r.t. to the optimized parallel looping implementations on multicores with $16 - 32$ cores, and up to $180\times$ speedup on Intel Xeon Phi manycores. Our optimization approach is systematic enough for automation and incorporation into a compiler.

▷ **[*Comparison with Codes Generated by Polyhedral Compilers*]** We show that our CORDAC implementations run significantly faster than parallel tiled DP implementations generated by PLuTo [3], PoCC [23] and Polly [16].

▷ **[*Energy, Power and Runtime Tradeoff*]** We show that CORDAC implementations consume significantly less energy than looping implementations. They can afford to slowdown (by using fewer cores) to reduce power consumption while still running faster than loops. We explore this tradeoff between power consumption and running time.

▷ **[*Extension to Heterogeneous Platforms*]** We show that CORDAC algorithms achieve almost linear scalability on multicores for large enough inputs, and reasonable scalability when run on a cluster of multicore machines under hierarchical dynamic load-balancing without any change in the basic CORDAC structure. Moreover, CORDAC also performs very well on manycores Xeon Phi as well as on hybrid CPU + Xeon Phi platforms.

II. ALGORITHMS

In this section we present standard parallel loop-based and CORDAC algorithms for the protein accordion folding, FW-APSP, gap, and parenthesization problems. For simplicity of exposition we assume $n = 2^t$ for some integer $t \geq 0$ for all problems, where $n \times n$ is the size of the DP table. Table I lists span and cache complexity of all the four CORDAC algorithms and their iterative counterparts. We show the analysis for the parenthesization problem and omit the rest since they can be derived similarly.

[3] since cache sizes on modern machines are almost never less than 8KB

[4] Cache-oblivious algorithms — algorithms that do not use the knowledge of cache parameters in the algorithm description.

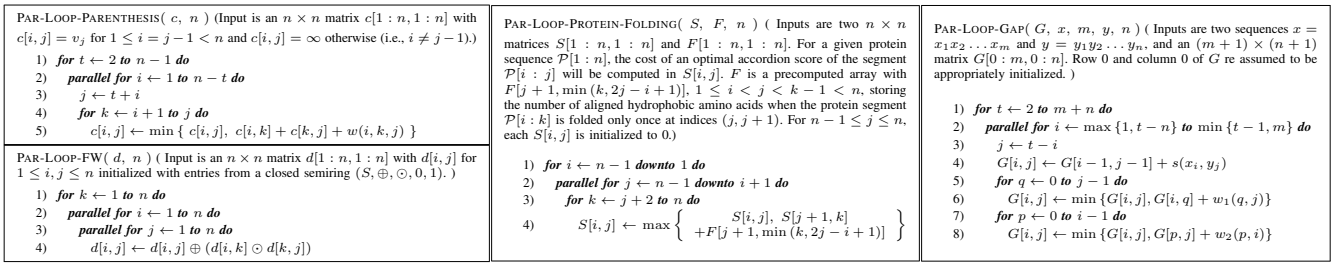

Figure 2. Loop-based parallel codes for the parenthesis problem (PAR-LOOP-PARENTHESIS), Floyd-Warshall's APSP (PAR-LOOP-FW), the gap problem (PAR-LOOP-GAP) and protein accordion folding (PAR-LOOP-PROTEIN-FOLDING). In addition to the parallel ***for*** loops already shown, the serial ***for*** loops in lines 5 and 7 of LOOP-GAP and in line 4 of LOOP-PARENTHESIS and PAR-LOOP-PROTEIN-FOLDING can be parallelized using reducers [12].

A. Parenthesization Problem

The *parenthesization problem* [15] asks for the minimum parenthesization cost of a given sequence $X = x_1 x_2 \cdots x_n$. Let $c[i, j]$ denote the minimum cost of parenthesizing $x_i \cdots x_j$. For $1 \leq i < n$, each $c[i, i + 1]$ is assumed to be already known ($= v_{i+1}$), and for $1 \leq i \leq n$ each $c[i, i]$ is assumed to be ∞.

A function $w(\cdot, \cdot, \cdot)$ is given such that for $1 \leq i < k \leq n$, $w(i, k, j)$ returns the cost of combining parenthesizations of $x_i \cdots x_k$ and $x_k \cdots x_j$ which can be computed without additional cache/memory accesses. Then for $0 \leq i < j-1 < n$, $c[i, j]$ is computed as follows.

$$c[i, j] = \min_{i \leq k \leq j} \big\{ (c[i, k] + c[k, j]) + w(i, k, j) \big\}$$

The optimal parenthesizing cost $c[1, n]$ for the entire sequence can be found using the parallel looping code PAR-LOOP-PARENTHESIS given in Fig. 2. Observe that the parallel looping code is different from the serial code LOOP-PARENTHESIS shown in Fig. 1 as none of the loops in that serial code can be directly parallelized because of the dependencies in the order of cell computation. PAR-LOOP-PARENTHESIS computes cells diagonal by diagonal starting from $c[1, 1]$ and ending at $c[n, n]$. All cells on the same diagonal can now be computed in parallel.

A parallel CORDAC algorithm for solving the parenthesization problem is shown in Fig. 3 which is a special case of the algorithm we proposed in [7]. This algorithm uses three recursive functions: A_{par}, B_{par} and C_{par}.

Function $A_{par}(X)$ updates the upper triangular part of square matrix X (initially $X \equiv c[1 : n, 1 : n]$) using data from X, i.e., each $c[i, j]$ in X is updated using only the $\langle c[i, k], c[k, j] \rangle$ pairs that lie completely inside X. The recurrence for $c[i, j]$ suggests that X_{11} and X_{22} are self-dependent like X, and hence can be updated recursively by A_{par}. Then we need to update the cells in X_{12}, and each such update of a cell $c[i, j] \in X_{12}$ must use $\langle c[i, k], c[k, j] \rangle$ pairs such that either $c[i, k] \in X_{11} \wedge c[k, j] \in X_{12}$ or $c[i, k] \in X_{12} \wedge c[k, j] \in X_{22}$. This is done by calling function $B_{par}(X, U, V)$ with $X = X_{12}$, $U = X_{11}$ and $V = X_{22}$, which updates a square matrix X ($= X_{12}$) using data from itself and upper triangular matrices U (to the left of X) and V (below X).

In function $B_{par}(X, U, V)$, clearly, X_{12} depends only on data in upper triangular submatrices U_{22} and V_{11}, and hence can be updated recursively. Observe that each update of a cell $c[i, j] \in X_{11}$ must use either (i) $c[i, k] \in U_{12} \wedge c[k, j] \in X_{21}$, or (ii) $c[i, k] \in U_{11} \wedge c[k, j] \in X_{11}$, or (iii) $c[i, k] \in X_{11} \wedge c[k, j] \in V_{11}$. Case (i) is handled by calling function $C_{par}(X_{11}, U_{12}, X_{21})$ which we describe later, and the remaining two cases are handled by calling $B_{par}(X_{11}, U_{11}, V_{11})$ recursively. Similar argument holds for updating X_{22}. Each update of a cell $c[i, j] \in X_{12}$ must use either (i) $c[i, k] \in U_{12} \wedge c[k, j] \in X_{22}$, or (ii) $c[i, k] \in X_{11} \wedge c[k, j] \in V_{12}$, or (iii) $c[i, k] \in U_{11} \wedge c[k, j] \in X_{12}$, or (iv) $c[i, k] \in U_{12} \wedge c[k, j] \in V_{22}$. The first two cases can be solved by calling $C_{par}(X_{12}, U_{12}, X_{22})$ and $C_{par}(X_{12}, X_{11}, V_{12})$ recursively, and the last two cases are solved by calling $B_{par}(X_{12}, U_{11}, V_{22})$ recursively.

Function $C_{par}(X, U, V)$ updates square X using data from squares U and V, i.e., $c[i, j] \in X$ is updated using $\langle c[i, k], c[k, j] \rangle$ pairs such that $c[i, k]$ lies inside U and $c[k, j]$ lies inside V, and hence, C_{par} is MM-like, and has the same form as recursive square matrix-multiplication algorithm.

Table I shows that the kernel function of C_{par} is asymptotically dominating (i.e., invoked asymptotically more times than the other two kernel functions) and is also the only flexible kernel among the three.

Problem	#invocations of iterative kernels ($m = n/b$)				Work (T_1)	Iterative		Recursive	
	A_{f-loop}	B_{f-loop}	C_{f-loop}	D_{f-loop}		Serial Cache Complexity (Q_1)	Span (T_∞)	Serial Cache Complexity (Q_1)	Span (T_∞)
Parenthesization (*f = par*)	$\Theta(m)$	$\Theta(m^2)$	$\Theta(m^3)$	-	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$
Protein Folding (*f = fold*)	$\Theta(m)$	$\Theta(m^2)$	$\Theta(m^2)$	$\Theta(m^3)$	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n \log n)$
Gap Problem (*f = gap*)	$\Theta(m)$	$\Theta(m^3)$	$\Theta(m^3)$	-	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$
FW-APSP (*f = FW*)	$\Theta(m)$	$\Theta(m^2)$	$\Theta(m^2)$	$\Theta(m^3)$	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n \log n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n \log^2 n)$

Table I

COMPLEXITIES OF THE ITERATIVE AND CORDAC ALGORITHMS, AND THE NUMBER OF INVOCATIONS OF ITERATIVE KERNELS BY CORDAC ALGORITHMS WHEN RUN ON AN INPUT MATRIX OF SIZE $n \times n$ WITH BASE-CASE SIZE $\leq b \times b$. FLEXIBLE KERNELS ARE SHOWN ON YELLOW BACKGROUND AND ASYMPTOTICALLY DOMINATING KERNELS ARE SHOWN IN RED. HERE, M = SIZE OF THE CACHE AND B = CACHE LINE SIZE. RUNTIME ON p PROCESSING ELEMENTS IS $T_p = \mathcal{O}(T_1/p + T_\infty)$, CACHE COMPLEXITY IS $Q_p = \mathcal{O}(Q_1 + p(M/B)T_\infty)$ (W.H.P.) WHEN RUN UNDER CILK'S WORK-STEALING SCHEDULER.

Serial Cache Complexity. For $f \in \{A, B, C\}$, let $Q_f(n)$ denote the cache complexity of f_{par} on a matrix

of size $n \times n$ when run on a serial machine. Then $Q_f(n) = \mathcal{O}\left(n + n^2/B\right)$ if $n^2 \leq \gamma_f M$ for some suitable constant $\gamma_f \in (0, 1]$. Otherwise, $Q_A(n) = 2Q_A(n/2) + Q_B(n/2)$, $Q_B(n) = 4(Q_B(n/2) + Q_C(n/2))$, and $Q_C(n) = 8Q_C(n/2)$. Solving, $Q_A(n) = \mathcal{O}\left(n + n^2/B + n^3/M + n^3/\left(B\sqrt{M}\right)\right)$.

Span. For $f \in \{A, B, C\}$, let $T_f(n)$ denote the span of f_{par} on a matrix of size $n \times n$. Then $T_f(n) = \Theta(1)$ if $n = 1$. Otherwise, $T_A(n) = T_A(n/2) + T_B(n/2) + \Theta(1)$, $T_B(n) = 3(T_B(n/2) + T_C(n/2)) + \Theta(1)$, and $T_C(n) = 2T_C(n/2) + \Theta(1)$. Solving, $T_A(n) = \mathcal{O}\left(n^{\log_2 3}\right)$.

B. Protein Accordion Folding

In the *protein accordion folding problem (PAF)* we assume that a protein is folded into a 2D square lattice in such a way that the number of pairs of hydrophobic amino acids that are next to each other in the grid (vertically or horizontally) without being next to each other in the protein sequence is maximized. We assume that the fold is always an *accordion fold* where the sequence first goes straight down, then straight up, then again straight down, and so on. Beta sheets in proteins often fold this way.

The recurrence below computes the optimal *accordion score* (see [18]) $S[i, j]$ of the protein segment $\mathcal{P}[i : j]$ which assumes $S[i, j] = 0$ for $j \geq n - 1$. The optimal score for the entire sequence is given by $\max_{1<j\leq n}\{S[1, j]\}$.

$$S[i, j] = \max_{j+1<k\leq n}\{\text{SOF}(i, j, k) + S[j + 1, k]\}$$

The function $\text{SOF}(i, j, k)$ counts the number of aligned hydrophobic amino acids when the protein segment $P[i : k]$ is folded only once at indices $(j, j + 1)$. Observe that $\text{SOF}(i, j, k) = \text{SOF}(1, j, k)$ if $k \leq 2j - i + 1$, and $\text{SOF}(i, j, k) = \text{SOF}(1, j, 2j - i + 1)$ otherwise. Hence, in $\mathcal{O}\left(n^2\right)$ time one can precompute an array $\text{F}[1 : n, 1 : n]$ such that for all $1 \leq i < j < k - 1 < n$, $\text{SOF}(i, j, k) = \text{F}[j + 1, \min\{k, 2j - i + 1\}]$. Then $S[i, j] = \max_{j+1<k\leq n}\{\text{F}[j + 1, \min\{k, 2j - i + 1\}] + S[j + 1, k]\}$ for $j < n-1$.

In Fig. 3 we present a CORDAC algorithm for computing $S[1 : n, 1 : n]$ based on the recurrence above. The algorithm uses four recursive functions A_{fold}, B_{fold}, C_{fold} and D_{fold}. Function $A_{fold}(\ X\ )$ updates the upper triangular part of X (which is originally set to $\langle S[1 : n, 1 : n], F[1 : n, 1 : n]\rangle$) using data completely inside that part of X. Function A_{fold} recursively calls itself and functions B_{fold} and C_{fold}. Function $B_{fold}(\ X,\ V\ )$ updates a square X using data from X and from the upper triangular part of another square V that lies below X in the original input $n \times n$ square. This function recursively calls itself and function D_{fold}. Function $C_{fold}(\ X,\ U\ )$ updates the upper triangular part of X using data from X and a square U that lies to the right of X. Function C_{fold} recursively calls itself and function D_{fold}. Finally, function $D_{fold}(\ X,\ V\ )$ updates a square X using data from another square V that lies below and to the right of X. This function recursively calls only itself and is flexible. Table I shows that though the iterative kernels $B_{fold-loop}$ and $D_{fold-loop}$ are both flexible (no read-write constraint), only $D_{fold-loop}$ is asymptotically dominating.

C. Sequence Alignment with General Gap Penalty

The problem of *sequence alignment with general gap penalty (gap problem)* [14], [15], [31] is a generalization of the edit distance problem that arises in molecular biology, geology, and speech recognition. When transforming a string $X = x_1x_2 \ldots x_m$ into another string $Y = y_1y_2 \ldots y_n$, a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. An affine gap penalty function is predominantly used in bioinformatics, for which $O(n^2)$ algorithms are available [31], [6]. However, in many applications the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In this paper, to handle any general case, two new cost functions w and w' are defined, where $w(p, q)$ $(0 \leq p < q \leq m)$ is the cost of deleting $x_{p+1} \ldots x_q$ from X, and $w'(p, q)$ $(0 \leq p < q \leq n)$ is the cost of inserting $y_{p+1} \ldots y_q$ into X. The substitution function $S(x_i, y_j)$ is the same as that of the standard edit distance problem. Let $G[i, j]$ denote the minimum cost of transforming $X_i = x_1x_2 \ldots x_i$ into $Y_j = y_1y_2 \ldots y_j$ (where $0 \leq i \leq m$ and $0 \leq j \leq n$) under this general setting. Then $G[0, 0] = 0$, $G[0, j] = w(0, j)$ for $1 \leq j \leq n$, and $G[i, 0] = w'(0, i)$ for $1 \leq i \leq m$. Otherwise,

$$G[i, j] = \min\left\{\begin{array}{c} G[i-1, j-1] + S(x_i,\ y_j), \\ \min_{0\leq q<j}\{\ G[i, q] + w(q, j)\ \}, \\ \min_{0\leq p<i}\{\ G[p, j] + w'(p, i)\ \} \end{array}\right\}.$$

In the rest of the paper we will assume $m = n$ for simplicity.

The parallel iterative DP algorithm PAR-LOOP-GAP shown in Fig. 2 solves the gap problem. In Fig. 3 we present a parallel CORDAC algorithm for solving the problem which uses three recursive functions A_{gap}, B_{gap} and C_{gap}. The iterative kernels invoked by B_{gap} and C_{gap} are asymptotically dominating and flexible (Table I). Table I shows the span and cache complexity of these algorithms.

D. All Pairs Shortest Path Problem

Consider a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$, and each edge (v_i, v_j) is labeled by an element $l(v_i, v_j)$ of some closed semiring $(S, \oplus, \odot, 0, 1)$. For $i, j \in [1, n]$ and $k \in [0, n]$, let $d^{(k)}[i, j]$ denote the cost of the smallest cost path from v_i to v_j with no intermediate vertex higher than v_k. Then $d^{(n)}[i, j]$ is the cost of the shortest path from v_i to v_j. The following recurrence computes all $d^{(k)}[i, j]$ for $k > 0$ assuming $d^0[i, i] = 1$ and $d^0[i, j] = l(v_i, v_j)$ for all $i, j \in [1, n]$:

$$d^{(k)}[i, j] = d^{(k-1)}[i, j] \oplus \left(d^{(k-1)}[i, k] \odot d^{(k-1)}[k, j]\right).$$

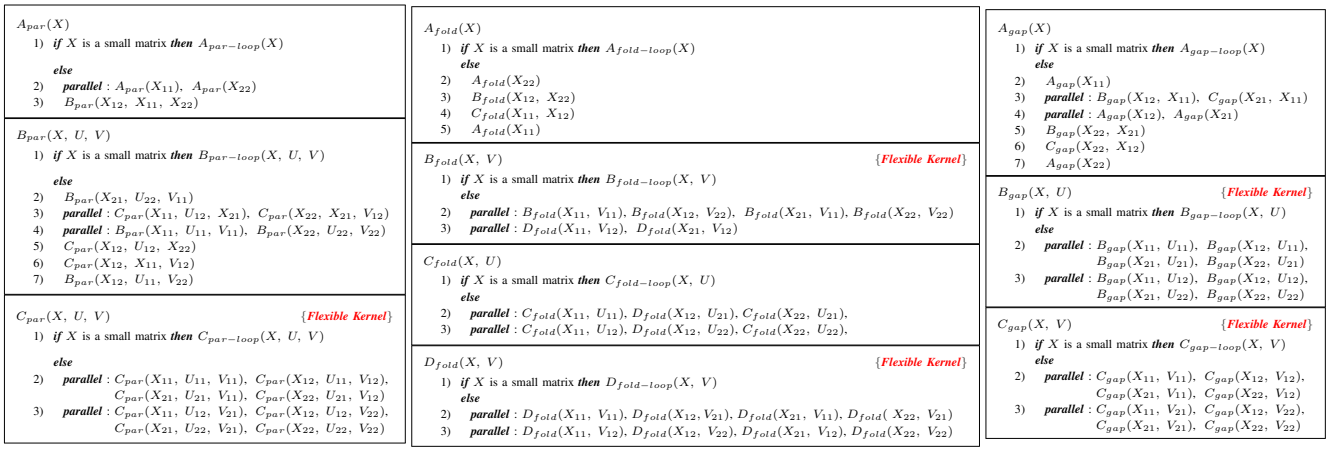

Figure 3. Parallel CORDAC algorithms for solving the parenthesization problem, the protein accordion folding problem, and the gap problem. For simplicity, we assume n to be a power of 2. Initial function calls are as follows. (1) parenthesization problem: $A_{par}(c)$ for an $n \times n$ input matrix c, (2) protein accordion folding: $A_{fold}(\ X\ )$, where $X = \langle S[1:n, 1:n], F[1:n, 1:n]\rangle$, and (3) gap problem: $A_{gap}(\ G[1:n, 1:n]\ )$, where $G[0:n, 0:n]$ is the $(n+1) \times (n+1)$ input matrix.

Floyd-Warshall's all pairs shortest path (FW-APSP) algorithm [10], [30] performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

Fig. 2 includes an iterative algorithm (PAR-LOOPS-FW) that computes the entries in $d[1:n, 1:n]$ assuming that each $d[i, j]$ is initialized with the weight of edge (v_i, v_j). The pseudocode for the CORDAC algorithm for solving this problem can be found in [8]. Table I shows that among the four recursive functions in the CORDAC algorithm, only D_{FW} is flexible which is also the dominating one.

III. OPTIMIZATIONS

In this section we discuss optimization strategies that we have used to significantly speed up implementations of the CORDAC algorithms described in Section II.

A. Hybrid CORDAC

To retain the benefits of both iterative and recursive algorithms, in practice all cache-efficient algorithms use a hybrid approach where recursive subdivision continues until the problem size becomes small enough (often called the base-case size) to fit into one of the cache levels (often the smallest one), after which a loop-based code is used to perform the computation [32]. These hybrid implementations expose optimization opportunities offered by neither the pure iterative nor the pure recursive implementation. The base-case kernels enjoy all benefits of loop-based DP (spatial locality, compiler assisted optimizations, such as, prefetching of required data, automatic vectorization, parallelization, processor pipelining, ILP, and so on), in addition to the temporal locality achieved by recursive divide-and-conquer. Moreover, for DP problems, this hybrid approach generates flexible instances of recursive functions and base-case kernels which brings the following additional benefits.

▷ *Asymptotic Improvement in Parallelism:* In our two-way divide-and-conquer approach a flexible recursive function can update all four quadrants of its output submatrix in parallel as long as it avoids race conditions by not updating the same quadrant simultaneously from two or more different recursive function calls. Such a function achieves $\Theta(n)$ span (i.e., $\Theta(n^2)$ parallelism) which is the same as that achieved by the cache-oblivious recursive matrix multiplication algorithm [13]. Table I shows that each of our CORDAC algorithms has at least one such flexible function and such functions are asymptotically dominating in the sense that almost all computations in the algorithm are performed inside the base cases of those functions. As a result, our CORDAC algorithms often have asymptotically better parallelism than the original parallel looping code (see Table I) with or without tiling.

▷ *Highly Optimizable Base Cases:* Running loop-based code on small flexible base-cases is often more efficient than running the same code on the original larger input as the former has better opportunities for *vectorization, parallelization* (comes from flexibility) and *spatial locality* (due to flexible loop reordering and *copy-optimization*).

For example, although $A_{par-loop}$ in Fig. 3 has the same inflexible implementation as PAR-LOOP-PARENTHESIS in Fig. 2, C_{loop} is much more flexible and amenable to optimizations than A_{loop}. While B_{loop} is not as optimizable as C_{loop}, it still can be optimized better than A_{loop}. Thus the recursive decomposition exposes many optimization opportunities over the traditional looping code. Simply by converting the loop algorithm to a hybrid CORDAC algorithm, we got around $4 - 75\times$ speedup without optimizing it further.

B. Optimizing Kernel Functions

In addition to compiler-assisted optimizations (e.g., vectorization) we use the following major optimizations to speed up the iterative kernels of our CORDAC algorithms.

▷ *Copy-optimization:* We copy the data into local $b \times b$ (b = base-case size) static arrays inside the kernel, and then

read from those local arrays during actual computation. This can improve performance provided the cost of computation is asymptotically higher than the cost of copying. Copy-optimization improves spatial locality as those copied arrays are allocated in thread local stacks, and can be accessed using a stride length of b instead of a stride length of n if originally read in a column-major order. Indeed, we only need to copy those matrices that are accessed in non-unit strides. The benefits of the copy-optimization become even more significant if one of the input matrices is accessed in column-major order (non-unit stride), and is converted to row-major while copying, so that it can be accessed in row-major order during the actual computation. Transposing the column-major accessed matrix during copy-optimization reduces cache-misses further as the converted local array can be accessed in unit-stride after the conversion. Copy-optimization also helps in reducing conflict misses significantly in set-associative caching systems.

▷ *Loop Reordering:* Inside flexible kernels it is possible to change the looping order without hampering the correctness of the algorithm which often improves spatial locality and vectorization efficiency. For example, it is well-known that for matrix multiplication (MM), i-k-j ordering (cache complexity: $\Theta\left(n^3/B + n^2\right)$, where B is cache line size) is typically faster than the i-j-k ordering (cache complexity: $\Theta(n^3)$). We observed the same for MM-like kernels when copy-optimization is not used. However, if copy-optimization is used inside the kernel to ensure unit stride data access, i-j-k looping order becomes faster than i-k-j looping order, especially for large n. Hence, we used i-j-k ordering with copy-optimization.

C. Data Layout

Laying out data in memory matching the order in which they are accessed during program execution can reduce cache misses by leveraging better spatial locality. For CORDAC algorithms, use of hybrid *Z-Morton Row-Major* (ZM_RM) layout is beneficial because that improves both temporal and spatial localities. In our experiments we have observed that for some values of n (e.g., powers of 2), use of ZM_RM layout instead of simple row-major layout can speed up a CORDAC algorithm by almost a factor of 2. Furthermore, use of ZM_RM layout reduces set-associativity conflict misses and capacity misses, if an appropriate base-case size is chosen. In all algorithms presented in this paper, we have used this ZM_RM layout and found that use of hybrid ZM_RM layout along with copy-optimization can remove the conflict miss problem almost entirely and gives consistently better performance.

▷ *Z-Morton for any n:* One of the contributions of this paper, which is, indeed, a by-product of our optimization efforts, is the use of a hybrid ZM_RM layout that works for any arbitrary $m \times n$ matrix and uses exactly mn space to store

$f_X(X, m, n, ...)$ (X is a pointer to an $m \times n$ matrix stored in ZM_RM layout. In $\mathcal{O}(1)$ time we compute pointers X_{11}, X_{12}, X_{21} and X_{22} pointing to the start of the 1st, 2nd, 3rd and 4th quadrants of X, respectively.)

1) c = largest power of $2 < \max(m, n)$
2) $m' = \min(c, m)$, $n' = \min(c, n)$, $m'' = \max(0, m - c)$, $n'' = \max(0, n - c)$
3) $X_{11} = X$, $X_{12} = X_{11} + m'n'$, $X_{21} = X_{12} + m'n''$, $X_{22} = X_{21} + m''n'$

Figure 4. On-the-fly computations of Z-Morton-row-major pointers.

mn elements. Although it is very straightforward to layout the data in ZM_RM when $m = n = 2^t$ for some $t > 0$, we are not aware of any prior work that uses hybrid ZM_RM for any arbitrary m, n while using exactly mn space. Also there is no closed form formula that can convert a row-major index to the corresponding ZM_RM index when the dimensions are arbitrary positive integers. There are ways of making Z-Morton work for any n through padding (see [28]), but padding uses extra space. As shown in the code snippet in Fig. 4, to use ZM_RM for any $m \times n$, in a (two-way) CORDAC algorithm, we first calculate a c such that $\max(m, n) > c \geq \max(m, n)/2$. Next we compute dimensions for four quadrants as shown in the pseudocode. Then we recursively put $m' \times n'$ items in the 1st quadrant (X_{11}), $m' \times n''$ items in the 2nd quadrant (X_{12}), $m'' \times n'$ in the 3rd quadrant (X_{21}), and remaining $m'' \times n''$ items in the 4th quadrant (X_{22}) in ZM_RM order. Hence, we do not need any extra space to hold the data for this kind of recursive ZM_RM data layout. From the size of each quadrant, we figure out the starting pointer of each quadrant (where to read/write the data) recursively using a CORDAC algorithm. After laying out data in ZM_RM layout in this way, during the computation, we use ZM_RM pointers and actual row-major indices to compute the locations on-the-fly inside the recursive functions which incurs only $\mathcal{O}(1)$ overhead per recursion level.

D. Auto vs. Explicit Vectorization

Sometimes explicit vectorization can lead to significant speedup over compiler-assisted auto-vectorization as compilers cannot always automatically detect all possible vectorization opportunities. Often vectorizing the base-case of the dominating kernel only (e.g., C_{loop} for the parenthesization problem) is enough to get the major share of the speedup.

Opportunities for Automation. Our optimization approach for CORDAC algorithms as described above is highly systematic, and we have observed that they work really well in practice. They are suitable for automation and perhaps incorporation into a smart compiler specialized for CORDAC.

IV. Experimental Results

In this section we demonstrate performance benefits of parallel CORDAC approach compared to parallel looping and parallel tiled approaches on multicores, manycores (Xeon Phi) and cluster of multicores. All implementations are in C++ with Intel Cilk Plus extension for shared memory and MPI for distributed memory parallelization. For each problem, we implemented four versions:

▷ *LOOPDP:* optimized parallel looping code with padding to mitigate set-associativity problem at powers of 2.

▷ *CO:* unoptimized parallel CORDAC.

▷ *CO_Opt:* optimized parallel CORDAC with copy-optimization.

▷ *COZ:* CO_Opt with ZM_RM layout for data storage.

For Parenthesization and FW-APSP problems, we further optimized COZ versions with explicit vectorization for CPU and Xeon Phi architectures separately. Furthermore, we implemented a *hybrid CPU + Xeon Phi* version where we dynamically offload subproblems to the coprocessor if it is idle and use the CORDAC approach to solve them on the coprocessor. All versions incorporate compiler-assisted optimizations. We compiled all programs with `-O3 -ip -parallel -AVX` optimization parameters. We used a base-case size of 64×64 for parenthesis, gap and protein folding and 128×128 for FW-APSP. Machines from the Stampede Supercomputing Cluster [27] were used to run the experiments and the system specifications can be found in Table II. We used PAPI-5.2 [4] to collect cache misses and LIKWID [29] for the energy/power statistics. Metrics shown in Table III were used to compare performance of different algorithms.

Property	Intel32	Intel16	XeonPhi	Intel16E (in house)
System	Intel Xeon E5 – 4650	Intel Xeon E5 – 2680	Knights Corner	XeonCPU E5 – 2650
Clock	2.70 GHz	2.70 GHz	-	2.00 GHz
# Cores	4x8 (32)	2x8 (16)	61	2x8 (16)
L1 data cache	32 KB	32 KB	32 KB	32 KB
Last-level cache	20 MB	20 MB	512 KB	20 MB
Memory	1 TB	32 GB	8 GB	32 GB
OS	CentOS 6.3	CentOS 6.3	CentOS 6.3	Debian
Compiler	icc v13.0	icc v13.0	icc v13.0	icc v13.0

Table II
SYSTEM SPECIFICATIONS. INTEL16E: USED FOR POWER & ENERGY ANALYSIS.

Metrics	Meaning	Expected
UPS	Number of Updates Per Second	Higher
Strong Scalability $\frac{T_1^l}{T_p}$	Running time of LOOPDP on 1 core/ Running Time of CORDAC on p cores	Higher

Table III
METRICS USED FOR PERFORMANCE COMPARISON.

A. Performance on Shared-Memory Machines

Speedup and Scalability on CPU, Xeon Phi and Hybrid Platforms. We ran all programs on the Intel16 and Xeon Phi Machines with $n \approx 100$ to $n \approx 16000$ where $n \times n$ is size of the DP table. Table IV summarizes the results. We observed that explicit vectorization contributed up to $5\times$ speedup over the auto-vectorized code. For parenthesization problem, the explicitly vectorized COZ runs $278\times$ and hybrid *CPU + Xeon Phi* version runs $395\times$ faster; for FW-APSP explicitly vectorized COZ is $24\times$ and *CPU + Xeon Phi* is $35\times$ faster than LOOPDP for $n = 32768$.

Overall, hybrid *CPU + Xeon Phi* version runs $42 - 50\%$ faster than the pure vectorized CPU version for $n \approx 2^{15}$.

Runtime, Cache Miss, Energy Performance and Scalability.

Problem	Speedup w.r.t parallel LOOPDP on 16 cores CPU $n = 16384$				Scalability ($n = 8192$, p varies from 1 to 16)	
	CPU		Xeon Phi	Hybrid: CPU + Xeon Phi	CO	LOOPDP
	Not Explicitly Vectorized	Explicitly Vectorized	Explicitly Vectorized	Explicitly Vectorized	Not Explicitly Vectorized	
Parenthesization	31	160	132	156	linear	linear
Gap	23	–	–	–	linear	linear
FW-APSP	11	22	26	25	linear	Does not scale
ProteinFolding	5	–	–	–	linear	Does not scale

Table IV
A SUMMARY OF THE EXPERIMENTAL RESULTS.

Fig. 5 shows performance trends for all four problems on Intel16 in terms of UPS, Strong Scalability, cache-miss and energy consumption ratios. Clearly, CORDAC algorithms (COZ, CO_Opt, CO) outperform LOOPDP under all these metrics. Overall, COZ algorithms are $1.2 - 2\times$ and CO_Opt algorithms are $1.1 - 1.8\times$ faster than the corresponding unoptimized CO algorithms. For all four problems, the UPS curve of the unoptimized CO algorithm has occasional dips due to set-associativity conflict misses. We were able to avoid those dips in CO_Opt and COZ versions using our optimizations. For parenthesization and gap problems the speedups w.r.t LOOPDP are more compared to FW-APSP and protein folding. Theoretical bounds listed in Table I also support this result. Observe that the serial cache complexity of iterative algorithms of the first group is $\Theta\left(n^3\right)$ and the second group is $\Theta\left(n^3/B\right)$. Similarly, the scalability of LOOPDP for the first group of problems is also better than the second group. Fig. $5(b)$ shows that CORDAC algorithms always incur fewer cache misses in all levels of caches for $n \geq 1000$ which is one of the main contributing factors in the reduction of running times and energy consumptions. Fig. $5(d)$ shows that in addition to being faster than LOOPDP, CORDAC algorithms consume $4 - 30\times$ less energy for the entire Package (die), PP0 (cores and their private caches) and DRAM. Clearly, smaller running times and fewer cache misses contributed to this reduction in energy consumption.

Results on Larger Input Sizes. In Fig. 6 we show that on Intel32 speedup of CORDAC algorithms w.r.t LOOPDP increases with the increase of the number of cores, p and input size, n. The speedup numbers on Intel32 are almost $2\times$ of what we achieved on Intel16 for $n \leq 16$K. Seemingly, the LOOPDP algorithms are not able to get the benefit of increased number of cores to the same extent as the CORDAC algorithms are able to do.

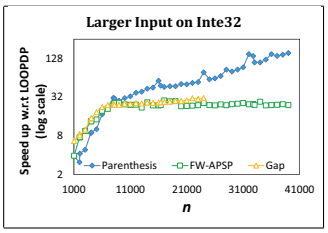

Figure 6. Speedup w.r.t LOOPDP with larger input sizes on Intel32.

Tradeoff Between Runtime and Power Consumption. Since $Power = \frac{Energy}{Time}$, as running time increases, power consumption decreases if energy remains constant. However, in general, energy consumption also increases with running time. Energy consumption per computation increases as

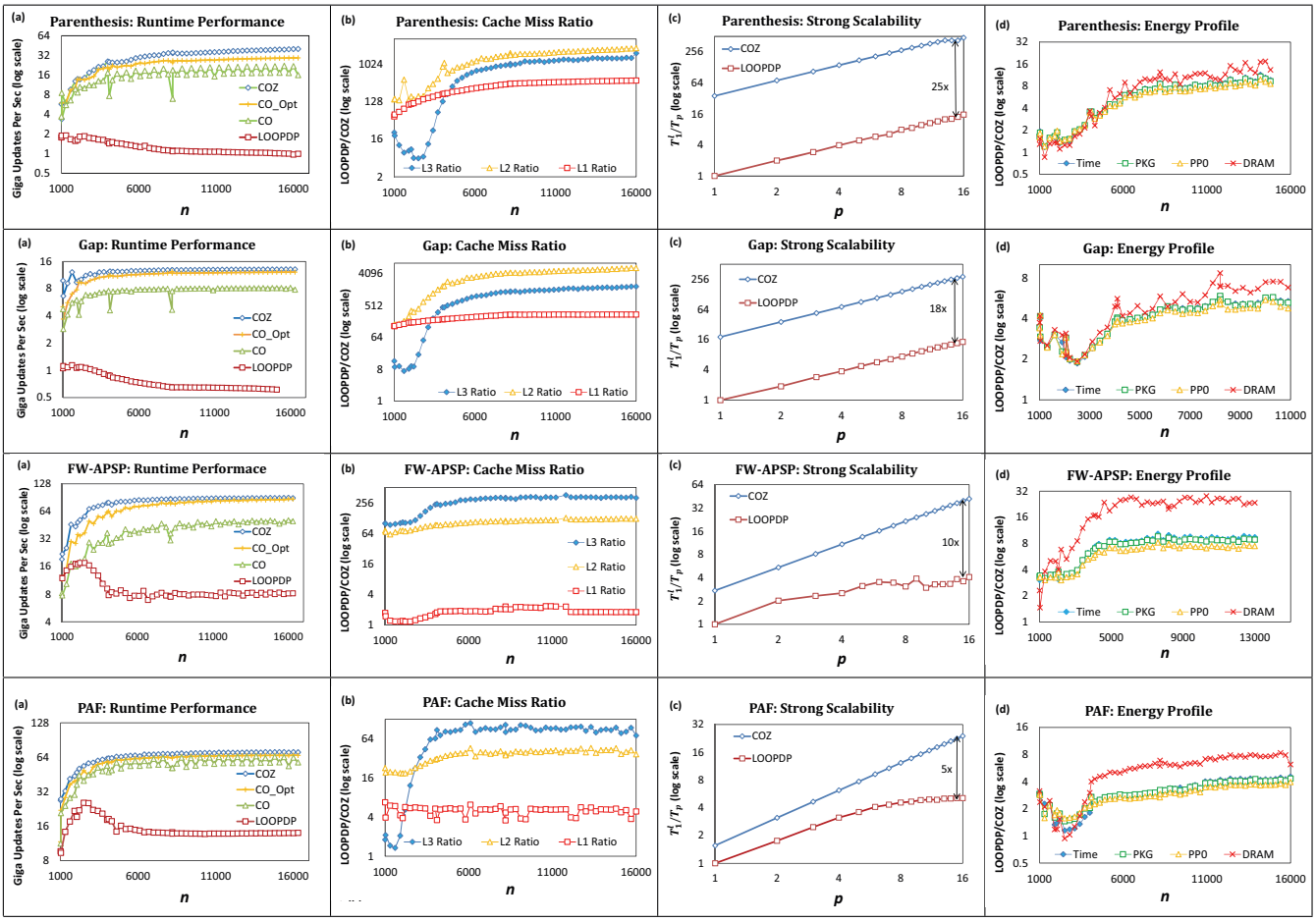

Figure 5. (a) Giga updates per second achieved by all algorithms, (b) ratios of total shared and private cache misses, (c) strong scalability with #cores, p when n is fixed at 8192, and (d) ratios of total joule energy consumed by Package (PKG), Power Plane 0 (PP0) and DRAM.

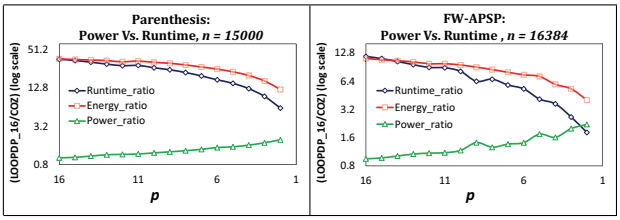

Figure 7. COZ has the flexibility to use fewer number of cores while still running faster but consuming less energy and power than LOOPDP.

the number of cores, p decreases, but energy consumption per memory access increases as p increases. Therefore, the relationship is not linear. To explore the power and runtime tradeoff, we ran the LOOPDP version on $p = 16$ cores on Intel16E and then varied p from 16 down to 1 to get the power, runtime and energy values for the COZ version. Fig. 7 shows the result. As we decrease p, power consumption of COZ algorithm decreases (ratio $\frac{LOOPDP}{COZ}$ increases), while the running time as well as energy consumption increases. On 16 cores, although CORDAC algorithms consume less energy and run $5 - 40\times$ faster than LOOPDP, the power consumed is approximately the same (ratio is close to 1) for a given input size. On the other hand, on 2 cores, although CORDAC algorithms consume less energy and power, they still run faster than LOOPDP. Therefore, if power consumption is a concern, CORDAC algorithms have the flexibility to run on fewer cores, while still running faster than LOOPDP.

Comparison with Parallel Tiled Codes Generated by Polyhedral Compilers. We compare our COZ and LOOPDP implementations with parallel tiled codes generated by state-of-the-art polyhedral compilers, PLuTo [3], PoCC [23] and Polly [16]. Fig. 8 and 9 show comparison of our implementations with the best result obtained from these compilers. Clearly, CORDAC algorithms run $3 - 30\times$ faster in the given input range. After analyzing the codes generated by these compilers, we found that

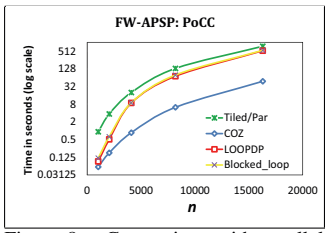

Figure 8. Comparison with parallel tiled code generated using polyhedral compilers PLuTo [3], PoCC [23] and Polly [16].

for each of these problems, the compilers were able to parallelize only one of the 3 nested *for loops*. Hence for FW-APSP, the parallelism and span of the generated code were worse than our LOOPDP implementation which had 2 parallel *for loops*.

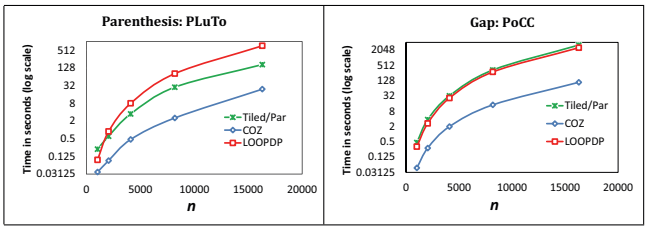

Figure 9. Comparison with parallel tiled code generated using polyhedral compilers PLuTo [3], PoCC [23] and Polly [16].

We implemented a parallel tiled version for FW-APSP with 2 nested *for loops*, and the results are comparable to that of LOOPDP. For parenthesization and gap problems, both our LOOPDP and the codes generated by the polyhedral compilers used 1 parallel *for loop*, however, the two codes were parallelized differently in each case. None of the generated tiled codes had temporal locality.

V. Extension to Distributed-Memory Settings

In this section we describe how to extend a CORDAC algorithm to the shared-distributed-shared-memory setting which applies to all four of our CORDAC algorithms.

Prior Work. To implement divide-and-conquer algorithms under distributed-memory settings, both static and dynamic load-balancing strategies have been used. In [26], a pure distributed-memory algorithm has been implemented for the parenthesization problem, where the rows are evenly distributed among the processes each of which uses a CORDAC algorithm to solve the subproblem assigned to it. Dynamic load-balancing approaches map the recursive division part to the available MPI processes, and then use a shared-memory algorithm inside each process when no more process is left to take the responsibility of a division. In [24] this approach has been used for mergesort algorithm, and the authors concluded that in general, a shared-memory algorithm provides better performance than the corresponding distributed or distributed-shared-memory algorithms while using the same number of cores. Although this approach can be used for our CORDAC algorithms, it is likely to be more complicated than mergesort due to the nested nature of multiple recursive functions, since we may need to devise different algorithms for each of the A, B, C and D functions.

Our Approach. In this paper, we propose a novel shared-distributed-shared-memory (SDSM) framework for our CORDAC algorithms which performs dynamic load-balancing on a cluster of multicores without any change in the basic CORDAC structure. We use hierarchical dynamic load-balancing and work-stealing to balance the load among the processes. In this approach the available processes are arranged in a multi-level hierarchy of masters and workers with all non-master pure workers placed as leaves. We describe a $3-$level hierarchy below.

If we have K processes, we use one of them as a super-master, some M' of them as masters and the rest as workers. The super-master, masters and workers run multithreaded codes on p cores. The master processes work as workers for the super-master. In the super-master and master processes, 1 out of p threads is used as a dispatcher which dispatches work dynamically to the available free workers and also collects the results back from the workers.

Each super-master and master process maintains a shared job queue that can be accessed exclusively by all threads in it. If a thread is about to run a function (e.g., C) on input size x, where *min offload threshold* $\leq x \leq$ *max offload threshold*, the thread tries to lock the job queue and in case of a successful locking, it puts at most $l-1$ out of its l parallelly executable recursive sub-divisions in the job queue and works on the rest while waiting for the results of the offloaded parts to come back. If a thread finishes before all its submitted jobs in the queue are processed, it steals back its latest submitted jobs left in the job queue in the current recursion level (if available) and works on the stolen part using the original SDSM algorithm as before. If a thread is unable to submit a job (because the job queue is full), it will simply go ahead and divide the job even further and try again to access the job queue in the next recursion level.

A worker process, on the other hand, waits for jobs from its master, and if it gets one, it solves that using the shared-memory parallel CORDAC approach and returns the result back after it is done.

Distribution with Cannon's/Fox's Algorithms. One may argue that the dynamic distribution is not scalable for distributed settings. We show that dynamic distribution works pretty well for these algorithms since the overall communication complexity is asymptotically lower than the computational complexity. Nevertheless, it is possible to adapt the popular Cannon's [5] or Fox's [11] algorithms for matrix-multiplication that have linear scalability and unit efficiency for distribution, if the flexible kernel looks MM-like (i.e., two input matrices and one output matrix) which is indeed the case for our dominating flexible kernels.

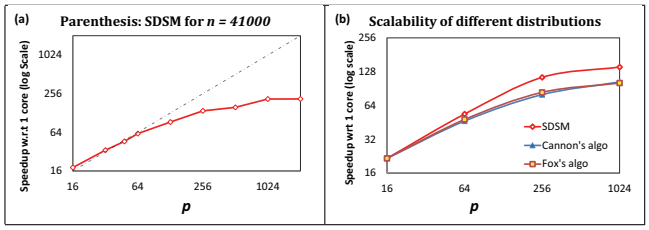

Figure 10. Parenthesization Problem: (a) Scalability of shared-distributed-shared-memory algorithm (offloading functions C and B). (b) Performance comparison of different work distribution techniques (offloading C only).

To distribute using Cannon's/Fox's method, one thread

of the master process first locks all worker processes and then uses the Cannon's/Fox's algorithm to distribute the work evenly among the processes. Once that thread of the master process gets the results back, it frees the lock and the workers become available for use by any thread.

A. Shared-Distributed-Shared-Memory Results

We implemented our SDSM algorithm for parenthesization problem. Fig. 10(a) shows strong scalability (T_1/T_p, where T_p = running time on p cores and T_1 = running time of CORDAC on 1 core) of our SDSM algorithm for the parenthesization problem where we allowed offloading of functions B and C. Function A was executed entirely on the super-master. Offloading B in addition to C improves performance by 20%. For this experiment, we fixed n at 41K and allowed offloading of problems with a size in the range of $256-2048$. We found that the scalability was almost linear till $p = 64$ cores, and overall the algorithm scaled well till 1024 cores. As we increased the number of cores, the average percentage of idle time as well as communication time increased suggesting that there was not enough work to keep all cores busy all the time (at each recursion level).

In Fig. 10(b) we compare the performance of 3 different work-distribution strategies (SDSM, Fox's and Cannon's) for distributing work in $C_{par}(X, U, V)$. For this version of SDSM we only allowed distribution of function C since function B is not MM-like and cannot be distributed using Cannon's/Fox's algorithms. We found that SDSM performs better than the other two approaches even though it uses a hierarchical dynamic load-balancing strategy.

B. Communication Complexity

Computing precise communication complexity of our SDSM algorithm is quite involved because of dynamic load-balancing and interactions with Cilk's randomized work-stealing scheduler. However, deriving an upper bound is fairly straightforward. Table I shows that each problem generates $O((n/b)^3)$ subproblems of size $b \times b$ each. If we only solve subproblems of size $2^s \times 2^s$ where $q \leq s \leq r$ on worker processes, communication complexity will be upper-bounded by $O(\sum_{q \leq s \leq r} ((n/2^s)^3) \times (2^s)^2) = O(n^3/2^q)$. For example, we used $2^q = \Omega(\sqrt{n})$ in our experiments which led to an $O(n^{2.5})$ bound. Since the master process holds the entire DP table, the same bound holds for our approach based on Cannon's MM algorithm.

REFERENCES

[1] V. Bafna and N. Edwards, "On de novo interpretation of tandem mass spectra for peptide identification," in *Proc. RECOMB*, 2003.

[2] R. Bellman, *Dynamic Programming*. Princeton Univ. Press, 1957.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *ACM SIGPLAN Notices*, 43(6):101–113, 2008.

[4] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proc. SC*, 2000.

[5] L. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. Thesis, Montana State University, 1969.

[6] R. Chowdhury, H.-S. Le, and V. Ramachandran, "Cache-oblivious dynamic programming for bioinformatics," *TCBB*, 7(3):495–510, 2010.

[7] R. Chowdhury and V. Ramachandran, "Cache-efficient dynamic programming algorithms for multicores," in *Proc. SPAA*, 2008.

[8] R. Chowdhury and V. Ramachandran, "The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation," *TOCS*, 47:878–919, 2010.

[9] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge Univ. press, 1998.

[10] R. Floyd, "Algorithm 97 (Shortest path)," *CACM*, 5:345, 1962.

[11] G. Fox, S. Otto, and A. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication," *Parallel Computing*, 4(1):17–31, 1987.

[12] M. Frigo, P. Halpern, C. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *Proc. SPAA*, 2009.

[13] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. FOCS*, 1999.

[14] Z. Galil and R. Giancarlo, "Speeding up dynamic programming with applications to molecular biology," *TCS*, 64(1):107–118, 1989.

[15] Z. Galil and K. Park, "Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency," *JPDC*, 21:213–222, 1994.

[16] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation, *PPL*, 22(4), 2012.

[17] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge Univ. Press, 1997.

[18] J. Karlander, "Algorithms and complexity (exercise 3+4)," 2013. [Online]. Available: www.csc.kth.se/utbildning/kth/kurser/DD2352/algokomp13/

[19] C. Leiserson, R. Rivest, C. Stein, and T. Cormen, *Introduction to Algorithms*. The MIT press, 2001.

[20] A. Lew and H. Mauch. *Dynamic Programming: A Computational Tool*, volume 38. Springer, 2006.

[21] R. Lyngs, M. Zuker, and C. Pedersen, "Fast evaluation of internal loops in RNA secondary structure prediction." *Bioinformatics*, 15(6):440–445, 1999.

[22] Y. Park, S. Shackney and R. Schwartz, "Network-based inference of cancer progression from microarray data". *TCBB*, 6: 200-212, 2009.

[23] L. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Combined iterative and model-driven optimization in an automatic parallelization framework," in *Proc. SC*, 2010.

[24] A. Radenski, "Shared memory, message passing, and hybrid merge sorts for standalone and clustered SMPs," in *Proc. PDPTA*, 2011.

[25] M. Sniedovich, *Dynamic Programming: Foundations and Principles*. CRC press, 2010.

[26] G. Tan, S. Feng, and N. Sun, "Locality and parallelism optimization for dynamic programming algorithm in bioinformatics," in *Proc. SC*, 2006.

[27] Texas Advanced Computing Center, https://www.tacc.utexas .edu/.

[28] J. Thiyagalingam, O. Beckmann, and P. Kelly, "Minimizing associativity conflicts in Morton layout," in *Proc. PPAM*, 2006.

[29] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. ICPPW*, 2010.

[30] S. Warshall, "A theorem on Boolean matrices," *JACM*, 9:11–12, 1962.

[31] M. Waterman, *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman & Hall Ltd, 1995.

[32] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proc. SPAA*, 2007.