Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Engineering High-performance Parallel Algorithms with Applications to Bioinformatics

A Dissertation presented

by

Jesmin Jahan Tithi

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2015

Copyright by
Jesmin Jahan Tithi
2015
ALL RIGHTS RESERVED.

Stony Brook University

The Graduate School

Jesmin Jahan Tithi

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Rezaul Chowdhury
Assistant Professor, Department Computer Science

Steven Skiena
Professor, Department Computer Science

I.V. Ramakrishnan
Professor, Department Computer Science

Michael Bender
Professor, Department Computer Science

Robert J. Harrison
Professor and Director, Institute for Advanced Computational Science - Stony Brook

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Engineering High-performance Parallel Algorithms with Applications to Bioinformatics

by

Jesmin Jahan Tithi

Doctor of Philosophy

in

Computer Science

Stony Brook University

2015

Since the beginning of the last decade, plateauing of the clock speed of computer processors has forced us to invest more in parallelism — for both hardware and software. This resulted in improvements in computing technologies that have favored parallelism over increased clock speed. Taking advantage of these improvements requires designing algorithms that can leverage parallelism well. In this dissertation, we show how to take advantage of several algorithm design techniques to harness modern heterogeneous parallel architectures for solving problems in bioinformatics efficiently. Our main goal while designing algorithms is to achieve high-performance in terms of running time and scalability. Other desirable goals include energy efficiency, portability and adaptivity.

We solve bioinformatics problems on grids (dynamic programming problems), on graphs (breadth-first search), and problems that can be solved using spatial trees (Molecular Dynamics using octrees). We present many novel algorithms, algorithm engineering techniques, theoretical analyses and performance evaluations on a range of state-of-the-art parallel architectures including multicores, manycores, and special purpose accelerators. Although we mainly target problems in bioinformatics, the algorithmic techniques we use to solve those problems have general applicability.

For many dynamic programming problems, we show that if we use a *cache-oblivious recursive divide-and-conquer* technique to solve them, the resulting algorithms become highly optimizable, cache-optimal and often have asymptotically better parallelism than their standard iterative counterparts. These algorithms not only have good theoretical bounds, but also perform better than standard iterative and tiled-loop algorithms in terms of running time, scalability, energy-efficiency, cache-adaptivity and portability. Furthermore, it is often possible to improve parallelism of these recursive algorithms without sacrificing cache-optimality by removing *artificial dependencies* among the tasks introduced by the recursive structure of the algorithm.

Breadth-first search is a popular graph traversal algorithm that has many applications in bioinformatics. We show how to use *lock- and atomic instruction-free optimistic parallelization* to improve parallelism and load balancing in a shared-memory parallel breadth-first search (BFS) algorithm. We present several work-efficient parallel BFS algorithms (including one that uses recursive divide and conquer) along with their theoretical and empirical performance analyses on state-of-the-art multicore and manycore architectures.

Spatial trees (e.g., quad tree, octree, k-D tree) are recursive space partitioning data structures that are often used to store biological molecules efficiently. We present octree-based distributed and distributed-shared-memory hybrid near-far approximation algorithms to compute molecular polarization energy. These algorithms outperform all other state-of-the-art Molecular Dynamics packages by orders of magnitude on multicores and clusters of multicores.

We conclude by discussing implications of our work for future parallel algorithm design, and ways to extend our research to other domains.

Dedication Page

The thesis is dedicated to my parents Md. Jahan Ali and Monju Ara for their endless love, support and encouragement.

Contents

Contents i

List of Figures v

List of Tables x

1 Introduction 1
- 1.1 Part I. Algorithms on grids: Dynamic programming . . . 5
- 1.2 Part II. Algorithms on graphs: Breadth-first search . . . 7
- 1.3 Part III. Algorithms using spatial trees: Molecular energetics . . . 8

I Algorithms on Grids 9

2 Exploiting Spatial Architectures for Edit Distance Algorithms 10
- 2.1 Abstract . . . 10
- 2.2 Introduction . . . 10
- 2.3 Background . . . 12
  - 2.3.1 Spatial Architectures . . . 12
  - 2.3.2 Edit Distance . . . 14
    - 2.3.2.1 Overview of the Edit Distance Problem . . . 14
    - 2.3.2.2 Exploitation of Pipeline Parallelism . . . 15
- 2.4 Edit Distance on Spatial Architectures . . . 15
  - 2.4.1 Designing a Worker . . . 16
    - 2.4.1.1 Optimization . . . 16
    - 2.4.1.2 Mapping . . . 18
  - 2.4.2 Naïve Implementation . . . 18
  - 2.4.3 Optimization: Use of Linear Memory Space . . . 18
  - 2.4.4 Strip Mining . . . 19
    - 2.4.4.1 Strip Mining using Memory . . . 19
    - 2.4.4.2 Strip Mining using PEs' scratchpad memory . . . 20
  - 2.4.5 Tiling . . . 21
  - 2.4.6 Linear Memory Space Traceback Path . . . 22
- 2.5 Experimental Setup . . . 23
  - 2.5.1 Spatial Architecture Performance . . . 23
  - 2.5.2 Spatial Edit Distance Implementation . . . 24
  - 2.5.3 x86 Comparison . . . 24
- 2.6 Experimental Results and Analysis . . . 25
  - 2.6.1 Overview of Performance Results . . . 25
  - 2.6.2 Memory References and Communication . . . 27
  - 2.6.3 Coding Effort Analysis . . . 27
- 2.7 Related Work . . . 28
- 2.8 Conclusion and Future Research . . . 29

3 Recursive Dynamic Programming With Matrix-multiplication-like Flexible Kernels **30**
3.1 Abstract 30
3.2 Introduction 31
3.3 Algorithms 33
3.3.1 Parenthesis Problem 33
3.3.2 Protein Accordion Folding 36
3.3.3 Sequence Alignment with General Gap Penalty 38
3.3.4 All Pairs Shortest Path Problem 40
3.4 Optimizations 40
3.4.1 Hybrid CORDAC 40
3.4.2 Optimizing Kernel Functions 41
3.4.3 Data Layout 42
3.4.4 Auto vs. Explicit Vectorization 43
3.4.5 Opportunities for Automation 43
3.5 Experimental Results 43
3.5.1 Performance on Shared-Memory Machines 44
3.5.2 Extension to Distributed-Memory Settings 49
3.5.3 Shared-Distributed-Shared-Memory Results 52
3.5.4 Communication Complexity 52
3.6 Conclusions 52

4 Cache-adaptivity, Bandwidth Benefit and Robustness of Recursive Divide and Conquer **54**
4.1 Abstract 54
4.2 Introduction 55
4.3 Adaptivity and Robustness 56
4.4 Experimental Results 57
4.4.1 Robustness 58
Parenthesis Problem. 58
Floyd-Warshall's APSP. 59
Gap Problem. 60
4.4.2 Cache-adaptivity 62
4.5 Conclusion 62

5 Provably Efficient Scheduling of Cache-Oblivious Recursive Wavefront Algorithms **64**
5.1 Abstract 64
5.2 Introduction 65
5.3 Deriving recursive wavefront algorithms 68
5.3.1 Constructing completion-time function 71
5.3.2 Deriving a recursive wavefront algorithm 74
5.4 Applications 75
5.5 Scheduling recursive wavefront algorithms 76
5.6 Experimental results 77
5.7 Future Research 82

6 An Efficient Cache-oblivious Viterbi Algorithm **83**
6.1 Abstract 83
6.2 Introduction 84
6.3 Viterbi algorithm using rank convergence 85
6.3.1 Maleki et. al.'s algorithm using rank-convergence 86
6.3.2 An improved processor-oblivious algorithm 87

6.4 Cache-efficient multi-instance Viterbi algorithm . . . 88
6.5 Cache-efficient Viterbi algorithm . . . 90
6.6 Experimental results . . . 91
6.6.1 Multi-instance Viterbi: Iterative vs. Recursive . . . 92
6.6.2 Single-instance Viterbi: Efficient recursive vs. Maleki et. al.'s . . . 93
6.7 Conclusions and Future Research . . . 93

II Algorithms on Graphs 95

7 Optimistic Parallelization: Avoiding Locks and atomic instructions in Shared-memory Parallel BFS 96
7.1 Abstract . . . 96
7.2 Introduction . . . 97
7.3 Prior Work . . . 99
7.4 Our Contributions . . . 99
7.5 Our Parallel BFS Algorithms for Multicores . . . 100
7.5.1 Based on Centralized Queues . . . 102
7.5.2 Based on Distributed Randomized Work-stealing . . . 103
7.6 Extension to NUMA . . . 104
7.7 Discussion: Further Improvements . . . 104
7.8 Correctness . . . 105
7.9 Complexity Analyses . . . 106
7.9.1 BFS_C (Centralize + Lock) . . . 106
7.9.2 BFS_W (Work-stealing + Lock) . . . 107
7.9.3 BFS_{WS}(Work-stealing + Scalefree + Lock) . . . 109
7.10 Experimental Results . . . 109
7.10.1 Simulation Environment and Input Graphs . . . 109
7.10.2 Performance Analysis . . . 110
7.10.3 Why Lockfree Does Better Load-balancing than Lock-based . . . 113
7.10.4 Explicit vs. Implicit Work-stealing . . . 113
7.10.5 Performance of Work-stealing on Intel Xeon Phi . . . 114
7.11 Conclusion and Future Research . . . 117

8 Theoretically Optimal Level-synchronous Parallel Work-aware BFS 118
8.1 Abstract . . . 118
8.2 Introduction . . . 118
8.3 Algorithm . . . 119
Theoretically optimal BFS . . . 119
8.4 Analysis . . . 120
8.5 Experimental Results . . . 122
8.6 Conclusion and Future Research . . . 125
8.7 Acknowledgment . . . 126

III Algorithms using Spatial Trees 127

9 Hybrid Algorithm using Octrees: Polarization Energy on Clusters of Multi-cores 128
9.1 Abstract . . . 128
9.2 Introduction . . . 128
9.3 Background . . . 130
9.4 Related Work . . . 133
Popular Parallel E_{pol} Implementations . . . 133

9.5 Our Contributions . . . 133
9.5.1 Load Balancing . . . 134
9.5.2 Algorithm . . . 136
9.5.3 Analysis of Time Complexity . . . 138
9.6 Simulation Results . . . 139
9.6.1 Dealing with NUMA Effect . . . 139
9.6.2 Scalability . . . 140
9.6.3 Running Time and Speedup . . . 140
9.6.4 Energy Value . . . 142
9.6.5 Change in Error and Runtime with Approximation Parameter . . . 142
9.6.6 Scalability with Larger Molecule . . . 143
9.6.7 Comparison with Amber GPU Implementations . . . 143
9.6.8 Full Vs. Half Energy . . . 144
9.7 Conclusion . . . 145
9.8 Acknowledgment . . . 145

10 Future Research **146**

List of Figures

2.1 Spatial programming example. Converting a dataflow graph to a spatial pipeline of regions. 12
2.2 Example spatial architecture. Network of PEs, scratchpad memory, and caches are shown alongside a PE diagram. 13
2.3 Solving for edit distance using dynamic programming. The dark-shaded cells are the edits for solving the base cases, and the light-shaded cells are the required minimum edits. 14
2.4 Data flow dependencies for calculating a cell in edit distance. Dependencies are found along the row and column of the cost matrix, which limits the ability to vectorize. 15
2.5 (A) A simple cell worker that computes a single cell of the cost matrix, (B) An optimized row worker. 16
2.6 Pseudocode for the calculation of a single cell. 16
2.7 Flow chart for the control flow path of a single worker. Blue arrows show state transitions, green arrows show self-feedback, and red arrows show communication with other PEs. 17
2.8 (A) Two row workers working on two consecutive rows and connected together using communication channels. (B) A possible layout of the row workers on a grid of PEs. 18
2.9 Strip mining using standard memory and scratchpad memory. Initial and final rows are read from and written to memory, while the intermediate rows use either memory or scratchpad memory. 19
2.10 Tiled approach. Computation occurs on a column strip of tiles. Initial and final rows are read from and written to memory, while intermediate rows use scratchpad memory. Column results are saved to memory as well. 21
2.11 Sample code for a module that matches S[i] with T[j] and computes the cost of cell M[i][j] based on the *diagonal* cell M[i-1][j-1]. 24
2.12 Execution pattern of the x86 based tiled-loop code. The numbers on the tile shows when a tile get executed. Tiles with the same numbers gets executed in parallel. 25
2.13 Results showing that implementations on triggered instruction spatial architecture (TIA) run 50× faster while consuming 1/100× energy compared to the optimized single-threaded x86 based tiled-loop implementation. 26
2.14 Comparison of memory, local communication, and scratchpad memory accesses between x86 and TIA based implementations. While register file activity is not shown, TIA local communication numbers include what would be register file accesses on x86. 27

3.1 Inflexible looping code for the parenthesis problem vs. the flexible looping code for matrix multiplication. 31

3.2 Loop-based parallel codes for the parenthesis problem (Par-Loop-Parenthesis), Floyd-Warshall's APSP (Par-Loop-FW), protein accordion folding (Par-Loop-Protein-Folding) and the gap problem (Par-Loop-Gap). 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 [82]. 34
3.3 A protein accordion fold where each star represents a hydrophobic amino acid and each circle a hydrophilic one. The accordion score of this folded sequence is 4 which is not the maximum possible accordion score for this sequence. 37
3.4 SOF(i, j, k) counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i : k]$ is folded only once at indices $(j, j + 1)$. In this figure, each star represents a hydrophobic amino acid and each circle a hydrophilic one. 37
3.5 Parallel cache-oblivious recursive divide-and-conquer (CORDAC) algorithms for solving the parenthesis and the protein accordion folding problems. For simplicity, we assume n to be a power of 2. Initial function calls are as follows. (1) parenthesis problem: $A_{par}(c)$ for an $n \times n$ input matrix c and (2) protein accordion folding $A_{fold}(\ X\ )$, where $X = \langle S[1 : n, 1 : n], F[1 : n, 1 : n]\rangle$ are $n \times n$ input matrices. . 38
3.6 Parallel cache-oblivious recursive divide-and-conquer (CORDAC) algorithms for solving the gap problem. For simplicity, we assume n to be a power of 2. $A_{gap}(\ G[1 : n, 1 : n]\ )$ is the initial function call, where $G[0 : n, 0 : n]$ is the $(n + 1) \times (n + 1)$ input matrix. 39
3.7 On-the-fly computations of Z-Morton-row-major pointers. 42
3.8 Benefit of explicit vectorization over autovectorization for FW-APSP code. . . . 43
3.9 Parenthesis problem: (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. 45
3.10 Gap problem: (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. 45
3.11 Floyd-Warshall's all pairs shortest path: (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. 46
3.12 Protein Accordion Folding: (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. 46
3.13 Evidence of better bandwidth utilization of CODRAC, wrt. the iterative algorithm (LOOPDP) for the parenthesis/matrix chain multiplication problem. . . . 47
3.14 Speedup w.r.t LOOPDP with larger input sizes on Intel32 machine. 47
3.15 Parenthesis Problem: Itemized breakdown of how much performance gain each optimization provides. Here, CO denotes an unoptimized CORDAC algorithm. . 48
3.16 Power and Runtime Tradeoff. CORDAC has the flexibility to use fewer number of cores while still running faster but consuming less energy and power than LOOPDP. 48
3.17 FW-APSP: Comparison with parallel tiled code generated using polyhedral compiler PoCC [148]. 49
3.18 Parenthesis and Gap Problems: Comparison with parallel tiled code generated using polyhedral compilers - PLuTo [29], PoCC [148] and Polly [95]. 50
3.19 Shared-Distributed-Shared-Memory (SDSM) framework for recursive divide-and-conquer algorithm with dynamic load-balancing on a cluster of multicores. 51
3.20 Parenthesis 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). 51

4.1 Code snippet for Tiled, CORDAC and Iterative algorithms. 56
4.2 (A) Rates of updates performed by multithreaded iterative and recursive (CORDAC) algorithms for the parenthesis problem, Floyd-Warshall's APSP [50], and the Gap problem [39] on $2^{13} \times 2^{13}$ integer matrices, as the number of processing cores varies. (B) Rates of updates performed by multithreaded iterative, recursive and tiled-loop code generated by PLuTo [29] for the parenthesis problem as the matrix dimension varies. The Optimized-Tiled-loop is a hand-optimized version of the tiled code auto-generated by PLuTo. 57
4.3 The plots show how the performances of standard iterative, tiled-loop and recursive codes for solving the parenthesis problem (for $n = 2^{13}$) are affected when multiple instances of the same program are run on an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. 59
4.4 The plots show how the performances of standard-iterative, tiled-loop and recursive codes for solving the Floyd-Warshall APSP problem (for $n = 2^{12}$) are affected as multiple instances of the same program are run on an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. 60
4.5 The plots show how the performances of standard iterative, tiled-loop and recursive codes for solving the Gap problem (for $n = 2^{13}$) are affected as multiple instances of the same program are run on an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. 61
4.6 The plots show how changes in the available shared L3 cache space affect the serial running time and the number of L3 cache misses of tiled-loop and recursive (CORDAC) algorithms for solving the parenthesis problem when $n = 2^{13}$. The code under test was run on a single core of an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. A multithreaded `Cache Pirate` [1, 70] was run on the remaining 7 cores to steal L3 cache space. 62

5.1 Left: The programmer derives the timing functions from a given standard 2-way recursive divide-and-conquer DP algorithm for the parenthesis problem. A matrix region Z has its top-left corner at (z_r, z_c) and is of size $n \times n$. Right: A recursive divide-and-conquer wavefront algorithm is generated for the parenthesis problem. The programmer derives the algorithm if work-stealing scheduler is used, and the scheduler derives the algorithm if modified hint-accepting space-bounded scheduler (Section 5.5) is used. The algorithm makes use of the timing functions derived by the programmer. 69
5.2 Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the Parenthesis Problem. All programs were run on 16 core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler. 78
5.3 Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the LCS Problem. All programs were run on 16 core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler. 78
5.4 Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the 2D FW-APSP Problem. All programs were run on 16 core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler. 79
5.5 Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the Parenthesis problem. All programs were run on 24 core machines in Comet. All implementations used Cilk Plus's work-stealing scheduler. 80
5.6 Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the LCS Problem. All programs were run on 24 core machines in Comet. All implementations used Cilk Plus's work-stealing scheduler. 81

5.7 Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the 2D FW-APSP Problem. All programs were run on 24 core machines in Comet. All implementations used Cilk Plus's work-stealing scheduler. 81

6.1 Processor-aware parallel Viterbi algorithm using rank convergence as given in Maleki et al. paper [126]. This algorithm is not cache-efficient. 85
6.2 Processor-oblivious parallel cache-inefficient Viterbi algorithm using rank convergence. 87
6.3 Cache-efficient parallel recursive divide-and-conquer multi-instance Viterbi algorithm. 88
6.4 An efficient cache- and processor-oblivious parallel Viterbi algorithm using rank convergence. Viterbi-MI refers to Viterbi-MI algorithm presented in Section 6.4. 90
6.5 Running time and L3 miss of our cache-efficient multi-instance Viterbi algorithm and the multi-instance iterative Viterbi algorithm. 91
6.6 Running time and L3 miss of our cache-efficient Viterbi algorithm and comparison with Maleki et. al.'s algorithm. 92
6.7 Energy / power consumption of our and Maleki et al.'s algorithms. 93

7.1 Scalability of lockfree parallel BFS algorithms running on (a) Lonestar and (b) Trestles. All algorithms were run on the Wikipedia graph. All programs were implemented using Intel® `Cilk++™`. 111
7.2 Performance in terms of *Traversed Edges Per Second* (TEPS) when traversing real-word graphs on machines from (a) Lonestar (12 cores) and (b) Trestles (32 cores). All programs were implemented using Intel® `Cilk++™`. 112
7.3 Performance in terms of TEPS for Wikipedia Graph on (a) Lonestar (12 cores) and (b) Trestles (32 cores). All implementations are in `Cilk++`. 113
7.4 Scalability of implicit and explicit work-stealing algorithms (Cilk Plus implementations) on Xeon Phi/MIC. 115
7.5 Scalability of implicit and explicit work-stealing algorithms (Cilk Plus implementations) on Many Integrated Cores: on inline_1 Graph. 115
7.6 Strong scalability on Intel® Xeon™Phi (a) Wikipedia, (b) RMAT-1M-100M. . 116

8.1 Theoretically optimal work-aware level-synchronous parallel breadth-first search. 121
8.2 An efficient way to find starting point in a list of vertices/edges from where a thread should start working on to get even partitioning of work. 122
8.3 Performance on Xeon (multicores). 123
8.4 Performance on Xeon Phi (manycores). 124
8.5 Strong scalability on Xeon and Xeon Phi. 124
8.6 This figure shows energy consumption by different components of the machine (Package = Socket (CPUs), PP0 = Power Plane0) for the first 3 BFS levels. Here 1 denotes energy consumed at BFS level 1, 1 + 2 denotes energy consumed at levels 1 and 2, and 1 + 2 + 3 denotes energy consumed by level 1, 2 and 3. . . . 124
8.7 Energy and power efficiency on many real-world graphs. 125

9.1 In the Born radius approximation algorithm two octrees are constructed: one for the atoms in the molecule, and the other for the quadrature points. Born radii of all atoms are approximated by recursively traversing both octrees simultaneously. For simplicity, the octrees are drawn as quadtrees. This figure has been reused from [45] with permission. 132
9.2 Octree-based algorithm for r^6-approximation of Born radii. 134
9.3 Octree-based algorithm for approximating E_{pol} from Born radii. 134
9.4 Octree-based distributed- and distributed-shared-memory algorithm. 137
9.5 Strong scalability with increasing number of cores. 140

9.6 Performance comparison of different octree based algorithms (results are sorted by the OCT_{CILK} time). 141
9.7 Performance comparison of different algorithms. Results are sorted by molecule size. 141
9.8 Energy value computed by different algorithms. 142
9.9 Change in performance of the $OCT_{MPI+CILK}$ algorithm with approximation parameter, ϵ; Born Radius ϵ is fixed at 0.9 and E_{pol} ϵ varies. 143
9.11 Speedup w.r.t. Amber when only half of the energy terms are computed. 144

10.1 Ideal pipeline for the automatic generation of efficient recursive algorithms and their implementations. 148

List of Tables

2.1 Block architectural parameters . . . 23
2.2 Performance of edit distance on the spatial architecture and a comparison with a typical modern processor. . . . 25
2.3 Coding effort for edit distance (code footprint). . . . 28

3.1 Complexities of the iterative and recursive divide-and-conquer (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 basecase size $\leq b \times b$. Flexible kernels are shown on yellow background and asymptotically dominating kernels are shown in bold 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. 36
3.2 System specifications. Intel16E is used for power and energy analyses. . . . 43
3.3 Metrics for performance comparison of different implementations. . . . 44
3.4 A Summary of the experimental results. . . . 47

5.1 Timesteps at which each DP table cell is updated ($\mathcal{F}_t$ means function $\mathcal{F}$ updates at timestep t) and the timestep at which each cell becomes fully updated (on the right) for the parenthesis problem on a DP table of size 8×8 using (a) top: standard 2-way recursive algorithm, and (b) bottom: recursive/iterative wavefront algorithm. Both recursive algorithms use a 1×1 base case. We assume that the number of processors is infinite. . . . 71
5.2 System specifications. . . . 77
5.3 Projected scalability of the new recursive wavefront algorithms computed by the Cilkview™Scalability Analyzer. The numbers denote till how many cores the implementation should scale linearly. The input size for LCS was 262144 and for both Parenthesis and FW-APSP, n was 16384. . . . 80

7.1 Naming convention. . . . 102
7.2 Program acronyms. . . . 102
7.3 Simulation environment. . . . 109
7.4 Graphs and their properties. In this table, n and m denote the number of vertices and the number of edges of the graph, respectively. The diameters in the table show the maximum diameters explored by the BFS rather than the actual diameters of these graphs. . . . 110
7.5 Running times of different algorithms (all times are shown in milliseconds). All programs were implemented using `Cilk++`. The minimum running time in each row is highlighted by color. If our algorithms do not achieve the minimum running time, we have highlighted the minimum time for our algorithms in addition to the absolute minimum time in a row. . . . 110
7.6 Statistics of successful and failed steal attempts on the Wikipedia graph when run from 100 sources. For each program we report the average of 5 independent runs. Implementations are in `Cilk++`. . . . 113

7.7 Running time (milliseconds) on Lonestar and Stampede for cilk++, Cilk Plus and OpenMP implementations (performance difference between explicit and implicit work-stealing (recursive divide and conquer)). 114
7.8 Number of vertices explored at a given BFS level. Here n denotes the number of vertices and m denotes the number of edges. 116

8.1 Input graphs and their properties. Here N = #vertices, M = #edges, and MaxD = maximum diameter explored. 122
8.2 System specifications. Intel16E is used for Power and Energy analyses. 123

9.1 Simulation environment. 139
9.2 Packages with GB models and types of parallelism used. 139
9.3 Comparison with Amber GPU implementations on CMV. 144

Acknowledgements

I have to thank Almighty Allah to keep His blessings always on me. Without His blessings and support, I would not be able to come to the point where I am now. Even though my parents cannot be with me, Allah stays always with me to support. My Amma (mother) Monju Ara and Abba (father) Md Jahan Ali deserve thanks and gratitude equally for their unconditional love, support and encouragement throughout my life. While I was seriously thinking about quitting my PhD, they encouraged me to not to quit and helped me to go through the tough time. My sister Sharmin Jahan and brother Jahid Hasan, friend Deep Mondal and auntie Shaheen Ara deserve my wholehearted thanks as well. I want to thank my eight months old nephew Tihami whose beautiful pictures helped me to feel happy when I was stressed. I would like to thank Margaret Munro, uncle and aunt Ashraf, my other uncles and aunts and my late grandparents.

Many thanks to my PhD Supervisor Dr. Rezaul Chowdhury for all his training, guidance, and support throughout the study and without his support it would be impossible to finish this dissertation. No matter how much I thank him, it would not be enough. I want to thank Professor I.V. Ramakrishnan whose fatherly support was always there when I needed it. I would also like to thank Professor Steven Skeina who encouraged me and gave his guidance on innumerable occasions. I thank professor Michael Bender for his guidance on how to improve my presentations. I would like to thank Professor Robert Harrison for agreeing to be on my PhD committee, all his encouragements and the travel grants that he approved for my conference travels.

I would like to thank my mentors Henry Mitchel and Neal Crago for all their support, all my coauthors and collaborators in research, my internship managers and hosts, my lab mates at Theoretical and Experimental Algorithms Lab and all my teachers. I would like to thank all my friends and roommates at Stony Brook. Special thanks to my friend Huck Bennett who helped me with my grammar.

It is impossible to list all the people who encouraged and supported me to reach this point. I would like to thank all of them for their contributions in my life.

Thanks to all.

Jesmin Jahan Tithi

Publications

Conference/Workshop Papers (Refereed)

2016

1. Rezaul A. Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C. Kuszmaul, Charles E. Leiserson, Armando Solar-Lezama, and Yuan Tang, *AUTOGEN: Automatic Discovery of Cache-Oblivious Parallel Recursive Algorithms for Solving Dynamic Programs.*, In Principles and Practice of Parallel Programming (PPoPP), 2016 (to appear) (Authors name in alphabetic order inside an institution).

2015

2. Jesmin Jahan Tithi, Pramod Ganapathi, Aakrati Talati, Sonal Aagarwal and Rezaul Chowdhury, *High-performance energy-efficient recursive dynamic programming using matrix-multiplication-like flexible kernels*, In 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2015.

3. Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi and Rezaul A. Chowdhury, *Cache-Oblivious Wavefront: Improving Parallelism of Recursive Dynamic Programming Algorithms without losing cache-efficiency*, In Principles and Practice of Parallel Programming (PPoPP), 2015.

4. Deukhyun Cha, Qin Zhang, Alexander Rand, Jesmin Jahan Tithi, Rezaul Chowdhury and Chandrajit Bajaj, *Molecular Mechanical and Solvation Energetics on Multicore CPUs and Manycore GPUs*, In the 6th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB), 2015.

2014

5. Jesmin Jahan Tithi, Neal Crago and Joel Emer, *Exploiting Spatial Architectures for Edit Distance Algorithms*, In the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2014 (Best Paper Nominee).

6. Jesmin Jahan Tithi, Neal Crago and Joel Emer, *Exploiting Spatial Architectures for Edit Distance Algorithms*, In Women in High-performance computing (WHPC) at the international Conference for High Performance Computing, Networking, Storage and Analysis (SC14), 2014.

7. Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi and Rezaul A. Chowdhury, *Improving Parallelism of Recursive Stencil Algorithms without Sacrificing Cache Performance*, In Workshop on Stencil Computations (WOSC), 2014.

2013

8. Jesmin Jahan Tithi, Dhruv Matani, Gaurav Menghani and Rezaul A. Chowdhury, *Avoiding Locks and Atomic Instructions in Shared-Memory Parallel BFS Using Optimistic Parallelization*, In 2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW), 2013.

9. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Polarization Energy on a Cluster of Multicores*, In 2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum (IPDPSW), 2013.

10. Rukhsana Yeasmin, Jesmin Jahan Tithi, Jeffrey Chen and Steven Skiena, *Designing Autocorrelated Genes*, In the 4th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB), 2013.

2012

11. I.V. Ramakrishnan, Jesmin Jahan Tithi, Ashish Bagate, Vinayak Khot, Faisal Ahmed, Donald Harrington and Ronak Talati, *Organizing RadLex Lexicon for Efficient Retrieval of Radiology Documents*, In the 2012 ACM SIGHIT International Health Informatics Symposium (IHI), 2012.

Poster (Refereed)

2015

1. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Energy efficiency, Cache adaptivity and Bandwidth benefits of Recursive Dynamic Programming*, In Grace Hopper Celebration of Women in Computing, 2015 (finalist at ACM-SRC).

2. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Efficient Computation of Distance Incorporated Codon Autocorrelation (DICA) Score using Fast Fourier Transform*, In the 6th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB), 2015.

3. Jesmin Jahan Tithi, Yoni Fogel and Rezaul A. Chowdhury, *Two Novel Shared-memory Parallel BFS Algorithms and Their Performance on Multicores and Manycores*, In Women in High-performance computing (WHPC) at the international Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015.

2014

4. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *High-performance Parallel Cache-oblivious Recursive Dynamic Programming with MM-like Flexible Kernels*, In Grace Hopper Celebration of Women in Computing, 2014.

5. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *High-performance Parallel Cache-oblivious Recursive Dynamic Programming with MM-like Flexible Kernels*, In the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM-BCB), 2014.

2013

6. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Polarization Energy on a Cluster-of-Multicores using MPI and cilk++*, In Grace Hopper Celebration of Women in Computing, 2013.

2012

7. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Polarization Energy on Cluster of Multicores*,

In the international Conference for High Performance Computing, Networking, Storage and Analysis (SC12), 2012.

Poster

2015

1. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Energy-efficiency and Cache-adaptivity of Recursive Divide and Conquer Algorithms*, In 2015 IEEE 29th International Symposium on Parallel & Distributed Processing Symposium (IPDPS) PhD Forum & Student Author Program, 2015.

2. Rezaul A. Chowdhury, Pramod Ganapathi, Mohammad M Javanmard, Isha Khanna, Premadurga Kolli and Jesmin Jahan Tithi, *Space-parallelism Tradeoff Under Cache-optimality*, The 12th International Conference and Expo on Emerging Technologies for a Smarter World (CEWIT2015).

2014

3. Jesmin Jahan Tithi, Rezaul A. Chowdhury, Neal Crago and Joel Emer, *Dynamic Programming using Pipelined Parallelism*, In 2014 IEEE 28th International Symposium on Parallel & Distributed Processing Symposium (IPDPS) PhD Forum & Student Author Program, 2014.

2013

4. Jesmin Jahan Tithi, Rezaul A. Chowdhury, Neal Crago and Joel Emer, *Dynamic Programming using Pipelined Parallelism*, The 10th International Conference and Expo on Emerging Technologies for a Smarter World (CEWIT2013).

5. Rukhsana Yeasmin, Jesmin Jahan Tithi, Jeffrey Chen and Steven Skiena, *Designing Autocorrelated Genes*, The 10th International Conference and Expo on Emerging Technologies for a Smarter World (CEWIT2013).

Presentation (Refereed)

1. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Dynamic load balancing on multicores*, Presented in the Early Research Showcase Event of The International Conference for High Performance Computing, Networking, Storage and Analysis, (SC), 2012.

Invited Talk

1. Jesmin Jahan Tithi, *Exploiting Spatial Architectures for Edit Distance Algorithms*, Intel Corporation, Hudson, MA, 13th March, 2014.

2. Jesmin Jahan Tithi and Rezaul A. Chowdhury, *Fast Polarization Energy on Multicores, Clusters of Multicores and GPUs*, Laufer Center Seminar, Stony Brook University, 26th Nov, 2013.

Papers Under Review

1. Rezaul A. Chowdhury, Pramod Ganapathi, Vivek Pradhan, Jesmin Jahan Tithi and Yunpeng Xiao, *A Cache-Efficient Viterbi Algorithm.* (Authors name in alphabetic order)

2. Rezaul A. Chowdhury, Pramod Ganapathi, Yuan Tang and Jesmin Jahan Tithi, *Provably Efficient Scheduling of Cache-Oblivious Wavefront Algorithms.* (Authors name in alphabetic order)

Chapter 1

Introduction

Since the beginning of the last decade, plateauing of the clock speed and per-core performance of computer processors due to the "power wall" constraint[1] has led the CPU manufacturers to leverage multiple cores (often heterogeneous) on a single chip. This plateau and the resulting multicore processors have forced the software community to redesign many algorithms so that they can take advantage of the parallelism to continue software performance scaling. Indeed, almost all machines in today's market have multiple cores, and performance scaling is impossible without exploiting these cores (i.e., without exploiting parallelism). The aim of this dissertation is to present algorithms/algorithmic frameworks that solve several problems in bioinformatics more efficiently than their existing solutions on modern heterogeneous parallel architectures. The contribution lies not only in the novel algorithms designed to solve the specific problems but also in the general algorithmic techniques, which can potentially be used solve many other existing and future problems to gain similar performance benefits.

Why Bioinformatics? Bioinformatics is one of the fastest growing industries which is predicted to be a $13 billion market by 2020 [7]. Bioinformatics finds markets in medical biotechnology (i.e., genomics, chemoinformatics and drug design, proteomics, transcriptomics, metabolomics, and molecular phylogenetics), animal biotechnology, agriculture biotechnology, environmental biotechnology, homeland security and synthetic life science [7]. That is why, it is interesting to design efficient algorithms to solve problems in bioinformatics. Due to the recent changes in computer architectures, faster generation of massive data by the new technology tools and smart devices, and the introduction of big-data analytics, new bioinformatics problems have started to emerge, existing problems have grown in scale, and demands for performance have increased. As a result, traditional computational techniques (e.g., serial sequencing algorithms) are no longer sufficient to solve those new and existing problems. All these have made bioinformatics a field of opportunities for algorithms research.

Challenges. After the realization of the fact that single-core performance scaling is not feasible, computer architectures have changed very rapidly to support performance scaling through parallelism. Changes appeared in terms of computing capability of the processing cores, number of the

[1] "If scaling continues at present pace, by 2005 high-speed processors would have power density of nuclear reactor, by 2010 a rocket nozzle, and by 2015, surface of the Sun" - Former Intel CTO Patric Gelsinger (ISSCC 2001).

processing cores, memory hierarchies and communication paradigms among the processing cores (e.g., shared-memory, distributed-memory, via PCIe bus or through on-chip channels). This rapid change has made programming for high-performance (i.e., developing efficient and scalable software) extremely difficult. At the same time, criteria for high-performance have changed: researchers are more interested in algorithms that are not only faster but also cache- and energy-efficient, make efficient use of space, require less data communication, and are portable. Designing high-performance parallel algorithms that are scalable, cache-, space- and energy-efficient, adaptive and portable is challenging. Nevertheless, it is important to make efforts to achieve that goal. In this dissertation, we show how to leverage several algorithm design techniques and data structures to efficiently solve problems that frequently arise in bioinformatics while achieving many of those performance goals on modern heterogeneous parallel architectures.

Dissertation overview. We explore algorithms under the broad categories of **`algorithms on grids`** (dynamic programming problems [20, 120, 121]), **`algorithms on graphs`** (breadth-first search [120]), and **`algorithms using spatial trees`** (octrees [107]). We target a variety of parallel architectures including multicores, manycores, clusters of multicores, special purpose accelerators (e.g., Triggered Instruction Spatial Architecture [179]), and hybrid combinations of those to develop algorithms. We present efficient parallel algorithms for solving many dynamic programming [14, 44, 49, 172, 179, 182] and graph problems (breadth-first search [178, 181]), and for computing various molecular energetics [40] terms required in molecular dynamics simulations [36, 177] on these target parallel architectures.

While designing an algorithm, our main goal is to achieve high-performance in terms of running time and scalability. However, we still want our algorithms to be energy-efficient, portable, and adaptive to dynamic fluctuations in the availability of shared resources (e.g., cache-space, bandwidth). We want our algorithms to be as resource-oblivious as possible since obliviousness often helps in portability and adaptivity. A *resource-oblivious* algorithm does not use any machine parameter in its algorithm description. Here, machine parameters include number of cache levels, cache size, block transfer size, number of processing cores, number of sockets, bandwidth limitations, energy-budget, structure of the communication network, etc. In the following paragraphs, we discuss more about these performance goals.

Parallel runtime, cache complexity and communication complexity. The parallel running time of a program, $T_p(n)$ is the time it takes to run on p processors, where n is the input parameter. The *work* of a multithreaded program, denoted by $T_1(n)$, is the total number of CPU operations performed when run on a single processor. $T_1(n)$ is also called the serial running time of the program. A parallel algorithm is called work-efficient if its $T_1(n)$ is not asymptotically larger than the work done by the fastest sequential algorithm for the problem. The *span* (also known as the critical-path length), denoted by $T_\infty(n)$, is the maximum number of operations performed on any processor when the program is run on an infinite number of processors. For each of these metrics, the smaller the value, the better it is. Hence, our target is to design algorithms that have small values for $T_1(n)$, $T_p(n)$ and $T_\infty(n)$.

Cache complexity of an algorithm is a performance metric that counts the number of block transfers (or cache misses or I/O transfers or page faults) triggered by a program between adjacent levels of caches in a memory hierarchy. Every cache miss results in a fetching of data from the upper level of cache/s, or RAM or even hard disk which takes a lot of time: up to 50

cycles for L3 cache, close of 80 nanoseconds for RAM, tens of millions of cycles for a hard disk. As a result, an increase in cache complexity may slow down a program. Therefore, our goal is to design algorithms that have low cache complexity.

Communication complexity of an algorithm measures the amount of communication among the processors required to run that algorithm. In case of a shared-memory algorithm, the cache complexity is the same as the communication complexity. For a distributed-memory algorithm, communication complexity measures the amount of data transfer among all participating processors through a physical network. In the case of a multicore CPU connected with a manycore coprocessor (e.g., GPU, Xeon Phi), communication complexity includes both the cache complexity and the cost of data transfer through a PCIe bus that connects the CPU and the coprocessor. Since increased communication complexity often increases running time, for all our algorithms, we make efforts to keep communication complexity as low as possible.

Scalability and parallelism. Scalability is a performance metric that shows how the runtime of a program changes as the number of cores and input size vary. There are two types of scalability metrics that are commonly used in practice: *strong scalability* and *weak scalability*. The strong scalability of a program is measured by keeping the input size fixed while increasing the number of cores. An algorithm has linear strong scalability if the speedup with respect to its serial running time is equal to the number of processing cores used. In general, it is harder to achieve good strong-scalability at larger process counts, since the communication or scheduling overhead of the program starts dominating at that point. The weak scalability shows how the runtime of a program varies with the number of processors when the problem size per processor is fixed. A program has good weak scalability if the run time stays constant while the input size is increased in direct proportion to the number of processors.

The *parallelism* of an algorithm is the average amount of work done per step of the critical path (i.e., $\frac{T_1(n)}{T_\infty(n)}$). Parallelism tells us till how many cores a program should scale in theory. So the higher the value of parallelism, the better it is.

Achieving good scalability and parallelism is one of our primary goals while designing algorithms.

Energy-efficiency. Energy-efficiency of algorithms has become a major concern these days, as the energy costs of running computer applications and equipment have grown to be a major factor of the total US energy expense [12]. The most desirable state, in this case, is to be energy-efficient without any sacrifice in other performance metrics. In general, an algorithm that runs significantly faster also consumes less CPU energy than a comparatively slower one. Similarly, an algorithm that incurs fewer cache misses is likely to consume less DRAM energy, since the latter directly relates to the number of memory accesses that is proportional to the number of last level cache misses. In general, a cache-efficient algorithm runs faster and consumes less energy than a cache-inefficient algorithm, especially on modern architectures with hierarchical caches. Given that future architectures are likely to be energy-limited [35] with deeper cache hierarchies than the existing ones, we need algorithms that are both cache- and energy-efficient.

Portability. Traditionally, an algorithm is considered portable if its same implementation performs reasonably well on different machines with the same basic architecture but different machine parameters (e.g., number of cache levels, block transfer size, overall cache/memory size,

number of cores, NUMA domains, etc.). Having a portable algorithm is very convenient since it saves time to re-design and re-implement.

In this dissertation we classify portability as intra-portability and inter-portability depending on the type of parallelism (e.g., shared-memory, distributed-memory and distributed-shared-memory) that an algorithm exploits. An algorithm has `intra-portability`, if it performs with reasonable efficiency on machines with different machine parameters while exploiting the same type of parallelism (e.g., shared-memory parallelism); without any change in the implementation. We call an algorithm `inter-portable` if it can be easily extended to exploit different types of parallelism offered by different parallel platforms, such as shared-memory (multicores, manycores), distributed-memory and distributed-shared-memory platforms (e.g., a cluster of multicores). In this case, the implementation for each of these parallel platforms is likely to be different, but the basic algorithm remains the same. For example, our research shows that the basic cache-oblivious recursive divide-and-conquer algorithm designed for a shared-memory platform can be easily extended to a distributed-shared-memory platform with reasonable scalability and performance [182]. Often inter-portability is a difficult goal to achieve.

Adaptivity. Like portability, `adaptivity` is a desirable property of an algorithm, which allows it to utilize all available shared resources at any point of time efficiently. Adaptivity is mainly defined in the context of a multiprogramming environment (e.g., typical OS, database, cloud systems, web servers), where multiple independent programs run in parallel and share common resources (e.g., cache-space, memory, bandwidth, processing cores, etc.). An algorithm is called *adaptive* to fluctuations of a particular shared resource, R, if it runs as fast as any other algorithm solving the same problem under the same profile of R. For example, a parallel algorithm is considered to be `cache-adaptive` [21] if it is less sensitive to dynamic fluctuations in shared-memory (and cache), and runs as fast as any other algorithm solving the same problem given any instantaneous memory profile (i.e., the actual size of available memory/cache).

Robustness. We will call an algorithm *robust* if its running time, energy and bandwidth performance remain relatively stable in response to dynamic fluctuations of shared-resources compared to other algorithms solving the same problem under the same resource profile. Performance stability is measured by computing the performance degradation due to fluctuations in resource profile, considering performance with no fluctuation in the total resource capacity as a baseline. It is quite desirable to have algorithms that are robust; because for robust algorithms, the given performance guarantees should hold despite minor anomalies in the system.

In this dissertation, we present parallel algorithms and algorithm design techniques that achieve many of those above mentioned performance goals. That is where the algorithm engineering and high-performance aspects of this dissertation lie. To group algorithms under the same category, we have divided this dissertation into three parts. Part I presents algorithms on grids, Part II presents algorithms on graphs, and Part III covers our work on algorithms using spatial trees. A brief overview of the rest of the dissertation is as follows:

1.1 Part I. Algorithms on grids: Dynamic programming

Part I consists of Chapters 2, 3, 4, 5, and 6, and covers several popular dynamic programming [121] problems encountered in bioinformatics including the edit distance [179] and longest common subsequence problems [44, 173] (used in sequence similarity and alignment), parenthesis problem [84] (used in RNA secondary structure predictions [125, 169]), sequence alignment with general gap penalty [83, 84, 182, 189], Floyd-Warshal's all pairs shortest paths (used in computing transitive closure and phylogeny analysis [144]), protein accordion folding [112] (models folding of alpha-beta sheets) and Viterbi algorithm (used in probabilistic sequence matching, and analysis of long biological sequences [160]). Those are some examples of applications that we are covering by the algorithms presented in this dissertation. In general, solving dynamic programming problems is interesting, since they arise in a wide range of application areas spanning from logistics to bioinformatics [16, 64, 67, 74, 85, 87, 97, 111, 140, 145, 151, 160, 163, 189, 192, 196].

Chapter 2: Dynamic Programming on Spatial Architecture. Chapter 2 shows how the use of special purpose accelerators can boost the performance of an algorithm by many folds if that architecture can efficiently leverage the inherent parallelism in the problem through hardware. We mapped the edit distance algorithm on a proposed triggered instruction spatial architecture [143] that consists of a grid of thousands of small efficient PEs (processing elements). These PEs can directly communicate with other PEs via on-chip network [179] providing the opportunity to convert expensive memory operations to inexpensive (faster and energy-efficient) local PE-to-PE communications. This triggered instruction spatial architecture can exploit the inherent pipeline parallelism in the edit distance algorithm very efficiently, and almost 97% of all memory/cache read and write operations can be converted to local PE-to-PE communication, which eventually translates to 50× better running time and 100× reduction in energy consumption compared to a highly optimized tiled-loop implementation on a standard x86 CPU. Although the algorithm designed for a spatial architecture is not portable, it demonstrates the recent trend of use of special purpose accelerators to boost an application's performance. Solving the sequence alignment or other edit distance like algorithms (i.e., algorithms with local dependencies) on this type of spatial architectures can be a boon for genomics analysis.

Chapter 3: Cache-oblivious recursive divide-and-conquer to solve dynamic programming problems. Chapter 3 shows how to achieve high-performance on standard general-purpose parallel architectures (e.g., multicores, manycores and clusters of multicores) in contrast to special purpose hardware, while solving dynamic programming problems. We show how to obtain high-performing parallel algorithms for a class of dynamic programming (DP) problems by reducing them to highly optimizable *flexible* kernels using a `cache-oblivious recursive divide-and-conquer` technique that reduces an *inflexible* iterative dynamic programming kernel into matrix-multiplication like flexible kernels. A flexible kernel reads from and writes to disjoint regions of a DP table, and hence there is no read/write dependency among the cells being updated. In contrast, for an inflexible kernel the read and write regions overlap. Thus, there are read-write dependencies among the cells being written to, which limit the parallelization and optimization opportunities. The generation of flexible kernels exposes optimization opportunities to make a program high-performing. We solve four non-trivial dynamic programming problems used in bioinformatics, namely the parenthesis problem, Floyd-Warshall's all-pairs shortest path, sequence alignment with general gap penalty and protein accordion folding using cache-oblivious

recursive divide and conquer. These algorithms are $5-150\times$ (resp. $3-30\times$) faster and consume $3-40\times$ (resp. $2-10\times$) less energy[2] than their standard iterative (resp. tiled-loop) counterparts on modern multicores with $16-32$ cores, and have better scalability. Furthermore, these algorithms have both intra- and inter-portability, and can be easily extended to hybrid multicores with manycore coprocessors, and shared-distributed-shared-memory platforms with reasonable practical performance [180, 182]. All our results basically show that cache-oblivious recursive divide and conquer is a very powerful algorithmic tool for solving DP problems in practice.

For convenience, we will use CORDAC to mean **C**ache-**O**blivious **R**ecursive **D**ivide-**a**nd-**C**onquer from now on.

Chapter 4: Adaptivity and robustness of CORDAC algorithms. In Chapter 4 we show cache-adaptivity, bandwidth benefits and robustness of CORDAC algorithms in a multiprogramming environment (e.g., typical operating system, mobile application runtime environment, cloud, etc.). In such an execution environment multiple independent programs can run concurrently which may influence the performance of those programs adversely by reducing available shared resources otherwise available in a dedicated execution environment where only one program is run at a time. We show that due to its cache-efficiency, cache-obliviousness, and recursive nature, a CORDAC algorithm is more adaptive to dynamic changes in the availability of shared caches compared to its tiled-loop and standard iterative counterparts. CORDAC algorithms are less sensitive to memory and bandwidth fluctuations, too. Furthermore, the running time, energy and bandwidth performance of CORDAC algorithms remain more stable than the corresponding tiled-loop and iterative algorithms during dynamic fluctuations of shared resources (i.e., robustness property). Understanding the relationships among cache-obliviousness, cache-optimality, cache-adaptivity, energy-consumption and bandwidth utilization of different algorithmic options (iterative, tiled-loop and recursive divide-and-conquer) for DP problems is very important in deciding which algorithm to choose in practice. To the best of our knowledge, we present the first empirical results to unravel some of those relationships in a multiprogramming setting by demonstrating adaptivity and robustness of CORDAC algorithms.

Chapter 5: Cache-oblivious wavefront algorithms. The standard cache-oblivious recursive divide-and-conquer, i.e., CORDAC algorithms for dynamic programming problems often have *artificial dependencies* [173] that may reduce their parallelism asymptotically. However, these artificial dependencies can be removed using a cache-oblivious wavefront technique [173]. In Chapter 5 we show how to systematically transform a CORDAC algorithm into a cache-oblivious wavefront algorithm by removing artificial dependencies among the tasks through appropriate scheduling. Our transformed algorithms achieve optimal parallel cache-complexity and high parallelism with negligible implementation overhead. We use closed-form formulas to compute at what time each divide-and-conquer function must be launched in order to achieve high parallelism without losing cache performance. We present experimental performance and scalability results showing the superiority of these new algorithms over existing algorithms for the longest common subsequence, Floyd-Warshall's all pairs shortest path, and parenthesis problems. Results in this Chapter show that CORDAC with improved parallelism (i.e., the cache-oblivious wavefront

[2] We have observed that the CPU energy ratio closely matches with the runtime ratio if we use the same machine and same input size in the experiment. However, due to limitations in the software tools used to measure the energy, the runtime and energy experiments were conducted on two different architectures with different input sizes. Hence, the ratios were not the same.

algorithms) will be useful for future architectures with many more cores than the state-of-the-art multicore machines.

Chapter 6: Cache-efficient CORDAC algorithm to solve the Viterbi problem. The Viterbi algorithm is used to find the most likely path through a Hidden Markov Model (HMM) given an observed sequence. This algorithm has numerous applications in bioinformatics spanning multiple sequence alignment [145], gene finding [32], CG island [67] and conserved elements detection [160], and protein secondary structure prediction [115]. Chapter 6 shows how the *Viterbi problem* (a problem that the Viterbi algorithm solves) can be solved cache-efficiently using a CORDAC technique by exploiting the rank convergence property of the problem [126]. Apart from its importance, solving the Viterbi problem is interesting because it has an irregular data access pattern and a space-compute ratio of 1 (per computation data access is $\omega(1)$). Both of these make designing a cache-efficient algorithm to solve this problem challenging. We present two algorithms to solve single-instance and multiple-instances of the Viterbi problem along with performance analyses and comparative results with the existing fastest algorithm showing the superiority of our approach [49].

One major contribution from this Part. This dissertation shows that cache-oblivious recursive divide and conquer is a very powerful algorithmic tool for solving non-trivial dynamic programming problems. So far, CORDAC algorithms for dynamic programming problems were mainly used by researchers in theoretical settings but was not adopted by general scientists and programmers, partly because they are difficult to develop, program and optimize (due to their complicated dependency structure). Another popular misconception about CORDAC algorithms is that although they have good theoretical bounds, in practice they are not high-performing. In this dissertation, we solve many non-trivial dynamic programming problems using CORDAC technique and show how to optimize them in a very systematic way to outperform traditional parallel iterative or tiled-loop algorithms by orders of magnitude. We show that these CORDAC algorithms are not only faster, scalable, portable, cache-, energy- and bandwidth-efficient, but also posses properties such as adaptivity and robustness, that other types of algorithms for solving dynamic programming problems do not have. Furthermore, we show how to improve parallelism of the CORDAC algorithms using cache-oblivious wavefront technique while retaining the cache-optimality. Our results make a very strong point in favor of using CORDAC algorithms in practice, especially in a multiprogramming environment such as standard operating system, mobile application runtime systems, database management system, systems that support multiple virtual machines simultaneously as well as the cloud. We believe that our research results will encourage scientists and programmers to adopt these algorithms and algorithmic techniques to solve their problems in practice.

1.2 Part II. Algorithms on graphs: Breadth-first search

Part II contains Chapter 7 and 8, and describes several level-synchronous parallel breadth-first search (BFS) graph traversal algorithms on shared-memory architectures. Breadth-first search has numerous applications in bioinformatics including analyses of biological interaction networks, metabolic pathway search, finding minimum gene subsets, betweenness centrality [80] and searching in tries [76]. Therefore, performing breadth first search efficiently is important.

Chapter 7: In Chapter 7 we show how to avoid locks and atomic instructions during dynamic load balancing using optimistic parallelization for shared-memory parallel level-synchronous BFS algorithms [178]. Use of locks and atomic instructions makes programs non-scalable due to serialization. We show that a lock and atomic-instruction free parallel BFS algorithm eventually does better load balancing, has improved parallelism, and as a result runs faster than the corresponding lock-based algorithm. We present algorithms based on recursive divide and conquer, centralized vertex queue, and randomized work-stealing on distributed queues. We derive theoretical performance bounds and prove correctness of our algorithms. We also present experimental results showing scalability of our algorithms on state-of-the-art multicore and manycore (Xeon Phi) machines, using different parallel programming platforms (cilk++™, cilk™plus, OpenMP), and on various types of graphs, demonstrating portability of these algorithms. We compare our algorithms with two other contemporary BFS algorithms by Hong et. al. (PACT, 2011) and Leiserson et. al. (SPAA, 2010), and show that our algorithms perform better than both of these benchmarks.

Chapter 8: In Chapter 8 we present a work-aware parallel level-synchronous BFS algorithm for shared-memory architectures which achieves the theoretical lower bound on parallel running time by using an optimal number of processing cores at each computation step. We also analyze energy performance of this algorithm.

1.3 Part III. Algorithms using spatial trees: Molecular energetics

In Part III, we describe algorithms that use spatial trees. Spatial trees are a class of recursive space partitioning data structure that can help to organize high-dimensional data. The R-tree [98], quad-tree [75], octree [33] and k-D tree [22] are well-known and widely used spatial trees. Spatial trees have been used in implementing molecular docking programs [41], molecular dynamics simulations [43], spatial phylogeny reconstructions, clustering and in medical imaging [165]. In our research we have used one such tree called *octree* that is cache-friendly, recursive, and uses only linear-space to represent biological (protein) molecules.

Chapter 9: In Chapter 9 we present hybrid distributed and distributed-shared memory parallel algorithms for computation of molecular polarization energy [176, 177]. These algorithms use octree-based Greengard-Rokhlin-type near-far approximation and are built on top of a shared-memory cache-oblivious recursive divide-and-conquer algorithm. This demonstrates the inter-portability of a recursive divide-and-conquer, i.e., CORDAC algorithm to different parallel programming platforms (i.e., shared-memory to distributed- and distributed-shared-memory). Using two levels of approximations (numerical and algorithmic), cache-friendly recursive octree data structure, and efficient hybrid load balancing strategy, our algorithms achieve $\sim 400\times$ speedup w.r.t Amber [57] (a popular Molecular Dynamics package) with less than 1% error w.r.t. the naïve exact algorithm, using as few as 144 cores (i.e., 12 compute nodes with 12 cores each) for molecules with as many as half a million of atoms.

We conclude in Chapter 10 by discussing extensions of our work, open problems and future research possibilities.

Part I

Algorithms on Grids

Chapter 2

Exploiting Spatial Architectures for Edit Distance Algorithms

2.1 Abstract

In this research, we demonstrate the ability of a spatial architecture to significantly improve runtime performance and energy efficiency of *edit distance*, a broadly used dynamic programming algorithm in bioinformatics. Spatial architectures are an emerging class of application accelerators that consist of a network of many small and efficient processing elements, and can be exploited by a large domain of applications. In this research, we show that spatial architecture is a good fit for edit distance algorithms which can efficiently map the dataflow characteristics and inherent pipeline parallelism within edit distance to develop efficient and scalable implementations.

We evaluate our edit distance implementations using a cycle-accurate performance and physical design model of a previously proposed triggered instruction-based spatial architecture in order to compare against real performance and power measurements on an x86 processor. We show that when chip area is normalized between the two platforms, it is possible to get more than a 50× runtime performance improvement and over 100× reduction in energy consumption compared to an optimized and vectorized x86 implementation.[1] This dramatic improvement comes from leveraging the massive parallelism available in spatial architectures and from the dramatic reduction of expensive memory accesses through conversion to relatively inexpensive local communication.

2.2 Introduction

There is a continuing demand in many application domains for increased levels of performance and energy efficiency. While the number of transistors is expected to continue to scale with

[1] The energy profiling has been done by Neal Crago from Intel Corporation. He also optimized the x86 implementation. This work was done during my internship at Intel and was managed by Emer Joel.

Moore's law for at least the next five years, the "power wall" has dramatically slowed single-core processor performance scaling. Recently, several accelerator architectures have emerged to further improve performance and energy efficiency over multi-core processors in specific application domains. These architectures are tailored using the properties inherently found in these domains, and can range in programmability. Perhaps the best-known examples are fixed-function accelerators, which are tailored to single algorithms such as video decoding, and GPUs, which are fully programmable and target data parallel and SIMT-amenable code.

Spatially programmed architectures are an emerging class of programmable accelerators that target application domains with workloads whose best-known implementation involves asynchronous actors performing different tasks while frequently communicating with neighboring actors. The application domains that spatially-programmed architectures target spans a number of important areas such as signal processing, media codes, cryptography, compression, pattern matching, and sorting.

Spatially programmed architectures are typically made up of hundreds of small processing elements (PEs) connected together via an on-chip network. When an algorithm is mapped onto a spatial architecture, the algorithm's dataflow graph is broken into regions, which are connected by producer-consumer relationships. A set of PEs from the spatial architecture is then assigned to execute a particular region. Input data is then streamed through this pipelined set of regions.

Edit distance is a broad class of algorithms that find use in many important applications, spanning domains such as bioinformatics, data mining, text and data processing, natural language processing, and speech recognition. The edit distance problem determines the minimum number of "non-match" data edits to convert a source string or data object S to a target string or data object T. The algorithm also keeps track of the specific data edits required to convert S to T, from a set of four possible data edits: insertion, deletion, match or substitution. For example, if S = "computer" and T = "commute", the minimum number of "non-match" edits to convert S to T is 2 and the edits are, $MMMSMMMD$ (where M = Match, S = Substitute, D = Delete).

The edit distance problem is ripe for acceleration, as the dynamic programming techniques typically used to solve the problem take $O(nm)$ time and space, where m and n are the lengths of the strings S and T respectively. However, the nature of data dependencies within the algorithm makes vectorization and parallelization non-trivial on modern CPUs and GPUs. On the other hand, these same data dependencies have very nice dataflow properties which are quite naturally mapped on spatial architectures using pipeline parallelism [25, 117]. Moreover, the exploitation of pipeline parallelism also enables the conversion of many memory references into much less expensive local PE-to-PE (PE = Processing Element) communication which further improves efficiency.

In this chapter, we first build intuition on why spatial architectures are a good fit for edit distance and similar algorithms with local dependencies. We then describe three different algorithmic implementations of edit distance tailored to spatial architectures. We evaluate these implementations by conducting a detailed experimental analysis of performance and energy consumption on a triggered instruction-based spatial architecture [143]. Finally, we compare the performance and energy consumption of our implementations to highly optimized and vectorized x86 implementations.

2.3 Background

2.3.1 Spatial Architectures

In the spatial programming paradigm, an algorithm's dataflow graph is broken up into regions so that it can be represented as a pipeline of computation. Sets of independent regions act as stages within the pipeline, with producer-consumer relationships between stages. Ideally, the number of operations in each region is kept small, as performance is usually determined by the *rate-limiting step*. Figure 2.1 presents an example dataflow graph and its corresponding representation in the spatial programming paradigm. In this example, each region is made up of three nodes from the original dataflow graph, and the total number of pipeline stages is two. Note that in Stage 0, two regions are independent and are executed in parallel. After the pipeline is generated, the input dataset can be streamed through the pipeline and the inherent pipeline parallelism can be exploited.

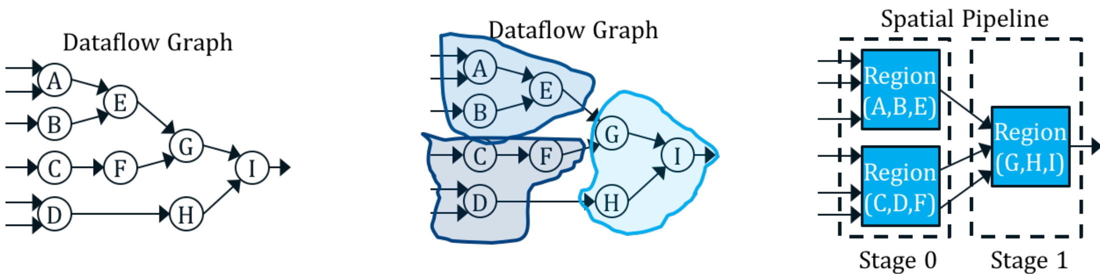


Figure 2.1: *Spatial programming example. Converting a dataflow graph to a spatial pipeline of regions.*

While the pipeline can, in theory, be mapped to general-purpose processors, executing the algorithm on the appropriate accelerator architecture can provide significant benefits. Accelerator architectures execute alongside general-purpose processors with the end goal of improving the performance and energy consumption of a selected set of algorithms and application domains. Similar to how vector engines and GPUs are chosen to accelerate many vectorizable algorithms, spatial architectures are chosen to accelerate algorithms amenable to spatial programming. Spatial architectures are a computational fabric of hundreds or thousands of small processing elements (PEs) directly connected together with an on-chip network. The algorithm's pipeline is successfully mapped onto a spatial architecture by utilizing some number of PEs to implement each region of the dataflow graph, and then by connecting the regions using the on-chip network. As performance depends on minimizing the execution time of each pipeline stage, the regions are typically sized as small as possible, with the algorithm utilizing all of the available PEs.

Spatial architectures broadly fall into two categories, primarily based upon the basic unit of computation: logic-grained and coarse-grained. Field-programmable gate arrays (FPGAs) are among the most well known logic-grained spatial architectures and are most commonly used for ASIC prototyping and as stand-alone general logic accelerators. FPGAs are designed to emulate a broad range of logic circuits and use very fine-grained lookup tables (LUTs) as their primary unit of computation [53, 128]. More complex logical operations are constructed by connecting many LUTs together using the on-chip network. While the use of fine-grained LUTs results in a high degree of generality, this generality results in much lower clock speeds for mapped algorithms when compared with ASIC implementations. In general, FPGAs and logic-grained

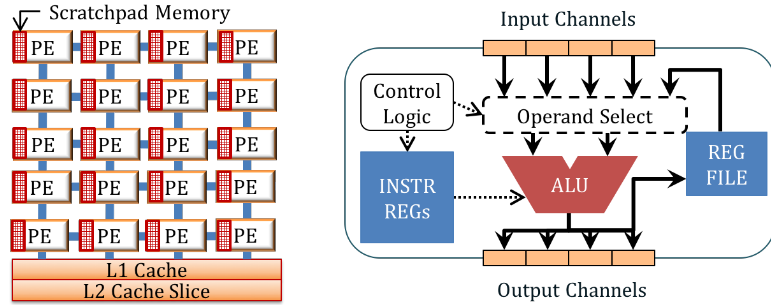


FIGURE 2.2: *Example spatial architecture. Network of PEs, scratchpad memory, and caches are shown alongside a PE diagram.*

spatial architectures sacrifice compute density for complete bit-level generality. It is also well known that the programming environment for FPGAs is particularly complex. FPGAs typically use a low-level programming model (e.g. VHDL or Verilog) due to being used primarily for ASIC prototyping. Additionally, the fine-grained nature of the LUTs creates a large solution search space for place and route algorithms, which can lead to unacceptably long compilation times.

However, a common observation is that many algorithms primarily utilize byte- or word-level primitive operations, which are not efficiently mapped to bit-level logic and logic-grained spatial architectures such as FPGAs. To partially address these inefficiencies, some FPGAs now provide digital-signal processing datapaths alongside the traditional LUTs. In contrast to FPGAs, coarse-grained spatial architectures are designed from the ground up to suit the properties of these algorithms. Coarse-grained spatial architectures optimize byte- and word-level primitive operations into hardened logic and through the utilization of ALUs within PE datapaths [99, 130, 132]. The hardened logic results in much higher compute density, which leads to faster clock speeds and reduces compilation times substantially compared to FPGAs. Generally speaking, these coarse-grained spatial architectures have a higher-level programming abstraction, which typically includes some notion of an instruction set architecture. In other words, PEs can be programmed by writing a sequence of software instructions, rather than requiring the hardware-level programming of an FPGA. The distinct advantages and large possible design space of coarse-grained spatial architectures have resulted in a significant amount of recent research. Specifically, there has been research into evaluating architectures, control schemes, and levels of integration with host processor cores [91, 143, 159, 168].

Given the clear benefits of coarse-grained compared to logic-grained, in this paper, we focus on implementing edit distance on coarse-grained spatial architectures. Figure 2.2 presents the high-level architecture and PE-level architecture of the coarse-grained spatial architecture we consider. The architecture consists of a collection of PEs, scratchpad memory, a cache hierarchy, and an on-chip network. For our architecture, each PE has some control logic, an instruction memory, a register file, an ALU, and some number of input and output connections to an on-chip network. To further support high compute density and provide efficiency, the instruction memory and register file within the PE are kept quite small, and the complexity of the ALU is kept low. PEs connect to each other, scratchpad memory, and the cache hierarchy using the on-chip network.

2.3.2 Edit Distance

In this section, we describe the edit distance problem and explain what makes it amenable to pipeline parallelism and spatial architectures. The edit distance problem is defined as finding the minimum edit cost to convert one string or data object into another string or data object. Solving the edit distance problem is interesting because of its prevalence in important application domains including bioinformatics, data mining, text and data processing, natural language processing, and speech recognition. In addition to those domains, achieving better performance and energy efficiency on edit distance through exploiting spatial architectures can also provide insight into how other applications with similar local dependencies might benefit when mapped to spatial architectures. Such domains include dynamic programming problems with local dependencies (e.g., longest common subsequence, Smith-Waterman and Needleman-Wunch algorithms), virus scanners, security kernels, stencil computations, and financial engineering kernels.

2.3.2.1 Overview of the Edit Distance Problem

$$ED(S[1:i], T[1:j]) =$$
$$\begin{cases} 0 \text{ if } i = j = 0, \\ CostOfInsert(T[1:j]) \text{ if } i = 0,\ 1 \leq j \leq n, \\ CostOfDelete(S[1:i]) \text{ if } j = 0,\ 1 \leq i \leq m, \\ \min \left\{ \begin{array}{c} MatchOrSub(S[i], T[j]) + ED(S[1:i-1], T[1:j-1]), \\ CostOfInsert(T[j]) + ED(S[1:i], T[1:j-1]), \\ CostOfDelete(S[i]) + ED(S[1:i-1], T[1:j]) \end{array} \right\} \\ \text{if } i, j > 0. \end{cases} \qquad \text{(Recurrence 1)}$$

Recurrence 1 solves the edit distance problem. The edit distance for converting S of length i to T of length j can be computed by taking the minimum of solutions to three smaller sub-problems. The first sub-problem is to *match or substitute* $S[i]$ with $T[j]$, and then recursively find the edit distance of converting S of length $i-1$ to T of length $j-1$. The second sub-problem is to *insert* the last character of T ($T[j]$) at the end of S, and then recursively find the edit distance of converting S of length i to T of length $j-1$. The third sub-problem is to *delete* the last character of S ($S[i]$), and then recursively find the edit distance of converting S of length $i-1$ to T of length j.

T[1:n] (columns), S[1:m] (rows)

	""	s	p	o	r	t
""	0	1, I	2, I	3. I	4, I	5, I
s	1, D	0, M	1, I	2, I	3, I	4, I
o	2, D	1, D	1, S	1, M	2, I	3, I
r	3, D	2, D	2, S	2, S	1, M	2, I
t	4, D	3, D	3, S	3, S	2, D	1, M

FIGURE 2.3: *Solving for edit distance using dynamic programming. The dark-shaded cells are the edits for solving the base cases, and the light-shaded cells are the required minimum edits.*

The costs of *match, substitute, delete* and *insert* are user-defined and can vary depending upon the application. In the most general edit distance problem, all costs are assumed to be the same (typically 1, except a cost of 0 for a match). In the rest of the Chapter, we assume that the cost of a single insertion, deletion, and substitution is 1. The three base cases for Recurrence 1 are as follows:

- ▷ Converting a string S of length 0 to another string T of length 0 is a cost of 0.
- ▷ To convert a string S of length 0 to any string T of any length j, we *insert* all j characters of T which incurs a total cost of the sum of inserting all characters from T of length j.

▹ To convert a string S of length i to a string T of length 0, we *delete* all characters from S which incurs a total cost of the sum of deleting all characters from S of length i.

It is inefficient to use recursion to solve the edit distance problem due to the large number of sub-problem re-computation required. Therefore, dynamic programming principles are typically used to solve the problem in a bottom-up manner. The dynamic programming approach saves the result of each sub-problem in a table, enabling reuse rather than requiring recomputation. To find the required number of edits to convert a source string S to the target string T, a two-dimensional $m \times n$ cost matrix "M" is allocated, where m and n are the lengths of the two strings S and T, respectively. Each cell of the matrix $M(i, j)$ gives us the minimum number of edits to convert $S[1 : i]$ to $T[1 : j]$. We first populate the matrix with the base case solutions and then compute the remaining cells row by row following the same recursive formula. Figure 2.3 shows the filled out cost matrix after executing the dynamic programming algorithm on S = "*sort*" and T = "*sport*" where cell $M[m][n] = M[4][5]$ gives the final edit distance.

2.3.2.2 Exploitation of Pipeline Parallelism

In the dynamic programming approach to the edit distance problem, the matrix cells have local dependencies. Figure 2.4 presents the data dependencies in calculating a single cell $M[i][j]$. Observe that the value of a cell $M[i][j]$ depends on its *top cell* ($M[i-1][j]$), *left cell* ($M[i][j-1]$) and *diagonal cell* ($M[i-1][j-1]$). Because of these dependencies, this computation is not vectorizable along the row or the column of the $2D$ cost matrix. While it is possible to vectorize along the diagonal by reshaping the cost matrix into a diamond, such an implementation requires dummy computations and frequent communication between cells must still be facilitated using expensive memory accesses. Conversely, the data dependencies found in edit distance naturally compose into pipeline parallelism: values produced by a worker responsible for computing a cell of the cost matrix can be consumed by workers computing adjacent cells. It is possible to get very efficient cell-level parallelism for edit distance in a spatial architecture because spatial architectures have the benefit of small efficient PEs and direct PE-to-PE communication capability. Note that cell-level parallelism is not feasible on multicore machines at all because of the prohibitive overhead of communication and scheduling.

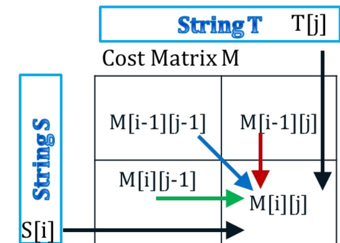


FIGURE 2.4: *Data flow dependencies for calculating a cell in edit distance. Dependencies are found along the row and column of the cost matrix, which limits the ability to vectorize.*

2.4 Edit Distance on Spatial Architectures

In this section, we discuss how the edit distance problem maps down to spatial architectures. We start by describing the basic unit of computation, a worker, and utilize that worker to develop an initial naïve implementation. We analyze that naïve implementation and describe several possible optimizations. We also explain how a spatial implementation of edit distance makes more efficient use of memory.

2.4.1 Designing a Worker

To implement edit distance, we first create a core module that incorporates the data flow and state transitions needed to compute the value of a single cell of the cost matrix M. Part of this process is deciding which inputs and outputs are required for a cell computation and the relationship between the inputs and outputs of a cell with its neighboring cells.

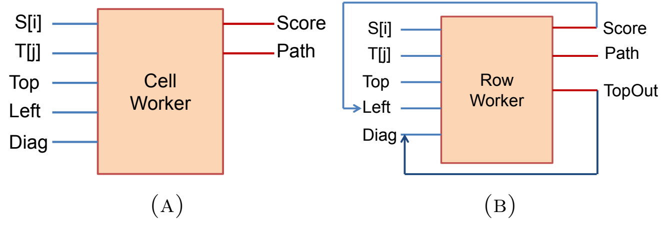


FIGURE 2.5: *(A) A simple cell worker that computes a single cell of the cost matrix, (B) An optimized row worker.*

Figure 2.5(a) depicts an abstract worker which implements the core module and its inputs and outputs. As shown in Figure 2.4, the *score* of a cell $M[i][j]$ of the cost matrix depends primarily on its *top* ($M[i-1][j]$), *left* ($M[i][j-1]$) and *diagonal* ($M[i-1][j-1]$) cells, as well as the string characters $S[i]$ and $T[j]$, where $S[i]$ and $T[j]$ is the ith character of S and j^{th} character of T, respectively. The score of the current cell $M[i][j]$ is determined by calculating the minimum of *insert cost* (*left* + CostOfInsert($T[j]$)), *delete cost* (*top* + CostOfDelete($S[i]$)) and *match/-substitution cost* (*diagonal* + MatchOrSub($S[i]$, $T[j]$)). The path chosen for each cell, i.e., the edit that resulted in the minimum score, can also be stored in a separate *path* array, where *path*$[i][j]$ = (*delete, insert, match/substitute*). The data stored in the *path* array can later be used to reconstruct the actual edits used to convert S to T in linear time. Therefore, in this initial approach we need 5 memory reads ($S[i]$, $T[j]$, Top, $Left$, $Diagonal$), 2 memory writes (score, path), 3 additions and 3 subtractions to compute the value for each cell.

Figure 2.6 shows simplified pseudocode for a single cell computation of $M[i][j]$. First, the insert, delete, and match/substitute costs are computed using values from $M[][]$, $S[i]$, and $T[j]$. Next, the minimum between the three costs is chosen, returning both the score and the path values. Finally, the score and path values are written out to memory. In practice, we find that we can split this core module into two other smaller modules, each of which can be mapped onto a PE, in order to reduce the length of the critical path. We define a *worker* as the unit of these two PEs that implements the core module. A collection of these workers is used to compute the score values of the entire cost matrix.

```
CalculateCell( row i, column j)
{
   insert_cost = M[ i ][ j-1 ] + CostOfInsert (T[ j ])
   delete_cost = M[ i-1 ][ j ] + CostOfDelete (S[ i ])
   match_substitute_cost = M[ i-1 ][ j-1 ] + MatchOrSub (S[ i ], T[ j ])
   [score, path] = Min (insert_cost, delete_cost, match_substitute_cost)
   M[ i ][ j ] = score;
   Path[ i ][ j ] = path;
}
```

FIGURE 2.6: *Pseudocode for the calculation of a single cell.*

2.4.1.1 Optimization

One possible implementation of edit distance on a spatial architecture would be to use distinct workers to calculate the value of each cell $M[i][j]$. However, the scalability of such an approach to large problem sizes is quite limited considering that the requirement for $O(mn)$ workers would require at least $O(mn)$ PEs. Spatial architectures have finite physical resources by definition,

and thus, a scalable implementation needs to be able to map to those finite resources regardless of m and n. Therefore, we need to find an alternative solution to compute all the mn cells using only a limited number of workers, w. For example, if we consider assigning one worker to compute all the cells of a row of the cost matrix, we can observe the following patterns in the inputs and outputs of that *row worker* (see Figure 2.5(b)):

- ▷ *Top* ($M[i-1][j]$) at current cell position $M[i][j]$ becomes the *diagonal* for the next cell $M[i][j+1]$.
- ▷ The current computed score $M[i][j]$ becomes $left$ for the next cell $M[i][j+1]$.
- ▷ $S[i]$ needs to be read only once from memory for the entire i^{th} row.
- ▷ $T[j]$ and *top* need to be read for each cell from memory.

Therefore, if we can reuse the values already read from memory and produced by the same row worker, that will save $O(3n)$ memory reads *($S[i]$, left, diagonal)* for each row (i.e., saves around $3mn$ reads out of $5mn$ reads in total).

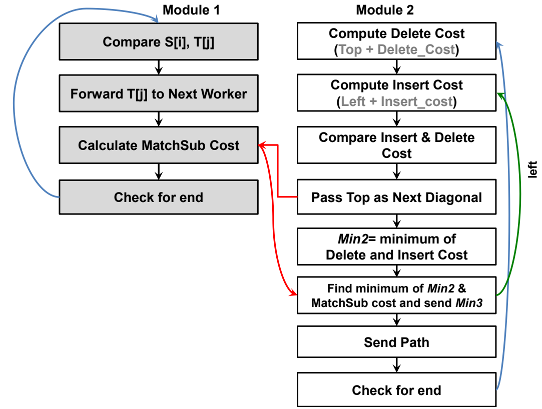


FIGURE 2.7: *Flow chart for the control flow path of a single worker. Blue arrows show state transitions, green arrows show self-feedback, and red arrows show communication with other PEs.*

Figure 2.7 presents a flow chart for the control path of the two row worker modules mapped to PEs. First, the delete and insert costs are computed and the minimum between the two is chosen, while in parallel the match-substitute cost is also computed and communicated. Then the minimum of the three costs is chosen, and the final score and path values are determined. Observe the feedback loops added to the module to reuse the top as diagonal and current score as left.

If we consider two consecutive row workers working on two consecutive rows of the score or cost matrix M, it is possible to observe some further memory access optimizations. A row worker working on the ith row can send its current computed score, $M[i][j]$ as the *top* value to the row worker working on the $(i+1)^{th}$ row. Similarly, $T[j]$ can also be reused by the row worker working on the jth column of the score matrix by propagating $T[j]$ using the local PE-to-PE communication channel. In fact if we have more row workers working on consecutive rows, once a character $T[j]$ has been read from memory by the first row worker, it can forward $T[j]$ to the second row worker, second row worker can forward it to the third row worker, and so on. Therefore, if we use w row workers to compute w consecutive rows of the cost matrix, any row worker k for $1 < k \leq w$, does not need to read $T[j]$ from memory. Similarly, each kth row worker $(1 < k \leq w)$ receives its *top* value from row worker $k-1$'s computed scores. Hence, only the first row worker of a strip of w rows (*strip: w consecutive rows of the cost matrix*) needs to read the *top* and $T[j]$ values from memory. Other workers can get them from prior worker through the PE channels. As a result, nearly all memory read operations remaining can be removed and converted into less expensive local PE-to-PE communication, saving both memory bandwidth and energy consumption.

2.4.1.2 Mapping

Note that the row workers here proceed as a diagonal wavefront. For example, when row worker k works on cell $M[i][j]$, row worker $k+1$ works on cell $M[i+1][j-1]$ and row worker $k+2$ works on cell $M[i+2][j-2]$ and so on, in a pipelined manner. Figure 2.8 shows the interconnection between two consecutive row workers and a possible serpentine layout of the row workers on a spatial architecture made with a grid of PEs where row worker W_k receives input from row worker W_{k-1} and sends output to row worker W_{k+1}.

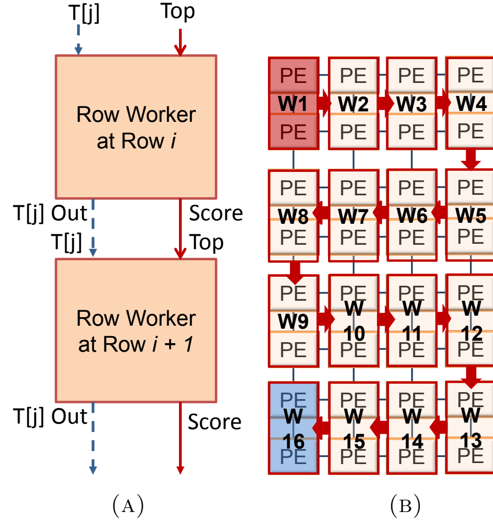


FIGURE 2.8: *(A) Two row workers working on two consecutive rows and connected together using communication channels. (B) A possible layout of the row workers on a grid of PEs.*

Based on these observations and the resulting optimized row worker (Figure 2.8), we have designed three different algorithms to solve the edit distance problem, namely: naïve, strip mining, and tiling. For each of these algorithms, we primarily focus on computing the score, as prior work has shown that the edits can be reconstructed by recomputing the required subsections of the cost matrix while tracing in the backward direction [46, 47, 103, 137]. Hence, showing that our algorithms do better in computing the score should mean that they will also perform better in computing the edits. In each of these algorithms we use w row workers to compute the scores (and paths) of the first w consecutive rows from 1 to w, and then compute rows from $1+w$ to $1+2w$, and so on until the entire cost matrix has been computed. Note that string S is padded (and T for the tiled version), as needed to make the length divisible by w (or tile height, d).

2.4.2 Naïve Implementation

In the naïve approach, we use $O(mn)$ memory space to store the scores of the cost matrix (and path), and each row worker stores the score (and path) value to memory for each cell it computes. We connect a row worker's output to its own input and to other row workers as shown in Figures 2.5(b) and 2.8. These optimizations remove most of the memory reads otherwise required. However, $O(mn)$ memory writes are still required to store the $O(mn)$ score values for all cells in the resulting cost matrix.

2.4.3 Optimization: Use of Linear Memory Space

"Quadratic space kills before quadratic time". In the standard edit distance problem, the two-dimensional cost matrix M consumes $O(mn)$ memory space. However, this quadratic use of memory space becomes infeasible for large strings. Fortunately, it is possible to use linear

memory space ($O(n)$) to store the cost matrix. Observe that for edit distance, the resulting output data is the final cost of converting S to T which can be found in cell $M[m][n]$. Therefore, we do not need to maintain data storage for other cells when they are no longer actively being used. To compute the score/cost for the ith row, we only need the $(i-1)^{th}$ row as input. Rows before $(i-1)$ can be forgotten. Therefore, it is possible to use a cost matrix of linear size $n+1$, from which the first row worker (from a set of w row workers) reads its *top* inputs, and the last row worker writes its computed cost values which can be used as input for the next set of rows (computed using the same w row workers). We use this linear memory space optimization in our strip mining and tiling based algorithms.

Although it is sufficient to use linear memory space ($O(n)$) to store the cost matrix, use of a two-dimensional cost matrix can aid in the reconstruction of actual edits/path in linear time. It is also possible to use a separate two-dimensional path matrix while using linear memory space for the cost matrix, which allows linear time reconstruction of edits. Finally, it is also possible to use only linear memory space for the cost matrix, and reconstruct the edits in quadratic time without saving any path matrix [46, 47] which is discussed later in this section.

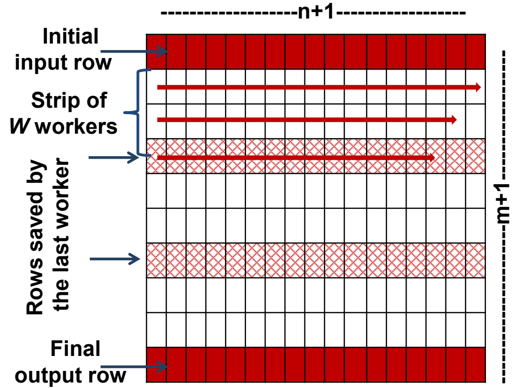


FIGURE 2.9: *Strip mining using standard memory and scratchpad memory. Initial and final rows are read from and written to memory, while the intermediate rows use either memory or scratchpad memory.*

2.4.4 Strip Mining

The *strip mining* technique involves computing the two dimensional cost matrix in a strip-by-strip manner (*strips* of w consecutive rows from the cost matrix). In this approach, we virtually divide the two-dimensional cost matrix into strips of size $w \times n$, where w denote the number of row workers and n denote the width of the cost matrix. In the strip mining approach, we use a linear cost matrix array of size $n+1$, from which the first row worker reads its *top* inputs and the last row worker writes its computed cost values which are used as input for the next strip. As before, we use the optimized row worker for this algorithm. We use two different strategies for the strip mining algorithm as described below:

2.4.4.1 Strip Mining using Memory

In the strip mining using memory algorithm, only the first row worker of a strip of size w reads the score values from cost matrix and string T from memory, and only the last row worker stores the computed score values to memory (Figure 2.9). This organization reduces the total number of memory writes to the cost matrix to $O\left(\frac{m}{w}n\right)$ from $O(mn)$ in addition to the reduction in memory reads as described before. Furthermore, the number of memory reads of string T as well as the cost matrix reduces to $O\left(\frac{m}{w}n\right)$. To optimize this approach further, each row worker starts computing from the 0th column instead of the 1st column of the cost matrix. Note that in a typical edit distance algorithm, computation starts from the 1st column while using the 0th column as input. If we had started computing from the 1st column, we would need to read the

$left$ and *diagonal* cells from memory. However, if we start from the 0th column, we can use the *top* value to compute the $left$ and *diagonal* by adding 1 to the *top* value (which comes from memory for the first row worker and from the previous row worker for all other row workers). This approach reduces the number of memory reads by $2m$. Figure 2.9 shows how the strip mining algorithm works. The hash-patterned cells in the cost matrix are stored in memory by the last row worker and read by the first row worker (in total $\frac{m}{w}$ times), while the white colored cells are not stored to memory and are instead communicated directly between the row workers.

2.4.4.2 Strip Mining using PEs' scratchpad memory

Note that memory accesses can be expensive and involve various levels of the cache hierarchy and main memory. As an alternative, we can leverage the scratchpad memory of the PEs (Figure 2.2) to store intermediate cost matrix values. In the strip mining using scratchpad memory algorithm, the first row worker reads the cost matrix from memory only during the first iteration (i.e., only for the first strip of the algorithm). Similarly, the last row worker stores the scores to the linear space cost matrix M in memory only in the last iteration (last strip of the algorithm). In all other intermediate iterations, the first row worker reads the cost matrix values from scratchpad memory, where the last row worker has saved its computed cost matrix values in the previous iteration. This reduces the number of memory reads and writes for the cost matrix to $O(n)$. The scratchpad memory based algorithm operates similarly as the memory based strip mining algorithm. The key difference is that only the initial and final rows are read from or written to memory. In Figure 2.9, the hash-patterned cells of the cost matrix are stored in internal PE scratchpad memory by the last row worker and read by the first row worker $\frac{m}{w} - 2$ times.

One drawback of this approach is that the amount of scratchpad memory on spatial architectures is limited and therefore can limit the maximum length of T. Note that although the use of scratchpad reduces the number of memory reads and writes from/to the linear cost matrix from $O\left(\frac{m}{w}n\right)$ to $O\left(n\right)$, $O\left(\frac{m}{w}n\right)$ reads of string T are required in both strip mining approaches. The tiling based computation as discussed in the next subsection provides a way to deal with this limited amount of scratchpad storage, and can reduce the number of memory reads and writes even further if a proper tile size is chosen.

Memory Loads/Stores and Time Complexity:

For the strip mining algorithm using memory, the total number of memory reads and writes is $O\left((3(\frac{m}{w}n) + m\right)$. This cost comes from $O\left(\frac{m}{w}\right)$ memory reads of the cost matrix and string T of length n, and $\frac{m}{w}$ memory writes to cost matrix of the same size. String S of size m must also be read once for the entire computation. Similarly, for the strip mining algorithm that uses scratchpad memory to store and read the cost matrix values, the total number of memory reads and writes is $O\left((\frac{m}{w}n) + 2n + m\right)$ where $(\frac{m}{w}n)$ comes from reading T, $2n$ comes from reading and writing to linear space cost matrix and m comes from reading S. The running time of this algorithm with w row workers is: $T_w = \Theta((\frac{m}{w}n) + w\frac{m}{w}) = \Theta((\frac{m}{w}n) + m)$ where the second term comes from the synchronization cost of w row workers at the end of each strip. Hence, the running time with an infinite number of row workers (as well as m row workers) is: $T_\infty = \Theta(m + n)$, which is the best span one can achieve for edit distance problem.

2.4.5 Tiling

In the tiling algorithm, we virtually divide the two-dimensional cost matrix into tiles of size $w \times D$, where w denote the number of row workers (also the height of the tiles in this case), and D denote the width of the tiles. We solve the column of tiles (*column strips*) one by one starting from the leftmost column of tiles, ending with the rightmost column of tiles. For the tiling algorithm also, we use linear $O(n)$ memory space to store the cost matrix values from which the first row worker reads its *top* inputs and the last row worker writes its computed cost values. Additionally, we use two other $O(m)$ sized memory arrays to store the left most column of a column strip and the right most column of a column strip. These two arrays work as input and output in alternative iterations (column strips). All w row workers first compute values for the first column strip of $\frac{m}{w}$ tiles of size $w \times D$, each of which ends at the $(D+1)^{th}$ column of the original cost matrix M. Then the row workers compute the next column strip consisted of $\frac{m}{w}$ tiles ending at $(2D+1)^{th}$ column as shown in Figure 2.10 and so on.

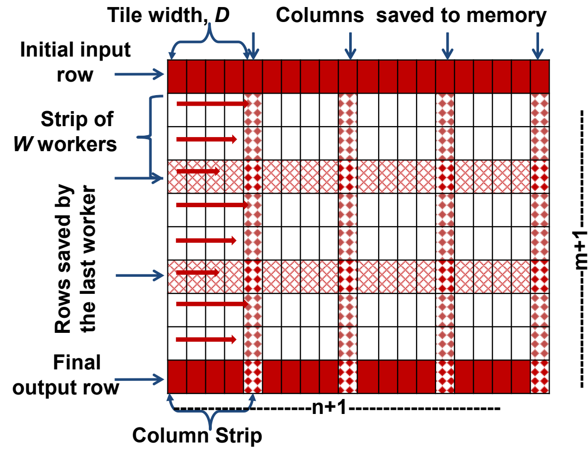


FIGURE 2.10: *Tiled approach. Computation occurs on a column strip of tiles. Initial and final rows are read from and written to memory, while intermediate rows use scratchpad memory. Column results are saved to memory as well.*

The computation for a single column of tiles (column strip) is similar to the strip mining algorithm using scratchpad memory with a few exceptions.

- ▷ Each row worker starts from the 1st column, instead of the 0th column. Therefore, each row worker needs to read $left$ and *diagonal* cells for the very first column of each tile from memory.
- ▷ Each row worker of a tile needs to store the last value of its row to memory so that they can be used as input (*left* and *diagonal* cells) for the next column strip.
- ▷ Only the first row worker of a column strip reads a segment of string T from memory at the beginning of that column strip and all other row workers receive T from the previous row worker and forward T to the next row worker. The last row worker stores T in local scratchpad memory, so that for the next tile the first row worker does not need to read T from memory.

Therefore, over all the tiles, T needs to be read only once (cost ($O(n)$) to compute all $O(mn)$ cells of the cost matrix.

Figure 2.10 shows how the tiling based algorithm works. The two-dimensional cost matrix has been divided into tiles where the hash-patterned cells along the rows are stored in scratchpad memory and the checkerboard-patterned cells along the columns are stored in memory. Those checkerboard-patterned cells are used as $left$, *diagonal* inputs and the hash-patterned cells are

used as *top* inputs for the next column of tiles. Note that we have a choice on whether the intermediate rows, intermediate columns, or both dimensions should be saved in scratchpad memory based upon each dimension's size and the amount of available scratchpad memory. In general the maximum tile width $D_{max} = min(\frac{Total\ Scratchpad\ Memory\ Size}{2}, \frac{n}{2})$.

Memory Loads/Stores and Time Complexity:

To show that the tiling approach has better theoretical memory bandwidth utilization than that of the strip mining approach, we count the total number of memory reads and writes required by the tiled approach. The total number of memory reads in this approach is $O(\big(3(n\frac{m}{D})\big) + 2n)$, where the $O\left(3(n\frac{m}{D})\right)$ comes from reading S, $left$ and *diagonal* cells at the beginning of each tile. Because for each of those terms, there are m memory reads for each column strip, and in total we have $\frac{n}{D}$ column strips/iterations. On the other hand, the $2n$ term comes from reading T and the cost matrix of size n from memory. In addition, there are $O(n)$ writes to memory for writing the final cost values (hashed-patterned cells in Figure 2.10) and $O\left(n\frac{m}{D}\right)$ writes for writing the end cells (rightmost column) for each column strip (checkerboard-patterned cells in Figure 2.10). Therefore, in the tiled approach, we have $\left(4n\frac{m}{D} + 3n\right)$ memory operations. Clearly, the tiled based approach will perform better than the strip mining with scratchpad memory based approach iff $\left(4n\frac{m}{D} + 3n\right) < \left((n\frac{m}{w}) + 2n + m\right) \implies D_{min} > 4w$ (considering $n = m$). As w is constrained by the number of PEs in the spatial architecture and D is constrained by the total aggregated scratchpad memory for all PEs, D will satisfy the condition trivially. The running time for this algorithm with w row workers is $T_w = \Theta\left(\frac{n}{D}(\frac{mD}{w} + w\frac{m}{w})\right) = \Theta\left(\frac{mn}{w} + \frac{mn}{D}\right)$.

2.4.6 Linear Memory Space Traceback Path

In both of our strip-mining and tiled algorithms, we use linear memory space for computing the score. In addition to that, we reduce the number of actual memory reads and writes in the spatial architecture by utilizing direct PE-to-PE communication, something not possible in a general-purpose processor architecture. However, the specific edits required to achieve the minimum edit distance are often needed alongside the scores. Storing the edits for each cell in the cost matrix requires quadratic memory space which is very expensive for large string inputs. Fortunately, there are algorithms that can reproduce the edits without storing the edits initially (Hirschberg [103] and Chowdhury [47]) and without the requirement for quadratic memory space. However, such algorithms require extra $O(mn)$ work to do so. The algorithm in [47] is a divide-and-conquer based recursive algorithm which executes the edit distance algorithm in two passes: the forward pass and the backward pass. The algorithm assumes that it is given input boundaries (the 1st row and 1st column of the cost matrix) and at the end it will produce output boundaries (rightmost column and bottommost row). In the forward pass, the algorithm computes the score by recursively dividing a virtual two-dimensional cost matrix into four quadrants and keeps dividing until it reaches a small base case size when it solves for edit distance using the standard dynamic programming algorithm using linear memory space. During the forward pass, the algorithm saves additional information about where the path from each cell of the output boundary (rightmost column and bottommost row) intersects the input boundary (leftmost column and topmost row). In the backward pass, it recursively executes the edit distance in backward direction to reconstruct the path information. The algorithm decides

PEs	32 Total - Each with 16 Instructions 8 local registers, 8 predicates
Network	Mesh (1 cycle link latency)
Scratchpad	8KB (distributed)
L1 Cache	4KB (4 banks, 1KB/bank)
L2 Cache	24 KB shared slice
DRAM	200 cycle latency
Estimated Clock Rate	3.4 GHz

TABLE 2.1: *Block architectural parameters*

which quadrant to explore based on where the path from the bottom-right point intersects the input boundaries and hence solves only those segments required to reconstruct the edits/paths.

As we have theoretically shown that use of spatial architecture reduces the total memory footprints significantly by using the PE-to-PE communications, it is easy to predict that all the traditional linear memory space algorithms (Hirschberg's or Chowdhury's) can achieve huge performance boost by using our optimized linear memory space edit distance row workers to compute the cost as well as the required edits on spatial architectures. In that case, the recursive control structure needs to be kept in the host process and whenever a basecase needs to be executed it can be executed on the spatial architecture.

2.5 Experimental Setup

2.5.1 Spatial Architecture Performance

We evaluate our edit distance implementations on a cycle-accurate performance model that simulates the triggered instruction-based scalable spatial architecture (TIA) in [143]. The performance model is developed using Asim, an established performance modeling infrastructure [71]. We use a model of the detailed microarchitecture of each TIA PE in the array, the mesh interconnection network, L1 and L2 caches, and DRAM.

The architectural organization of our evaluation architecture can be found in Figure 2.2. The architecture is built from an array of TIA PEs organized into blocks. Each block contains a grid of interconnected PEs, a set of scratchpad memory slices distributed across the block, a private L1 cache, and a slice of a shared L2 cache that scales with the number of blocks on the fabric. Table 2.1 provides the parameters that we use in our evaluation. The TIA PEs use 32-bit integer/fixed-point datapaths, and do not include hardware floating point units. As a reference, 12 blocks (each including PEs, caches, etc.) are about the same size as our baseline Intel® Core™i7 – 3770 core (including L1 and L2 caches), normalized to the same technology node. Also note, that the length of a clock cycle of both a high-end x86 core and TIA are estimated to be the same [143]. We used one block of PEs (32 PEs in total) to conduct all our experiments, and then extrapolated the results to 12 blocks. As in other accelerators, a general-purpose processor is used as the host device responsible for the setup of the kernel on the TIA architecture. The interface between the two devices is shared memory managed using cache coherence to transfer data, eliminating much of the communication overhead to transfer the initial strings as input and score (and the path) as output.

2.5.2 Spatial Edit Distance Implementation

We translate the data flow diagram of the row worker to triggered instruction code by mapping each step of the data flow diagram to one or more rules and their corresponding guard conditions that fire/trigger those rules. A sample code snippet that calculates the match/-substitute cost is shown in Figure 2.11 that follows a similar convention to code as in [143]. In this example, the predicates help to define states and guard conditions that determine relevant program state transitions and enable specific instructions to fire. For example, a PE first checks whether it is in state 0 and both the channels Si_In (S[i]) and Tj_In (T[j]) have inputs, then the PE compares Si_In with Tj_In, decides whether there is a match, dequeues the Si_In channel, and moves to state 1 (p1=0, p0=1). In state 1 the PE forwards Tj_In to the next row worker through its Tj_Out channel, dequeues the Tj_In channel and moves to state 3 (p1=1, p0=1). Finally, based on whether the PE found a match or mismatch in state 0, it computes the match/substitute cost from either *diagonal* or *(diagonal* + 1*)* and again moves to state 0.

```
Module MatchCost()
{
   predicate match;
   predicate p0 = false, p1 = false;
   doCMP_Si_Tj:
          when (!p0 && !p1 && %Si_In.tag != EOL && %Tj_In.tag != EOL) do
          cmp.ge match, %Si_In.data, %Tj_In.data (deq % Si_In, p0:=1)

    Forward_Tj:
          when (p0 && !p1) do
            enq %TjOut, % Tj_In.data (deq % Tj_In, p1:=1)

     Write_Match_Cost:
          when (p0 && p1 && match && %diag.tag != EOL  ) do
            enq %score, % diag.data (deq % diag, p0 := 0, p1=0)

   Write_Substitute_Cost:
          when (p0 && p1 && !match && %diag.tag != EOL  ) do
          enq %score, ADD(% diag.data, 1)  (deq % diag, p0 := 0, p1=0)
}
```

FIGURE 2.11: *Sample code for a module that matches S[i] with T[j] and computes the cost of cell M[i][j] based on the diagonal cell M[i-1][j-1].*

It turned out that we need only 2 PEs to implement a row worker. For integration with the host processor, we set up the input and output datasets and leverage control registers within the TIA architecture. The host sets up the memory space for the cost matrix and writes the memory pointers of the cost matrix and input strings into the TIA architecture registers. Then the host signals the TIA architecture to start computation. The host waits for the TIA to signal that the row workers have finished their computation before leveraging cache coherence hardware to collect the resultant data.

2.5.3 x86 Comparison

For our comparisons to a general-purpose processor, we performed real runtime and power measurements using an Intel® Core™i7-3770. The i7-3770 is a four-core, eight-thread processor which operates at a 3.4 GHz frequency (3.9 GHz Turbo Boost) and is manufactured in 22nm technology. To capture runtime for each experiment, we looped over the computation with enough iterations so that the caches were properly warmed up and so that execution took on the order of seconds in wall time, which enabled us to use operating system timers. To capture energy consumption, we utilized the LIKWID tool set which reads registers included in the Intel®Core® i7-3770 processor to read energy consumption and power [184]. Note that while we do not model power for DRAM accesses, we eliminated most DRAM accesses using code optimizations.

Algorithm	Platform	Workers / PEs per Block	Cells / cycle	Speedup over x86	Area-Normalized Speedup over x86
Score and Path	**x86 Optimized**	–	0.36	1.0	1.0
	TIA StripMem	16 : 32	1.11	3.08	37.0
	TIA StripSP	14 : 30	1.53	4.25	51.0
	TIA Tiled	10 : 22	1.12	3.11	37.3
Score Only	**x86 Optimized**	–	0.48	1.00	1.00
	TIA StripMem	16 : 32	2.14	4.45	53.5
	TIA StripSP	14 : 30	1.80	3.75	45.0
	TIA Tiled	14 : 30	1.88	3.92	47.0

TABLE 2.2: *Performance of edit distance on the spatial architecture and a comparison with a typical modern processor.*

For our x86 software version, we started with a C++ naïve implementation of edit distance and progressively applied both source-level and compiler-level optimizations to improve performance and energy consumption. To enable parallelization of the x86 implementation of edit distance to multiple threads, we tiled computation and used OpenMP. Tiling divides the two-dimensional cost matrix into blocks, and then uses multiple threads to compute blocks of cells along a diagonal starting from the top-left diagonal and ending to the bottom-right diagonal (see Figure 2.12).

Each diagonal wavefront of blocks can be computed in parallel, with synchronization occurring between wavefronts. To improve scalability of our x86 implementation for edit distance to larger string sizes, we performed an additional source-level transformation to our tiled version to enable the use of linear memory space and reduce the size of the cost matrix [47]. In practice, this often reduces the memory footprint requirements from gigabytes of memory to kilobytes, which further improves cache behavior and avoids energy expensive DRAM accesses. For the final source-level transformation, we transformed each tile computation to enable vectorization by the compiler. As parallelism is found along the diagonals, we transformed each tile into the shape of a diamond so that parallel work exists horizontally in memory.For our compiler, we used the Intel® *icc* compiler version 14.0 and experimented with optimization levels $-O0$ through $-O3$ and AVX vectorization. For details please see [179].

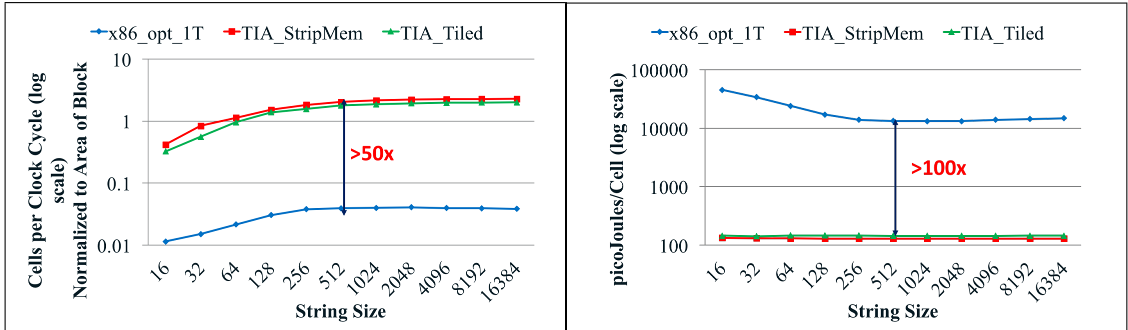
FIGURE 2.12: *Execution pattern of the x86 based tiled-loop code. The numbers on the tile shows when a tile get executed. Tiles with the same numbers gets executed in parallel.*

2.6 Experimental Results and Analysis

2.6.1 Overview of Performance Results

Table 2.2 presents the overview of the performance results for a fixed input string size of 1024 for both S and T. While in this paper we focus on computing the score matrix, the results

for computing the score and path (i.e., edits) together are also presented. Scores are calculated using $O(n)$ memory space and edits are saved using $O(nm)$ memory space. Recall that it is possible to reconstruct the edits without saving the edits at the first place, by recomputing a smaller subset of the scores in the backward direction. Therefore, showing that computing the scores on TIA is significantly faster than computing the score on x86 is still of great interest. The x86 implementation presented uses all the optimizations previously mentioned (i.e., linear space tiling, vectorization and compiler optimizations). Though the TIA implementations are scalable with the total number of PEs available, we limit the computation fabric to a single block and normalize to the area of an x86 core by multiplying by a factor of 12 (since approximately 12 blocks consume the area of an Intel® Core$^{\mathrm{TM}}$i7 − 3770 core).



(A) *Throughput performance comparison.* (B) *Energy consumption comparison.*

FIGURE 2.13: *Results showing that implementations on triggered instruction spatial architecture (TIA) run* 50× *faster while consuming* 1/100× *energy compared to the optimized single-threaded x86 based tiled-loop implementation.*

From Table 2.2 we can see that strip mining using scratchpad memory is the fastest (achieves a 51 times speedup with respect to x86) among the algorithms that compute both score and path. We were only able to use 10 row workers (22 PEs) for the tiled based approach as we were resource constrained by the number of communication channels used by the row workers to read to and write from the memory. Despite these constraints, the tile-based approach was slightly better than strip mining using memory in this case. For algorithms that only compute the score, the performance of all algorithms improved because the overall work was reduced by not saving $O(nm)$ edits to memory. When computing the score only, all our algorithms on TIA also performed, at least, 45 times faster than the x86 based version. Note that our extrapolated performance when scaling TIA to 12 blocks is conservative, as we assume the number of row workers is kept constant for each block. For example, strip mining using scratchpad memory requires two control PEs to work as a multiplexer and de-multiplexer attached to the first and last row workers, effectively limiting the maximum number of row workers to 14 instead of 16. However, this control overhead need not be replicated when scaling to 12 blocks, and performance would therefore be better than our extrapolated number. Figures 2.13a and 2.13b show that in addition to be 50× faster, TIA based implementation consumes 100× less energy and 2× less power than the x86 based implementation. The experiments related to energy/power consumption and memory footprints were conducted by my co-author Neal Crago [179][2]. Nevertheless,

[2]My contributions in this work are to find this class of DP problems that can be mapped naturally to spatial architecture, all algorithms and their implementations for edit distance on the TIA architecture and the naïve linear space x86 implementation. Results presented in Table 2.2 and Figure 2.13a were also produced by me, which were later updated with the x86 optimized results. All theoretical analyses of memory operations and runtime complexity were also done by me.

some of those results are reproduced for showing significance of this work.

2.6.2 Memory References and Communication

Figure 2.14 compares the number of memory accesses, local communication between PEs, and scratchpad memory accesses between x86 and TIA. In this analysis, we compare our fully-optimized single-threaded x86 implementation with two of our TIA implementations on a dataset where strings S and T are both of length 1024. Activity numbers for TIA were gathered from the performance model, while the number of memory accesses on x86 were gathered using cachegrind, a performance instrumentation tool from Linux [138]. We find that over 95.2% and 97.3% of memory accesses in both TIA implementations are successfully converted into less expensive local communication between PEs and scratchpad memory accesses. The dramatic reduction in memory accesses is a key benefit in accelerating applications using spatial architectures. Note that the amount of local communication on TIA is of the order of the number of memory accesses in the x86 version. While register file accesses are not explicitly shown, the TIA local communication numbers include accesses that would be facilitated on x86 using the register file. This conversion of memory accesses to local communication corroborates well with the organization of our TIA implementations, where local communication is primarily used to send scores and other inputs between PEs.

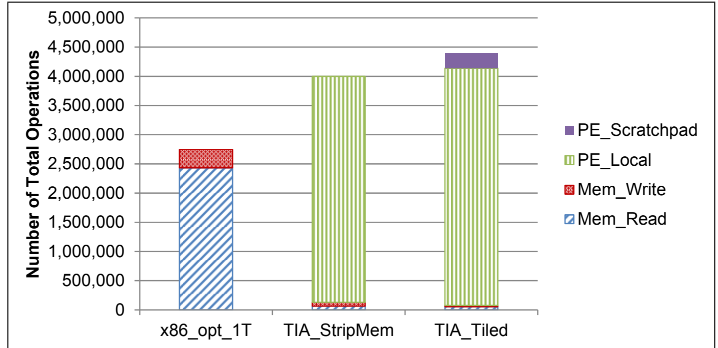


FIGURE 2.14: *Comparison of memory, local communication, and scratchpad memory accesses between x86 and TIA based implementations. While register file activity is not shown, TIA local communication numbers include what would be register file accesses on x86.*

2.6.3 Coding Effort Analysis

To further compare the two platforms, we analyzed the code footprint of the x86 and TIA kernels. Performing this analysis provides further insight into the natural mapping of edit distance onto spatial architectures. Table 2.3 presents the lines of source code for x86 (C++) and TIA (Assembly). For this analysis, we only included lines in each kernel containing real work, and excluded lines entirely devoted to comments, whitespace characters, and control characters such as braces {}. For the C++ x86 version, we further optimized code footprint by aggressively modularizing code into reusable functions that could be inlined by the compiler.

We find that the number of lines of TIA assembly is nearly on the order of the fully optimized C++ x86 version. This is an interesting result, given the superior expressibility of C++ for computation. However, the effort required to optimize the x86 version for scalability and performance adds significant code complexity. While scaling the number of row workers is trivial in TIA, x86 must change the algorithm and implement blocking and OpenMP support to facilitate parallelization. Similarly, implementing vectorization on x86 results in an algorithmic change

Platform	Version	Lines of source code
x86 (C++)	Naive	10
	+blocking	65
	+linear memory space	144
	+vectorization	168
TIA (Assembly)	StripMem	222
	Tiled	313

TABLE 2.3: *Coding effort for edit distance (code footprint).*

and corner cases that must be handled with additional code. Overall, we feel that this data analysis reflects the natural mapping of edit distance onto TIA.

2.7 Related Work

There is considerable amount of prior research in developing and optimizing edit distance and its several variants on general-purpose processors such as x86. Hirschberg [103] first discovered a linear-space algorithm for sequence alignment, which was then popularized and extended by Myers and Miller [137]. There are also several multicore-based implementations of edit distance as well as sequence alignment. In [47] the authors have presented a cache-oblivious divide-and-conquer algorithm for multicores, where the cost matrix is divided into four quadrants to be solved recursively, and the diagonal quadrants are solved in parallel. Other prior research on edit distance has focused on parallelization targeting both MIMD [106] and SIMD [94] architectures. Vectorization of the sequence alignment problem by reshaping the cost matrix has been described in [101].

With the recent surge of research on accelerator architectures, edit distance and its variants have also been mapped to GPUs, FPGAs, and reconfigurable hardware. These prior researches focus on mapping to the data-parallel nature of the architectures, with optimization utilizing inter- and intra- task parallelism, tiling, pruning, and approximate solutions. Given the difficulty in vectorizing edit distance, much of this prior research finds parallelism by performing computation across multiple sets of small gene sequences [62, 73]. However, there exists some research which focuses on improving performance for a single large sequence alignment problem [114]. Some FPGA and reconfigurable hardware research leverage the strong dataflow properties within the algorithm in order to exploit parallelism [68, 150, 156]. While much of this research focuses on improving throughput performance, some work also emphasizes reducing logic footprint [113].

In contrast to our work, no prior research focuses on analyzing energy or power consumption of edit distance or compares fully-optimized implementations on two platforms with different architectural properties. We develop edit distance algorithms for the emerging class of coarse-grained spatial architectures, and leverage best-known techniques to develop an optimized x86 implementation of edit distance. We concentrate on developing a scalable implementation for spatial architectures that operates on arbitrarily large source and target strings while minimizing memory bandwidth by optimizing communication among the PEs. We also optimize for a single instance of edit distance operating on a single pair of strings, rather than gaining parallelism through multiple computations. Finally, both x86 and TIA implementations are evaluated on high-end evaluation platforms to further ensure a fair comparison.

2.8 Conclusion and Future Research

This research shows that finding the right type of parallelism and a right architecture to map that parallelism can save both memory bandwidth and energy consumption significantly while giving the desired speedup. We demonstrate the ability of a triggered instruction-based spatial architecture to solve the edit distance problem, a broadly used dynamic programming problem in bioinformatics. Our experiments show that this proposed spatial architecture has a tremendous potential of providing high performance for applications with local communications. We conclude that for applications where vectorization is not straightforward or inefficient due to horizontal and vertical dependencies between the computation elements, it is possible to map them to a spatial architecture more efficiently than an x86 processor.

Exploring the possibility of mapping of other dynamic programming problems with non-local dependencies efficiently on this kind of spatial architecture can be considered as the next step in this research. Solving cache-oblivious wavefront (COW) algorithms [173] on the Triggered Instruction Spatial Architectures (TIA) seems to be a very interesting research to pursue, since both use the concept of the trigger: COW algorithms use the triggers in its software scheduler, where triggered instruction spatial architecture implements that in hardware. Offloading basecases of cache-oblivious recursive divide-and-conquer algorithm for edit distance algorithms to TIA may improve performance even further. All these are open problems and need further investigations.

Chapter 3

Recursive Dynamic Programming With Matrix-multiplication-like Flexible Kernels

3.1 Abstract

In Chapter 2 we have shown how to get high-performance for edit distance like dynamic programming algorithms using a special-purpose hardware that can efficiently leverage the inherent data-flow property of the algorithm. In this chapter we show how to achieve high-performance while solving dynamic programming problems on general-purpose hardware, e.g., traditional multicore processors.

We show how to obtain high-performing parallel implementations for a class of dynamic programming (DP) problems by reducing them to highly optimizable flexible kernels using cache-oblivious recursive divide and conquer (CORDAC). We implement parallel CORDAC algorithms for four non-trivial DP problems, namely the parenthesis 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 the 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 for these CORDAC algorithms achieve $5 - 150\times$ speedup and $3 - 40\times$ reduction in energy consumption over their standard loop based DP counterparts 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 than these auto-generated tiled codes. Finally, we present results on manycores (Intel Xeon Phi) and clusters

of multicores obtained using simple extensions for SIMD (Single Instruction Multiple Data) and shared-distributed-shared-memory architectures, respectively, demonstrating the versatility and portability of these algorithms. Our optimization approach is highly systematic and suitable for automation.

3.2 Introduction

Dynamic programming (DP) [20, 120, 162] 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 [121] 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 [67, 97, 189].

Traditional Loop-based DP Algorithms. Dynamic programs are traditionally implemented using simple loop-based algorithms. For example, Figure 3.2 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 result 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 × n matrix c[1 : n, 1 : n] with c[i, j] = v_j for
1 ≤ i = j − 1 < n and c[i, j] = ∞ otherwise (i.e., i ≠ j − 1).)

1. for i ← n − 2 downto 1 do
2.     for j ← i + 2 to n do
3.         for k ← i + 1 to j do
4.             c[i, j] ← 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 × n matrices a, b and d. This function
computes the product of a and b in d.)

1. for i ← 1 to n do
2.     for j ← 1 to n do
3.         for k ← 1 to n do
4.             d[i, j] ← d[i, j] + a[i, k] × b[k, j]
```

Figure 3.1: *Inflexible looping code for the parenthesis 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 Figure 3.1 incurs $\Theta\left(n^3\right)$ cache misses under the *ideal-cache model* [81]. 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,

[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.

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 Figure 3.1. 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. 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. Furthermore, 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 [195]. In this research, 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] [81].

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. Tiling for machines with hierarchical caches is very inconvenient and challenging. 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 better parallelism than its parallel looping counterpart.

Our Contributions. We consider four DP problems with applications to computational biology. Among them, the *parenthesis problem* [84] arises in sequence analyses and in RNA secondary structure prediction [125, 169] 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 [144], and the *protein accordion folding (PAF) problem* has its roots in protein structure prediction.

Our major contributions in this work are as follows.

[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.

▷ [***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 polyhedral compilers - PLuTo [29], PoCC [148] and Polly [95].

▷ [***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 the looping codes. 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, the same basic CORDAC algorithms also perform very well on manycores Xeon Phi as well as on hybrid CPU + Xeon Phi platforms. These results show portability of these algorithms on different parallel platforms.

3.3 Algorithms

In this section, we present standard parallel loop-based and CORDAC algorithms for the parenthesis, protein accordion folding, gap, and FW-APSP 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 3.1 lists span and cache complexity of all the four CORDAC algorithms and their iterative counterparts.

3.3.1 Parenthesis Problem

The *parenthesis problem* [84] 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 $x_i \cdots x_j$ (e.g., in case of the equivalent matrix-chain multiplication problem, the cost refers to the computational cost

```
PAR-LOOP-PARENTHESIS( c, n ) (Input is an n × n matrix c[1 : n, 1 : n] with c[i, j] = v_j for 1 ≤ i = j − 1 < n and
c[i, j] = ∞ otherwise (i.e., i ≠ j − 1).)
1. for t ← 2 to n − 1 do
2.     par for i ← 1 to n − t do
3.         j ← t + i
4.         for k ← i + 1 to j do
5.             c[i, j] ← min { c[i, j], c[i, k] + c[k, j] + w(i, k, j) }
```

```
PAR-LOOP-FW( d, n ) (Input is an n × n matrix d[1 : n, 1 : n] with d[i, j] for 1 ≤ i, j ≤ n initialized with entries from a
closed semiring (S, ⊕, ⊙, 0, 1). )
1. for k ← 1 to n do
2.     par for i ← 1 to n do
3.         par for j ← 1 to n do
4.             d[i, j] ← d[i, j] ⊕ (d[i, k] ⊙ d[k, j])
```

```
PAR-LOOP-PROTEIN-FOLDING( S, F, n ) (Inputs are two n × n matrices S[1 : n, 1 : n] and F[1 : n, 1 : n]. For a given
protein sequence P[1 : n], the cost of an optimal accordion score of the segment P[i : j] will be computed in S[i, j]. F is
a precomputed array with F[j+1, min (k, 2j − i + 1)], 1 ≤ i < j < k − 1 < n, storing the number of aligned hydrophobic
amino acids when the protein segment P[i : k] is folded only once at indices (j, j + 1). For n − 1 ≤ j ≤ n, each S[i, j] is
initialized to 0.)
1. for i ← n − 1 downto 1 do
2.     par for j ← n − 1 downto i + 1 do
3.         for k ← j + 2 to n do
4.             S[i, j] ←
               max { S[i, j], S[j + 1, k] + F[j + 1, min (k, 2j − i + 1)] }
```

```
PAR-LOOP-GAP( G, x, m, y, n ) (Inputs are two sequences x = x_1 x_2 ... x_m and y = y_1 y_2 ... y_n, and an (m+1) × (n+1)
matrix G[0 : m, 0 : n]. Row 0 and column 0 of G re assumed to be appropriately initialized. )
1. for t ← 2 to m + n do
2.     par for i ← max {1, t − n} to min {t − 1, m} do
3.         j ← t − i
4.         G[i, j] ← G[i − 1, j − 1] + s(x_i, y_j)
5.         for q ← 0 to j − 1 do
6.             G[i, j] ← min {G[i, j], G[i, q] + w_1(q, j)}
7.         for p ← 0 to i − 1 do
8.             G[i, j] ← min {G[i, j], G[p, j] + w_2(p, i)}
```

FIGURE 3.2: *Loop-based parallel codes for the parenthesis problem (*PAR-LOOP-PARENTHESIS*), Floyd-Warshall's APSP (*PAR-LOOP-FW*), protein accordion folding (*PAR-LOOP-PROTEIN-FOLDING*) and the gap problem (*PAR-LOOP-GAP*). 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 [82].*

of multiplying matrices from $x_i \cdots x_j$). For $1 \le i < n$, each $c[i, i+1]$ is assumed to be already known ($= v_{i+1}$), and for $1 \le i \le n$ each $c[i, i]$ is assumed to be ∞.

A function $w(\cdot, \cdot, \cdot)$ is given such that for $1 \le i < k \le n$, $w(i, k, j)$ 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 $1 \le i < j - 1 \le n$, $c[i, j]$ is computed as follows.

$$c[i, j] = \min_{i \le k \le j} \left\{ (c[i, k] + c[k, j]) + w(i, k, j) \right\}$$

The optimal parenthesizing cost $c[1, n]$ for the entire sequence can be found using the parallel looping code PAR-LOOP-PARENTHESIS given in Figure 3.2. Observe that the parallel looping code is different from the serial code LOOP-PARENTHESIS shown in Figure 3.1 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 $c[1, n]$. All cells on the same diagonal can now be computed in parallel.

A parallel CORDAC algorithm for solving the parenthesis problem is shown in Figure 3.5 which is a special case of the algorithm we proposed in [47]. 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_{21} depends only on data in upper triangular submatrices U_{22} and V_{11}, and hence can be updated recursively before updating X_{11}, X_{22} and X_{12}. Next we can update X_{11}, X_{22} while using X_{21} as input. 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}. Once we have updated X_{11} and X_{22}, next we can update X_{12}. 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) $c[i, k] \in X_{12} \wedge c[k, j] \in V_{22}$. 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 the recursive square matrix-multiplication algorithm.

Table 3.1 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.

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, $Q_A(n) = 2Q_A(n/2) + Q_B(n/2)$, $Q_B(n) = 4\left(Q_B(n/2) + Q_C(n/2)\right)$, 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, $T_A(n) = T_A(n/2) + T_B(n/2) + \Theta(1)$, $T_B(n) = 3\left(T_B(n/2) + T_C(n/2)\right) + \Theta(1)$, and $T_C(n) = 2T_C(n/2) + \Theta(1)$. Solving, $T_A(n) = \mathcal{O}\left(n^{\log_2 3}\right)$.

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_∞)
Parenthesis (*f* = *par*)	$\Theta(m)$	$\Theta(m^2)$	$\mathbf{\Theta(m^3)}$	-	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$
Protein Folding (*f* = *fold*)	$\Theta(m)$	$\mathbf{\Theta(m^2)}$	$\Theta(m^2)$	$\mathbf{\Theta(m^3)}$	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n\log n)$
Gap Problem (*f* = *gap*)	$\Theta(m)$	$\mathbf{\Theta(m^3)}$	$\mathbf{\Theta(m^3)}$	-	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$
FW-APSP (*f* = *FW*)	$\Theta(m)$	$\Theta(m^2)$	$\Theta(m^2)$	$\mathbf{\Theta(m^3)}$	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n\log n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n\log^2 n)$

TABLE 3.1: *Complexities of the iterative and recursive divide-and-conquer (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 basecase size $\leq b \times b$. Flexible kernels are shown on yellow background and asymptotically dominating kernels are shown in bold 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.*

3.3.2 Protein Accordion Folding

A protein can be viewed as a string $P[1:n]$ over the alphabet { *A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V* } of amino acids[5]. A protein sequence is never straight, and instead, it folds itself in a way that minimizes the potential energy. Some of the amino acids (e.g., *A, I, L, F, G, P, V*) are called *hydrophobic* as they do not like to be in contact with water. A desire to minimize the total hydrophobic area exposed to water is a major driving force behind the folding process. In a folded protein, hydrophobic amino acids tend to clump together in order to reduce water-exposed hydrophobic area.

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 (see [112]). 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 often fold this way (Figure 3.3).

The recurrence below computes the optimal *accordion score*, $S[i,j]$ 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 $\text{SOF}(i,j,k)$, which stands for SCORE-ONE-FOLD, counts the number of aligned hydrophobic amino acids when the protein segment $\mathcal{P}[i:k]$ is folded only once at indices $(j, j+1)$. The function is illustrated graphically in Figure 3.4. Observe that

$$\text{SOF}(i,j,k) = \begin{cases} \text{SOF}(1,j,k) & \text{if } k \leq 2j-i+1, \\ \text{SOF}(1,j,2j-i+1) & \text{otherwise.} \end{cases} \tag{3.1}$$

[5] Amino acids: Alanine (A), Arginine (R), Asparagine (N), Aspartic acid (D), Cysteine (C), Glutamic acid (E), Glutamine (Q), Glycine (G), Histidine (H), Isoleucine (I), Leucine (L), Lysine (K), Methionine (M), Phenylalanine (F), Proline (P), Serine (S), Threonine (T), Tryptophan (W), Tyrosine (Y), Valine (V).

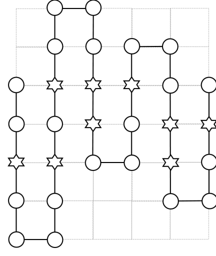


FIGURE 3.3: *A protein accordion fold where each star represents a hydrophobic amino acid and each circle a hydrophilic one. The accordion score of this folded sequence is 4 which is not the maximum possible accordion score for this sequence.*

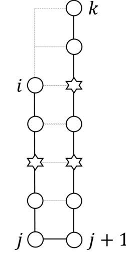


FIGURE 3.4: $\mathrm{SOF}(i,j,k)$ *counts the number of aligned hydrophobic amino acids when the protein segment* $\mathcal{P}[i:k]$ *is folded only once at indices* $(j, j+1)$*. In this figure, each star represents a hydrophobic amino acid and each circle a hydrophilic one.*

Hence, in $\mathcal{O}\left(n^2\right)$ time one can precompute an array $\mathrm{F}[1:n, 1:n]$ such that for all $1 \le i < j < k-1 < n$, $\mathrm{SOF}(i,j,k) = \mathrm{F}[j+1, \min\{k, 2j-i+1\}]$.

Thus Recurrence for accordion fold, S reduces to the following.

$$\mathrm{S}[i,j] = \begin{cases} 0 & \text{if } j \ge n-1, \\ \max_{j+1<k\le n}\{\mathrm{SOF}[j+1, \min\{k, 2j-i+1\}] + \mathrm{S}[j+1,k]\} & \text{otherwise.} \end{cases} \quad (3.2)$$

In Figure 3.5 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 3.1 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.

Serial Cache Complexity. For $f \in \{A, B, C, D\}$, let $Q_f(n)$ denote the cache complexity of f_{fold} on a sequence of length n when run on a serial machine. Then $Q_f(n) = \mathcal{O}\left(n + n^2/B\right)$ if $n^2 \le \gamma_f M$ for some suitable constant $\gamma_f \in (0,1]$. Otherwise, $Q_A(n) = 2Q_A\left(\frac{n}{2}\right) + Q_B\left(\frac{n}{2}\right) + Q_C\left(\frac{n}{2}\right)$, $Q_B(n) = 4Q_B\left(\frac{n}{2}\right) + 2Q_D\left(\frac{n}{2}\right)$, $Q_C(n) = 4Q_C\left(\frac{n}{2}\right) + 2Q_D\left(\frac{n}{2}\right)$, and $Q_D(n) = 8Q_D\left(\frac{n}{2}\right)$.

Solving, $Q_A(n) = \mathcal{O}\left(n + \frac{n^2}{B} + \frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right)$.

Span. For $f \in \{A, B, C\}$, let $T_f(n)$ denote the span of f_{fold} on a sequence of length n. Then $T_f(n) = \Theta(1)$ if $n = 1$. Otherwise, $T_A(n) = 2T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + T_C\left(\frac{n}{2}\right) + \Theta(1)$, $T_B(n) = T_B\left(\frac{n}{2}\right) + T_D\left(\frac{n}{2}\right) + \Theta(1)$, $T_C(n) = 2\max\left\{T_C\left(\frac{n}{2}\right), T_D\left(\frac{n}{2}\right)\right\} + \Theta(1)$, and $T_D(n) = 2T_D\left(\frac{n}{2}\right) + \Theta(1)$

Solving, $T_A(n) = \mathcal{O}(n\log n)$.

```
A_par(X)
1. if X is a small matrix then A_par-loop(X)
   else
2.     par : A_par(X11), A_par(X22)
3.     B_par(X12, X11, X22)

B_par(X, U, V)
1. if X is a small matrix then B_par-loop(X, U, V)
   else
2.     B_par(X21, U22, V11)
3.     par :
       C_par(X11, U12, X21), C_par(X22, X21, V12)
4.     par :
       B_par(X11, U11, V11), B_par(X22, U22, V22)
5.     C_par(X12, U12, X22)
6.     C_par(X12, X11, V12)
7.     B_par(X12, U11, V22)

C_par(X, U, V) //Flexible Function
1. if X is a small matrix then C_par-loop(X, U, V)
   else
2.     par :
       C_par(X11, U11, V11), C_par(X12, U11, V12),
       C_par(X21, U21, V11), C_par(X22, U21, V12)
3.     par :
       C_par(X11, U12, V21), C_par(X12, U12, V22),
       C_par(X21, U22, V21), C_par(X22, U22, V22)

A_fold(X)
1. if X is a small matrix then A_fold-loop(X)
   else
2.     A_fold(X22)
3.     B_fold(X12, X22)
4.     C_fold(X11, X12)
5.     A_fold(X11)

B_fold(X, V) //Flexible Function
1. if X is a small matrix then B_fold-loop(X, V)
   else
2.     par : B_fold(X11, V11), B_fold(X12, V22),
             B_fold(X21, V11), B_fold(X22, V22)
3.     par : D_fold(X11, V12), D_fold(X21, V12)

C_fold(X, U)
1. if X is a small matrix then C_fold-loop(X, U)
   else
2.     par : C_fold(X11, U11), D_fold(X12, U21),
             C_fold(X22, U21),
3.     par : C_fold(X11, U12), D_fold(X12, U22),
             C_fold(X22, U22),

D_fold(X, V) //Flexible Function
1. if X is a small matrix then D_fold-loop(X, V)
   else
2.     par : D_fold(X11, V11), D_fold(X12,V21),
             D_fold(X21, V11),D_fold( X22, V21)
3.     par : D_fold(X11, V12), D_fold(X12, V22),
             D_fold(X21, V12), D_fold(X22, V22)
```

Figure 3.5: *Parallel cache-oblivious recursive divide-and-conquer (CORDAC) algorithms for solving the parenthesis and the protein accordion folding problems. For simplicity, we assume* n *to be a power of* 2. *Initial function calls are as follows.* (1) *parenthesis problem:* $A_{par}(c)$ *for an* $n \times n$ *input matrix* c *and* (2) *protein accordion folding* $A_{fold}(\ X\ )$*, where* $X = \langle S[1:n, 1:n], F[1:n, 1:n] \rangle$ *are* $n \times n$ *input matrices.*

3.3.3 Sequence Alignment with General Gap Penalty

The problem of *sequence alignment with general gap penalty (gap problem)* [83, 84, 189] 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. Although, an affine gap penalty function is predominantly used in bioinformatics, for which $O(n^2)$ algorithms are available [189], [50], 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. To handle any general case, we define two new cost functions w and w', where $w(p, q)$ $(0 \le p < q \le m)$ is the cost of deleting $x_{p+1}\ldots x_q$ from X, and $w'(p, q)$ $(0 \le p < q \le n)$ 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 $0 \le i \le m$ and $0 \le j \le n$) under this general setting. Then $G[0, 0] = 0$, $G[0, j] = w(0, j)$ for $1 \le j \le n$, and $G[i, 0] = w'(0, i)$ for

```
A_gap(X)
1. if X is a small matrix then A_gap-loop(X)
   else
2.     A_gap(X11)
3.     par : B_gap(X12, X11), C_gap(X21, X11)
4.     par : A_gap(X12), A_gap(X21)
5.     B_gap(X22, X21)
6.     C_gap(X22, X12)
7.     A_gap(X22)
```

```
B_gap(X, U) //Flexible Function
1. if X is a small matrix then B_gap-loop(X, U)
   else
2.     par : B_gap(X11, U11), B_gap(X12, U11), B_gap(X21, U21), B_gap(X22, U21)
3.     par : B_gap(X11, U12), B_gap(X12, U12), B_gap(X21, U22), B_gap(X22, U22)
```

```
C_gap(X, V) //Flexible Function
1. if X is a small matrix then C_gap-loop(X, V)
   else
2.     par : C_gap(X11, V11), C_gap(X12, V12), C_gap(X21, V11), C_gap(X22, V12)
3.     par : C_gap(X11, V21), C_gap(X12, V22), C_gap(X21, V21), C_gap(X22, V22)
```

FIGURE 3.6: *Parallel cache-oblivious recursive divide-and-conquer (CORDAC) algorithms for solving the gap problem. For simplicity, we assume n to be a power of 2. $A_{gap}(\ G[1:n, 1:n]\ )$ is the initial function call, where $G[0:n, 0:n]$ is the $(n+1) \times (n+1)$ input matrix.*

$1 \le i \le m$. Otherwise,

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

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

The parallel iterative DP algorithm PAR-LOOP-GAP shown in Figure 3.2 solves the gap problem. In Figure 3.6 we present a parallel CORDAC algorithm for solving the problem which uses three recursive functions A_{gap}, B_{gap} and C_{gap}. A_{gap} updates a square X based on values inside itself, B_{gap}(X, U) updates a square X using values from another square U that lies left to X, and C_{gap}(X, U) updates a square X based on another square V that lies bellow X. The iterative kernels invoked by B_{gap} and C_{gap} are asymptotically dominating and flexible (Table 3.1). Table 3.1 shows the span and cache complexity of these algorithms.

Serial Cache Complexity. For $f \in \{A, B, C\}$, let $Q_f(n)$ denote the cache complexity of f_{gap} on sequences of length n when run on a serial machine. Then $Q_f(n) = \mathcal{O}\left(n + n^2/B\right)$ if $n^2 \le \gamma_f M$ for some suitable constant $\gamma_f \in (0, 1]$. Otherwise, $Q_A(n) = 4Q_{\mathcal{A}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{B}}\left(\frac{n}{2}\right) + 2Q_{\mathcal{C}}\left(\frac{n}{2}\right)$, $Q_B(n) = 8Q_{\mathcal{B}}\left(\frac{n}{2}\right)$, and $Q_C(n) = 8Q_{\mathcal{C}}\left(\frac{n}{2}\right)$. Solving, $Q_{\mathcal{A}}(n) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}} + \frac{n^3}{M} + \frac{n^2}{B} + n\right)$.

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

3.3.4 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 $i, j \in [1, n]$:

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

Floyd-Warshall's all pairs shortest path (FW-APSP) algorithm [77, 188] performs computations over a particular closed semiring $(\Re, \min, +, +\infty, 0)$.

Figure 3.2 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 [50]. Table 3.1 shows that among the four recursive functions in the CORDAC algorithm, only D_{FW} is flexible which is also the dominating one.

3.4 Optimizations

In this section, we discuss optimization strategies that we have used to significantly speed up implementations of the CORDAC algorithms described in Section 3.3.

3.4.1 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 basecase size) to fit into one of the cache levels (often the largest private cache), after which a loop-based code is used to perform the computation [195]. The basecase size also needs to be large enough so that the computation done inside the basecase is able to subsume the overhead of recursion. These hybrid implementations expose optimization opportunities offered by neither the pure iterative nor the pure recursive implementation. The basecase 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 basecase kernels, which brings the following additional benefits.

▷ *Asymptotic Improvement in Parallelism:* In our two-way divide-and-conquer approach (where, each dimension of the subtask is half the dimension of its parent task) 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 [81]. Table 3.1 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 3.1) with or without tiling.

▷ *Highly Optimizable Base Cases:* Running loop-based code on small flexible basecases 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 Figure 3.5 has the same inflexible implementation as Par-Loop-Parenthesis in Figure 3.2, 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 better optimized 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.

3.4.2 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 = basecase size) static arrays inside the kernel, and then read from those local arrays during actual computation. This improves 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 improves data locality, vectorization efficiency and helps in reducing conflict misses significantly in set-associative caching systems. We found that copy-optimization can improve running times over $2\times$.

▷ *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 the 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.

3.4.3 Data Layout

Laying out data in memory matching the order in which they are accessed during the program execution can reduce cache misses by leveraging better spatial locality.

For CORDAC algorithms, use of a 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 basecase size is chosen.

$f_X(X, m, n, ...)$ (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 3.7: *On-the-fly computations of Z-Morton-row-major pointers.*

In all algorithms presented in this chapter, 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 a consistently better performance.

▷ *Z-Morton for any n:* One of the **contributions of this work**, 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 mn elements. Although it is very straightforward to lay out 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. Furthermore, 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 [175]), but padding uses extra space. As shown in the code snippet in Figure 3.7, 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 actual computation, we use ZM_RM pointers and the original row-major indices to compute desired locations on-the-fly inside the recursive functions, which incurs only $\mathcal{O}(1)$ overhead per recursion level.

3.4.4 Auto vs. Explicit Vectorization

Often explicit vectorization can lead to significant speedup over the compiler-assisted auto-vectorization as compilers cannot always automatically detect all possible vectorization opportunities, especially for complicated code with pointers. By explicitly vectorizing, we were able to achieve up to 5× speedup over the auto-vectorized CORDAC code (see Figure 3.8 and Table 3.4). Often vectorizing the basecase of the dominating kernel only (e.g., C_{loop} for the parenthesis problem) is enough to get the major share of the speedup.

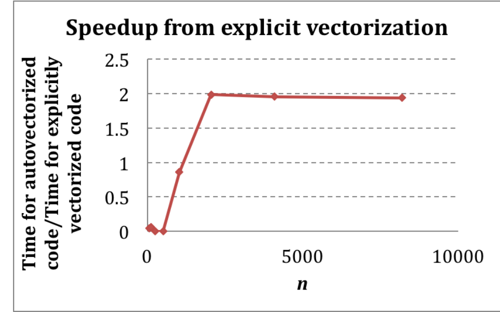


FIGURE 3.8: *Benefit of explicit vectorization over autovectorization for FW-APSP code.*

3.4.5 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.

3.5 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 clusters of multicores.

Property	Intel32	Intel16	XeonPhi	Intel16E (in house)
System	Xeon E5 − 4650	Xeon E5 − 2680	Knights Corner	Xeon E5 − 2650
Clock	2.70 GHz	2.70 GHz	-	2.00 GHz
# Cores	4×8 (32)	2×8 (16)	61	2×8 (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 3.2: *System specifications. Intel16E is used for power and energy analyses.*

All programs were implemented using `C++` with Intel© Cilk™Plus extension for shared memory and MPI for distributed memory parallelization. For each DP 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 CORDAC with copy-optimization.

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

For parenthesis and FW-APSP problems, we further optimized COZ versions with explicit vectorization for CPU and Xeon Phi architectures separately. However, unless otherwise stated, no COZ version incorporates explicit vectorization. We also implemented a *hybrid CPU + Xeon Phi* version where we dynamically offload subproblems to the coprocessor (Xeon Phi) 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 basecase size of 64×64 for parenthesis, gap and protein folding and 128×128 for FW-APSP. Machines from the Stampede Supercomputing Cluster [5] were used to run the experiments and the system specifications can be found in Table 3.2. We used PAPI-5.2 [4] to collect cache misses and LIKWID [184] for the energy/power statistics. The energy/power measurements were end-to-end. Metrics shown in Table 3.3 were used to compare the performance of different algorithms.

Metrics	**Meaning**	**Expected**
UPS	Number of Updates Per Second	Higher
Strong Scalability (T_1^l/T_p)	Runtime of LOOPDP on 1 core / Runtime on p cores	Higher
Cache-miss ratio	Cache-miss of LOOPDP / Cache-miss of CORDAC	Higher
Energy profile	Energy consumption ratio of LOOPDP vs. CORDAC	Higher

TABLE 3.3: *Metrics for performance comparison of different implementations.*

3.5.1 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 3.4 summarizes the results. We observed that explicit vectorization contributed up to $5\times$ speedup over the auto-vectorized code. For parenthesis 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 when $n = 32768$. Overall, the hybrid *CPU + Xeon Phi* version runs $42-50\%$ faster than the pure vectorized CPU version when $n \approx 2^{15}$.

Runtime, Cache Miss, Energy Performance and Scalability. Figures 3.9, 3.10, 3.11 and 3.12 show 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 the parenthesis and gap problems, we get better speedups w.r.t LOOPDP than for FW-APSP and Protein Folding. Theoretical bounds listed in Table 3.1 also

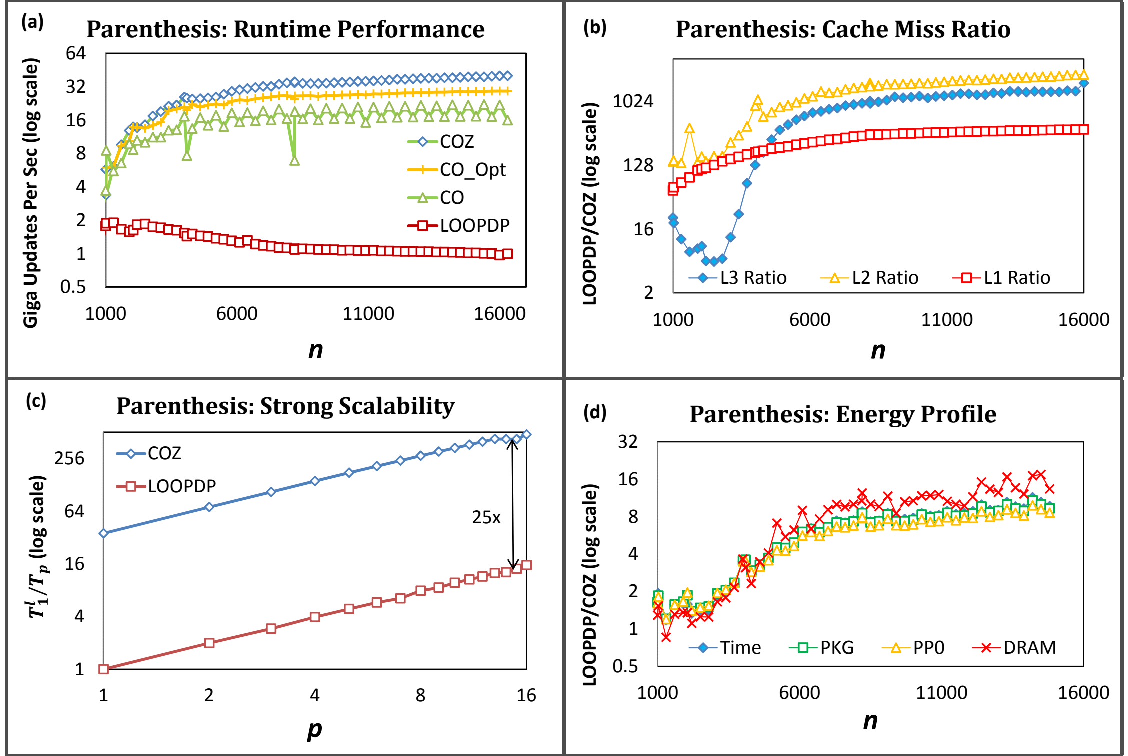


FIGURE 3.9: *Parenthesis problem: (a) Giga updates per second achieved by all algorithms, (b) ratios of total shared and private cache misses, (c) strong scalability with #cores, p when n is fixed at 8192, and (d) ratios of total joule energy consumed by Package (PKG), Power Plane 0 (PP0) and DRAM.*

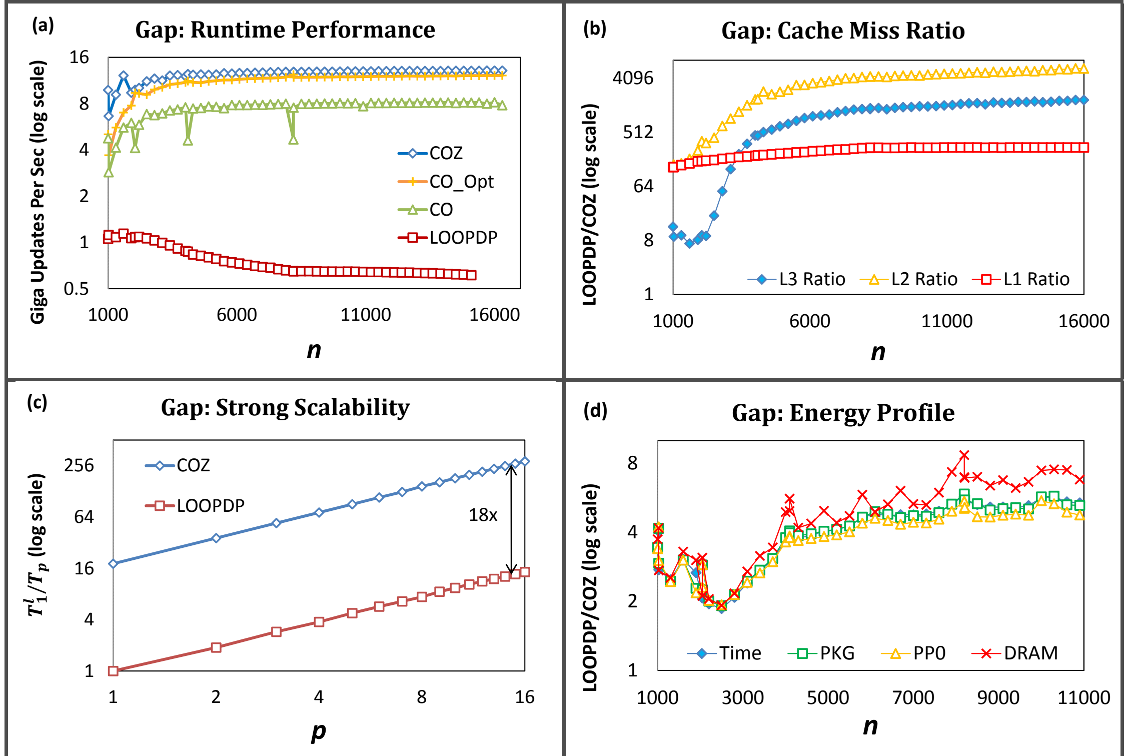


FIGURE 3.10: *Gap problem: (a) Giga updates per second achieved by all algorithms, (b) ratios of total shared and private cache misses, (c) strong scalability with #cores, p when n is fixed at 8192, and (d) ratios of total joule energy consumed by Package (PKG), Power Plane 0 (PP0) and DRAM.*

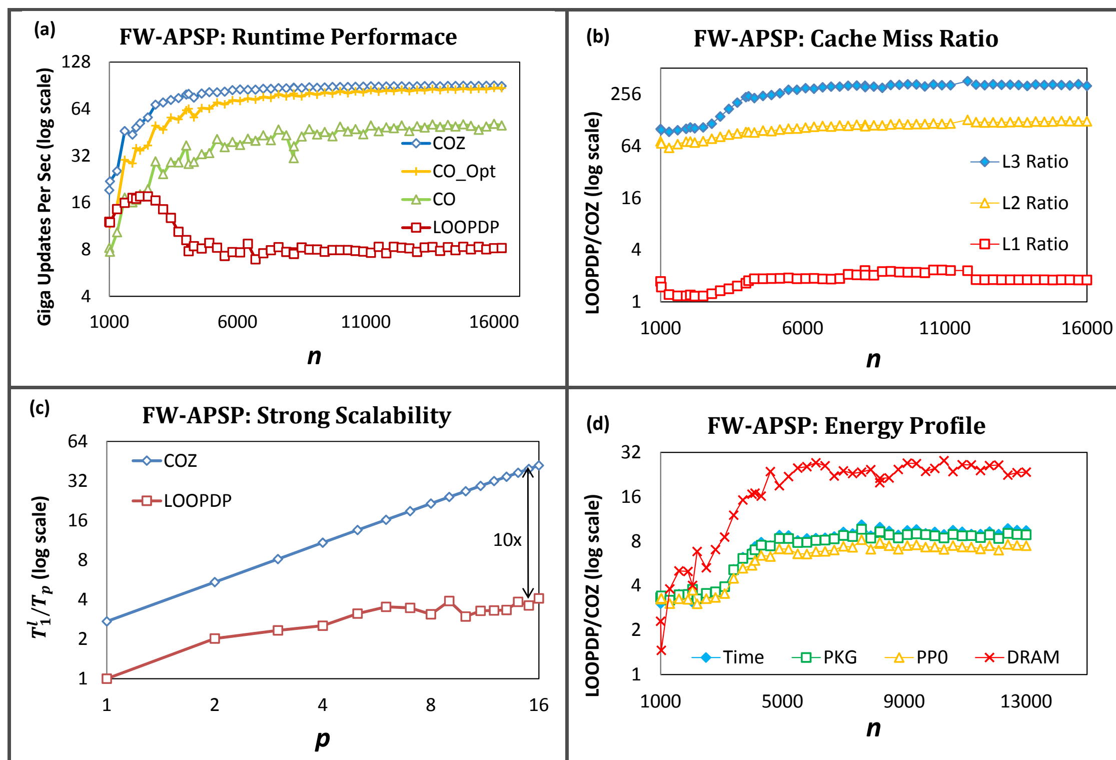


Figure 3.11: *Floyd-Warshall's all pairs shortest path: (a) Giga updates per second achieved by all algorithms, (b) ratios of total shared and private cache misses, (c) strong scalability with #cores, p when n is fixed at 8192, and (d) ratios of total joule energy consumed by Package (PKG), Power Plane 0 (PP0) and DRAM.*

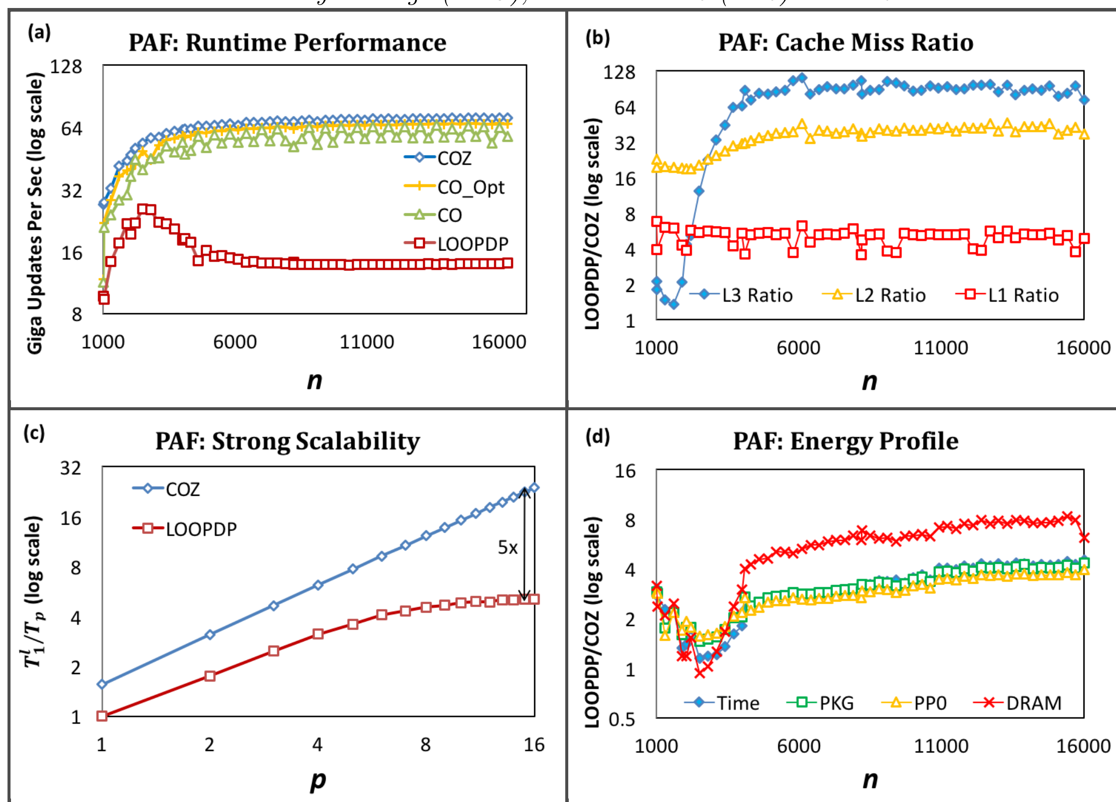


Figure 3.12: *Protein Accordion Folding: (a) Giga updates per second achieved by all algorithms, (b) ratios of total shared and private cache misses, (c) strong scalability with #cores, p when n is fixed at 8192, and (d) ratios of total joule energy consumed by Package (PKG), Power Plane 0 (PP0) and DRAM.*

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	
Parenthesis	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 3.4: *A Summary of the experimental results.*

support this result. Observe that the serial cache complexity of iterative algorithms of the first group (parenthesis and gap) is $\Theta\left(n^3\right)$ and the second group (FW-APSP and PAF) 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.

Figures 3.9 (b), 3.10 (b), 3.11 (b) and 3.12 (b) show that CORDAC algorithms always incur less cache misses in all levels of caches for $n \geq 1000$ which is one of the main contributor in the reduction of running times and energy consumptions. Figures 3.9 (c), 3.10 (c), 3.11 (c) and 3.12 (c) show that CORDAC algorithms scale almost linearly with number of cores, whereas the parallel iterative algorithms sometimes do not. Figures 3.9 (d), 3.10 (d), 3.11 (d) and 3.12 (d) show that, in addition to be faster, CORDAC algorithms consume $4-19\times$ less energy in joule for the entire Package (die), PP0 (cores and their private caches) and DRAM. Fast running time and cache-efficiency have clear contributions in the energy-efficiency of these CORDAC algorithms.

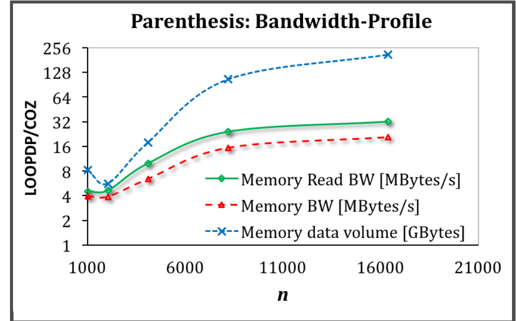


FIGURE 3.13: *Evidence of better bandwidth utilization of CODRAC, wrt. the iterative algorithm (LOOPDP) for the parenthesis/-matrix chain multiplication problem.*

Bandwidth utilization. Since the recursive divide-and-conquer algorithms presented in this chapter are cache-efficient, whereas the corresponding LOOPDP algorithms often aren't, in theory, a LOOPDP algorithm would need to move a lot of data around for its computations compared to the corresponding CORDAC algorithm. As a result, the overall bandwidth utilization of a CORDAC algorithm is likely to be better than that of the corresponding LOOPDP algorithm. Our experimental results show the same. Figure 3.13 shows that for parenthesis problem, LOOPDP's read bandwidth consumption is $32\times$ more, overall (read+write) bandwidth utilization per second

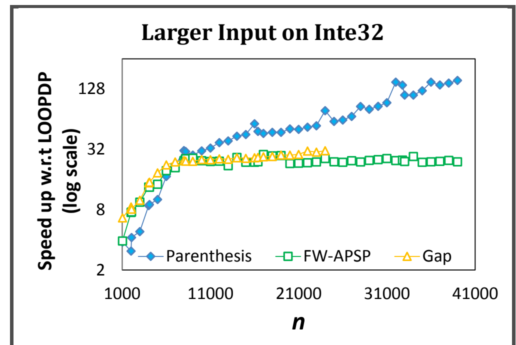


FIGURE 3.14: *Speedup w.r.t LOOPDP with larger input sizes on Intel32 machine.*

is 20× more and total data volume consumed is around 200× more than that of CORDAC. We found similar results for other DP problems presented in this chapter as well.

Results on Larger Input Sizes. In Figure 3.14 we show that speedup of the CORDAC algorithms w.r.t LOOPDP increases with the increase of the number of cores, p and input size, n. This experiment was performed on Intel32. The speedup numbers on Intel32 are almost 2× of what we achieved on Intel16 for $n \leq 16K$. 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. We stopped running Gap problem at $n = 25K$ because the LOOPDP took over 48 hours to finish after that point.

Impact of optimization effort. Figure 3.15 shows impact of various optimizations (i.e., improvement w.r.t. LOOPDP) on running time, CPU and DRAM energy consumption for the parenthesis problem when input size, (n) is 16384 and number of cores (p) is 16. We used Intel16E for this experiment. In each of the plots, the numbers denote the ratio of performance (left: runtime, middle: CPU/package energy, and right: DRAM energy) of an CORDAC implementation with the given optimization/s vs. the performance of the LOOPDP/iterative implementation.

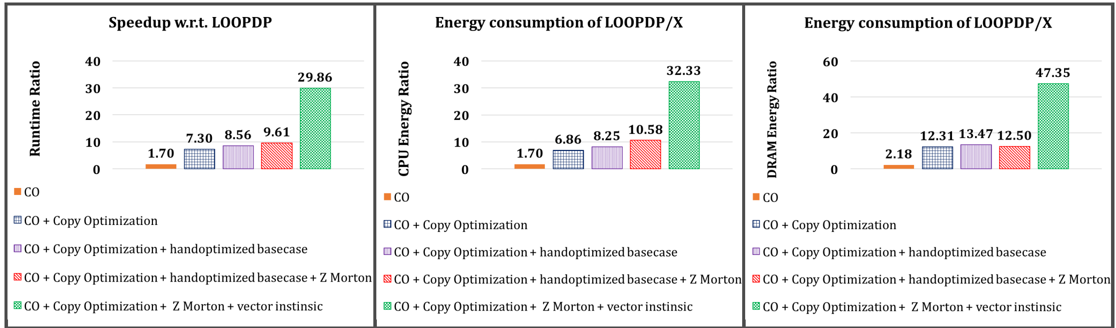


FIGURE 3.15: *Parenthesis Problem: Itemized breakdown of how much performance gain each optimization provides. Here, CO denotes an unoptimized CORDAC algorithm.*

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,

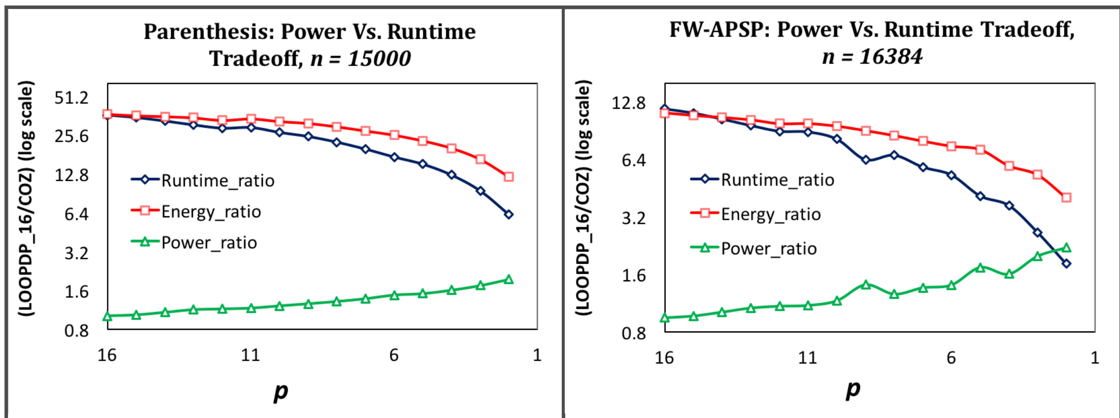


FIGURE 3.16: *Power and Runtime Tradeoff. CORDAC has the flexibility to use fewer number of cores while still running faster but consuming less energy and power than LOOPDP.*

energy consumption also increases with running time. Energy consumption per computation increases as 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 trade-off, 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. Figure 3.16 shows the result. As we decrease p, the 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 number of 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 [29], PoCC [148] and Polly [95]. Figure 3.17 and 3.18 show a 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 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*. We implemented a parallel tiled version for FW-APSP with 2 nested *for loops* and the results are comparable to that of LOOPDP. For parenthesis and gap problems, both our LOOPDP and the polyhedral compiler generated codes used 1 parallel *for loop*, however, the two codes were parallelized differently in each case. None of the generated tiled codes had temporal locality. The tiled code generated for the parenthesis problem does not produce correct result for all intermediate DP cell values, although the final cell value was correct. However, removing weight function $w(i, k, j)$ produces correct values for all intermediate cells. The results reported here includes the weight functions.

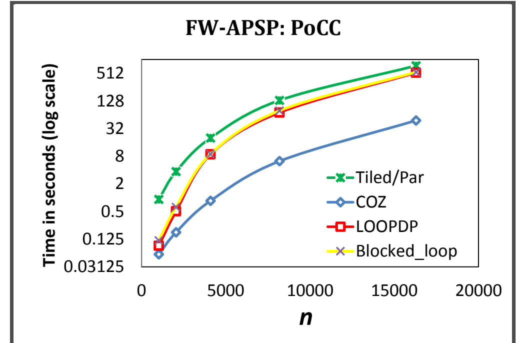


FIGURE 3.17: *FW-APSP: Comparison with parallel tiled code generated using polyhedral compiler PoCC [148].*

3.5.2 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 [169], a pure distributed-memory algorithm has been implemented for the parenthesis problem, where the rows are evenly

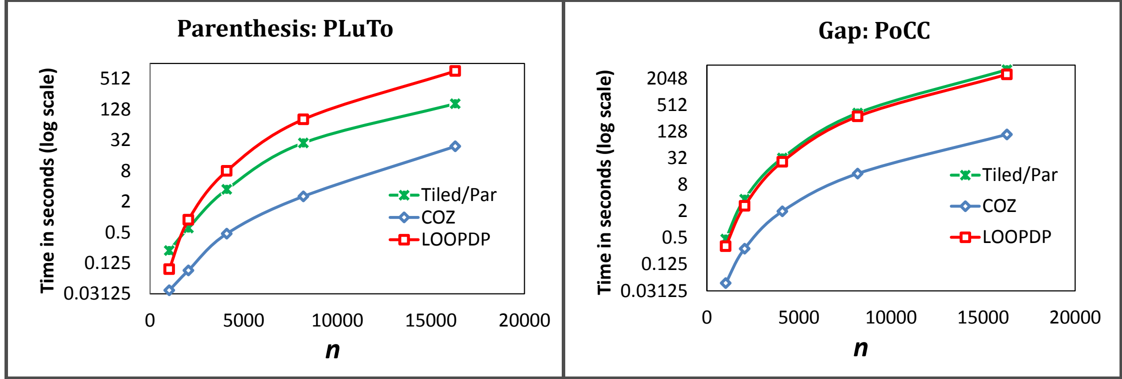


FIGURE 3.18: *Parenthesis and Gap Problems: Comparison with parallel tiled code generated using polyhedral compilers - PLuTo [29], PoCC [148] and Polly [95].*

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 [152] this approach has been used for mergesort algorithm, and the authors concluded that in general, a shared-memory algorithm provides better performance than 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 research work, 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. The pseudocode of this algorithm is shown in Figure 3.19. 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 another thread is holding the lock or 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. Figure 3.19 gives a pseudocode for this algorithm.

```
main function (sdsm)
if (super-master) do
   spawn dispatcher()
   F_A-distributed()
   sync
   done = true
else if  (master) do
   spawn dispatcher()
   waitforsupermaster()
   sync
   done = true
else waitformaster()
```

```
dispatcher()
1) while (!done) do
2)    test for busy workers to return and free the returning worker
3)    if (job q is not empty or there is free worker) do
4)       grab the next job from the queue and assign to the next free worker
```

```
waitformaster()
1) while (true) do
2)    receive the first msg from master and end the loop if master sends -1
3)    receive input data from master
4)    compute the appropriate function on the input
5)    send the result back to the dispatcher
```

Shared-distributed-shared-memory Cache-oblivious Algorithm

FIGURE 3.19: *Shared-Distributed-Shared-Memory (SDSM) framework for recursive divide-and-conquer algorithm with dynamic load-balancing on a cluster of multicores.*

Distribution using 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 [34] or Fox's [79] algorithms for matrix multiplication that have linear scalability and unit efficiency for job distribution, if the flexible kernel looks MM-like (i.e., two input matrices and one output matrix) which is indeed the case for most of our dominating flexible kernels.

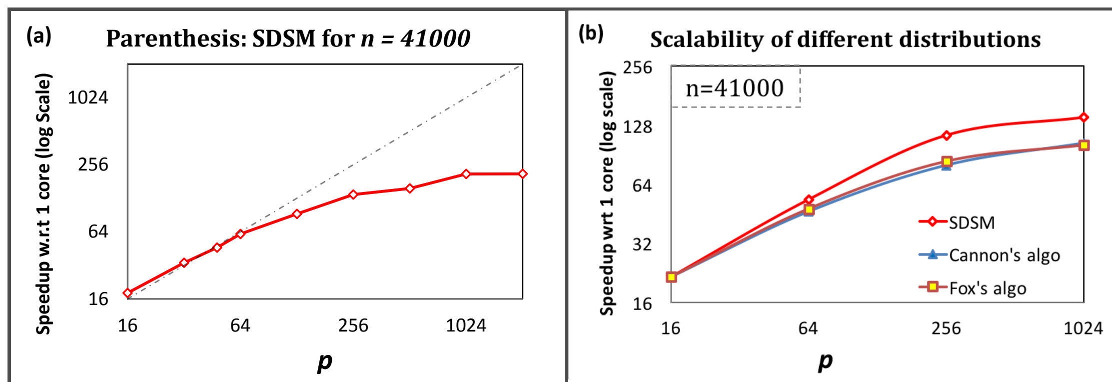


FIGURE 3.20: *Parenthesis Problem: (a) Scalability of shared-distributed-shared-memory algorithm (offloading functions C and B). (b) Performance comparison of different work distribution techniques (offloading C only).*

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.

3.5.3 Shared-Distributed-Shared-Memory Results

We implemented our SDSM algorithm for the parenthesis problem. Figure 3.20(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 parenthesis 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 Figure 3.20(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 Cannons/Fox's algorithms. We found that SDSM performs better than the other two approaches even though it uses a hierarchical dynamic load-balancing strategy.

3.5.4 Communication Complexity

Computing precise communication complexity of our SDSM algorithm is quite involved because of the dynamic load-balancing and interactions with Cilk's randomized work-stealing scheduler. However, deriving an upper bound is fairly straightforward. Table 3.1 shows that each problem generates $O\left((n/b)^3\right)$ subproblems of size $b \times b$ each. If we only solve subproblems of size $(2^s) \times (2^s)$ where $q \le s \le r$ on worker processes, communication complexity will be upper-bounded by $O\left(\sum_{q \le s \le r} \left((n/2^s)^3\right) \times (2^s)^2\right) = O(n^3/2^q)$. 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.

3.6 Conclusions

In this research, we show that there is a class of dynamic programming problems that reduces to MM-like flexible recursive functions and highly optimizable flexible kernels when decomposed using recursive divide and conquer. We present high-performance cache-oblivious recursive divide-and-conquer parallel algorithms for four such problems with important applications in bioinformatics. We show that optimizing the dominating flexible functions/kernels properly is enough to get a significant performance boost for these problems. We also discuss some general optimization strategies that work well for all our recursive dynamic programming algorithms in practice. Our implementations following these optimization strategies achieve more than $5 - 150\times$ speedup over the corresponding standard parallel and optimized loop-based algorithms on modern multicores and manycores. We also show simple extensions of these algorithms to a shared-distributed-shared setting which also demonstrates the versatility and portability of the cache-oblivious recursive divide-and-conquer approach.

Our recent work shows that it is also possible to automatically generate recursive divide-and-conquer (i.e., CORDAC) algorithms for this class of DP problems. However, it is still an open problem to generate CORDAC algorithms for more non-trivial problems. Building a specialized CORDAC compiler that can automatically optimize CORDAC algorithms is an interesting research direction to pursue. The shared-distributed-shared-memory algorithm presented for CORDAC algorithms is processor-aware. Designing a generic processor-oblivious distributed-shared memory algorithm for this class of DP problems is interesting.

Chapter 4

Cache-adaptivity, Bandwidth Benefit and Robustness of Recursive Divide and Conquer

4.1 Abstract

In Chapter 3 we have shown that cache-oblivious recursive divide-and-conquer (CORDAC) algorithms for solving a class of dynamic programming (DP) problems are high-performing in terms of running time, scalability, energy-efficiency and bandwidth performance on multicores, manycores, hybrid multicores+manycores and clusters of multicores. However, all those performance analyses were conducted assuming an execution environment where only one program runs at a time on the entire system, which is often the case for standard production runs. Nevertheless, there are cases where we need to run programs in a multiprogramming environment such as our laptops, smart phones, many web servers and clouds. An interesting question to ask is how the performance of a program is affected by other concurrently running programs sharing the same system resources? In this chapter, we show that our CORDAC algorithms are more robust and adaptive to dynamic fluctuations in shared resources (e.g., cache-space, bandwidth, etc.) in a shared-memory multiprogramming environment compared to their tiled-loop and standard iterative counterparts.

Adaptivity with respect to a particular shared resource means, a program will run as fast as any other program solving the same problem, given a particular resource profile (e.g., size). *Robustness* means if a shared resource fluctuates, a program's performance (time, cache-miss, energy consumption) will be relatively more stable than other programs under the same fluctuation. The performance stability is measured by computing the performance degradation due to fluctuations in resource profile, considering performance with no fluctuation in the total resource capacity as a baseline. Performance of an adaptive and robust algorithm is more predictable making such algorithms more desirable in a multiprogramming or multi-user environment such

as the standard operating system, database systems and compute platforms that support multiple VMs (e.g., cloud). To the best of our knowledge, we present the first empirical results on cache-adaptivity.

4.2 Introduction

In a *multiprogramming environment* multiple programs run concurrently, sharing common resources (e.g., bandwidth, L3 cache space). Typical examples of multiprogramming environments include standard operating systems, OS of smart phones/tablets that allow multiple apps to run concurrently, virtual machine platforms, cloud and database management systems that allow multiple queries to be processed concurrently. In a multiprogramming environment, a program running concurrently with other programs is likely to impact the performance of other programs by kicking out cached data and/or by using a portion of overall system's bandwidth. Therefore, performance in such an execution environment is less predictable.

Having algorithm whose performance remains relatively stable in a multiprogramming environment is useful. We call such algorithms *robust algorithms. Robustness* means the ability of an algorithm to continue operating despite abnormalities. An algorithm is *cache-adaptive* if the program runs as fast as any other program solving the same problem, if some portion of the cache space is taken away. In this research, we show that parallel cache-oblivious recursive divide-and-conquer algorithms for solving a class of dynamic programming problems are more robust, cache-adaptive, energy-efficient and have better bandwidth utilization than their corresponding tiled-loop and iterative implementations.

Dynamic Programming (DP) algorithms are typically implemented using iterative loops. However, iterative implementations of DP algorithms are in general cache-inefficient, and also inflexible in the sense that the loops often cannot be suitably reordered to improve spatial or temporal cache locality and parallelism, while maintaining the correctness of the algorithm. Using tiled- or blocked-loop approach, it is often possible to improve spatial and/or temporal cache locality and parallelism which often leads to better performance compared to the straightforward iterative implementations. However, a tiled-loop approach is cache-aware and the same blocking does not work well on all machines. Blocking efficiently for different cache levels is also complicated on modern multicores with hierarchical caches. The tiled-loop approach lacks portability due to its cache-awareness. A more portable and efficient way to implement DP algorithms is to use a cache-oblivious recursive divide-and-conquer (CORDAC) technique which often achieves optimal cache-performance without sacrificing portability by being cache-oblivious.

Cache-oblivious recursive divide and conquer is an algorithmic technique that divides the problem into manageable pieces recursively until it reaches a basecase size large enough to amortize the cost of recursion. After that, it solves those small problems and then combines those solutions to get the solution for the original problem. An algorithm is cache-oblivious if it does not use knowledge of cache-parameters in the algorithm description. For example, standard iterative and CORDAC algorithms for DP problems are cache-oblivious. An algorithm is cache-aware if it uses knowledge of cache-parameters (e.g, block size of a cache) in the algorithm description. In general, a tiled-loop algorithm is cache-aware since it uses the cache-block sizes to block the loops.

Figure 4.1 shows code snippets for iterative, tiled-loop and CORDAC algorithms for the Longest Common Subsequence (LCS) problem. In this figure, the iterative code does not use any cache

```
Iterative_LCS(X, S, T, n, xi, xj) {
// upper triangle
  for (t = 0; t < n; t++){
    parallel_for(i = 0; i <= t;i++){
      j = t - i;
      idx = N+(xj+j–xi-i);
      X[idx]=
      MAX(
      X[idx] + (S[xi+i] != T[xj+j]),
      X[idx + 1],
      X[idx - 1]
      );
    }
  }// similarly compute lower triangle
}
```

```
Tiled_LCS(X, S, T, n, tile_size) {
 // upper triangle
 num_tile = n/tile_size

 for (int t = 0; t < num_tile; t++){
  parallel_for(i = 0; i <= t;i++){
   j = t - i;
   xi = i*tile_size;
   xj = j*tile_size;
     Iterative_LCS(X, S, T,
         tile_size, xi, xj);
   }
 }// similarly compute lower triangle
}
```

```
CORDAC_LCS(X, xi, xj, n) {
if(n<=base)
  Iterative_LCS(X, S, T, n, xi, xj);
else {
  nn = n >> 1;
  //X11
  CORDAC_LCS (X, xi, xj, nn);
  //X21 and X12
  spawn CORDAC_LCS(xi, xj+nn, nn);
  CORDAC_LCS(X, xi+nn, xj, nn);
  sync
  //X22
  CORDAC_LCS (xi+nn, xj+nn, nn);
  }
}
```

FIGURE 4.1: *Code snippet for Tiled, CORDAC and Iterative algorithms.*

parameter. The tile_size in the tiled-loop algorithm is chosen in such a way that all the input and output sub-matrices to compute values for a tile fit into the desired level of cache. On the other hand, the base parameter in the CORDAC algorithm is chosen in such a way that the cost of computation inside the basecase becomes large enough to subsume the overhead of recursion.

Contributions. The relationships between *i)* cache-misses and energy-consumption; *ii)* cache-obliviousness, cache-optimality, and cache-adaptivity; and *iii)* cache-efficiency and bandwidth-utilization of different algorithmic options (iterative, tiled-loop and CORDAC) for solving DP problems are not yet known. Understanding these relationships is very important in deciding which algorithm to use in practice. It has been shown in theory for a few recursive divide-and-conquer algorithms that they are cache-adaptive [21]. However, no empirical result has been shown yet. To the best of our knowledge, we present the first empirical results to unravel some of those relationships in a multiprogramming setting by demonstrating adaptivity and robustness of recursive divide-and-conquer algorithms.

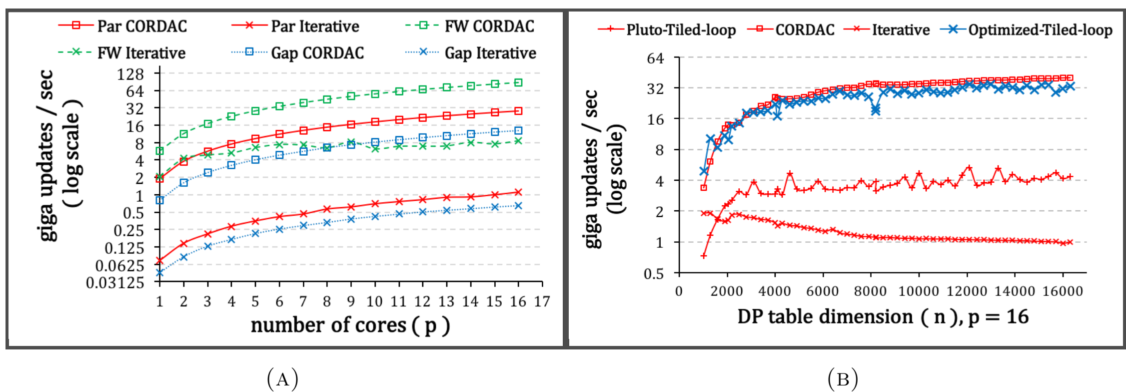
4.3 Adaptivity and Robustness

We consider an algorithm, $\mathcal{A}$ to be more robust than another algorithm, $\mathcal{B}$ for solving the same problem if $\mathcal{A}$'s performance (runtime, cache miss, energy, bandwidth, etc.) is less degraded in a multiprogramming environment compared to $\mathcal{B}$. We show that CORDAC algorithms for solving dynamic programming problems are more robust compared to the corresponding tiled-loop implementations auto-generated from the state-of-the-art polyhedral compilers Pluto [29], Pocc [148], and Polly [95] in a shared-memory multiprogramming environment. In such an environment when different independent processes fight for the shared cache-space, a CORDAC implementation is more likely to adapt to less cache-space than a tiled implementation which is tiled for a particular tile size and works iteratively inside a tile. As other programs start to kick out cache blocks being used by a tiled implementation, it can not adapt to the loss of cached data efficiently. As a result, its performance is likely to get hampered noticeably. On the other hand, a CORDAC implementation works recursively and keeps working on the same recursive portion until it is completely done with that portion before moving to other parts of the matrix.

In recursion, the implementation only goes deeper into that portion and even if its least recently used data is kicked out from the cache by the LRU cache replacement policy, it will still have the required portion of the data that it has already brought into the cache. Since data is accessed in a recursive fashion, even if a bigger block is kicked out from an upper level of cache, a recursive smaller block of data will be available in a lower cache which it can still utilize fully during computation. Therefore, performance will be less hampered. Although the iterative code is cache-oblivious, it is cache-inefficient (no temporal locality). Because of that it incurs orders of magnitude more cache misses than a cache-efficient implementation (e.g., CORDAC). Although its cache miss does not increase that much with the reduction in cache space, an iterative implementation saturates the bandwidth very quickly due to its cache-inefficiency, which slows it down a lot.

4.4 Experimental Results

In this section we mainly focus on the CORDAC algorithms for the parenthesis, gap and Floyd-Warshall's all pairs shortest paths (APSP) problems from Chapter 3. For each of these problems, we show performance degradation of CORDAC, tiled and iterative implementations when multiple program instances run simultaneously compared to when they run alone (without any other programs running in the system). Figure 4.2a shows that the cache-oblivious recursive-divide-and-conquer algorithms for the parenthesis, gap, and Floyd-Warshall's APSP problems significantly outperform their iterative counterparts when run on a 16 core Intel Xeon machine with no other program running except the program being timed. Figure 4.2b shows that CORDAC for parenthesis problem outperforms tiled-loop version generated using state-of-the-art polyhedral compilers PLuTo [29], PoCC [148] and Polly [95]. The tile size was optimized for 16 core run. Please review Chapter 3 for more results.



(A) (B)

FIGURE 4.2: *(A) Rates of updates performed by multithreaded iterative and recursive (CORDAC) algorithms for the parenthesis problem, Floyd-Warshall's APSP [50], and the Gap problem [39] on $2^{13} \times 2^{13}$ integer matrices, as the number of processing cores varies. (B) Rates of updates performed by multithreaded iterative, recursive and tiled-loop code generated by PLuTo [29] for the parenthesis problem as the matrix dimension varies. The Optimized-Tiled-loop is a hand-optimized version of the tiled code auto-generated by PLuTo.*

Next we show how the performance changes in a multiprogramming environment. We always use a less optimized version of the CORDAC implementations (CO_Opt without hand-optimizing the basecases from Chapter 3) for the results presented in this section.

Experimental setup. We used a dual-socket machine with 8-cores per socket ($2 \times 8 = 16$-cores total) and 2 GHz Intel Sandy Bridge processors with 32 GB RAM to conduct all experiments presented in this section. Each core was connected to a 32 KB private L1 and a 256 KB private L2 cache. All cores in a socket shared a 20 MB 10-way L3 cache.

We used C++ with Intel®Cilk™ Plus extension to implement all algorithms and compiled with icc v13.0 compiler. All programs were compiled with `-O3 -ip -parallel -AVX -xhost` optimization parameters. We used PAPI 5.2 [4] to count cache misses, and LIKWID [184] for energy and bandwidth measurements. Energy and bandwidth measurements were end-to-end capturing everything from the start to the end of the program.

4.4.1 Robustness

We performed the following experiment to show how the performance of a program (CORDAC, iterative and tiled code) changes if multiple copies of the same program are run on the same CPU. We ran up to 4 instances of the same program on an 8-core Sandy Bridge processor with 2 threads (i.e., cores) per program/instance/process. The block size of the tiled code was optimized for best performance with 2 threads.

Parenthesis Problem. Figure 4.3 shows that with 4 concurrent processes iterative implementation slowed down by 82% and tiled code by 46%, but CORDAC implementation lost only 16% of its performance. The slowdown of tiled code resulted from its inability to adapt to the loss in the shared cache space (L3 misses increased by $> 5\times$), whereas L3 misses incurred by CORDAC implementation increased by less than $2.5\times$. Since the iterative implementation does not have any temporal locality, loss of cache space did not significantly change its L3 misses. However, iterative implementation already incurred $90\times$ more L3 misses than CORDAC implementation, and with 4 such concurrent processes the burden on the DRAM bandwidth increased considerably (1126 GB total whereas for tiled-loop it was 190 GB and for CORDAC it was only 14 GB) causing significant slowdown of the program.

Figure 4.3 also demonstrates changes in energy consumption of each process as the number of concurrent processes increases. Energy values were measured using `likwid-perfctr` (included in LIKWID) which reads them from the MSR registers. Three types of energy were measured: `package energy` (PKG), which is the energy consumed by the entire processor die, `PP0 energy`, which is the energy consumed by all cores and their private caches, and `DRAM energy`, which is the energy consumed by the directly-attached DRAM. We omitted PP0 energy plots because they almost always follow the same pattern as the package energy. A single instance of tiled code consumed 39% more package energy than a CORDAC program instance while iterative implementation consumed $14\times$ more energy. Average package and PP0 energy consumed by the tiled code increased at a faster rate than that by the CORDAC implementation as the number of processes increased. This happened because both its running time and L3 performance degraded faster than the CORDAC implementation, both of which contributed to the faster increase in

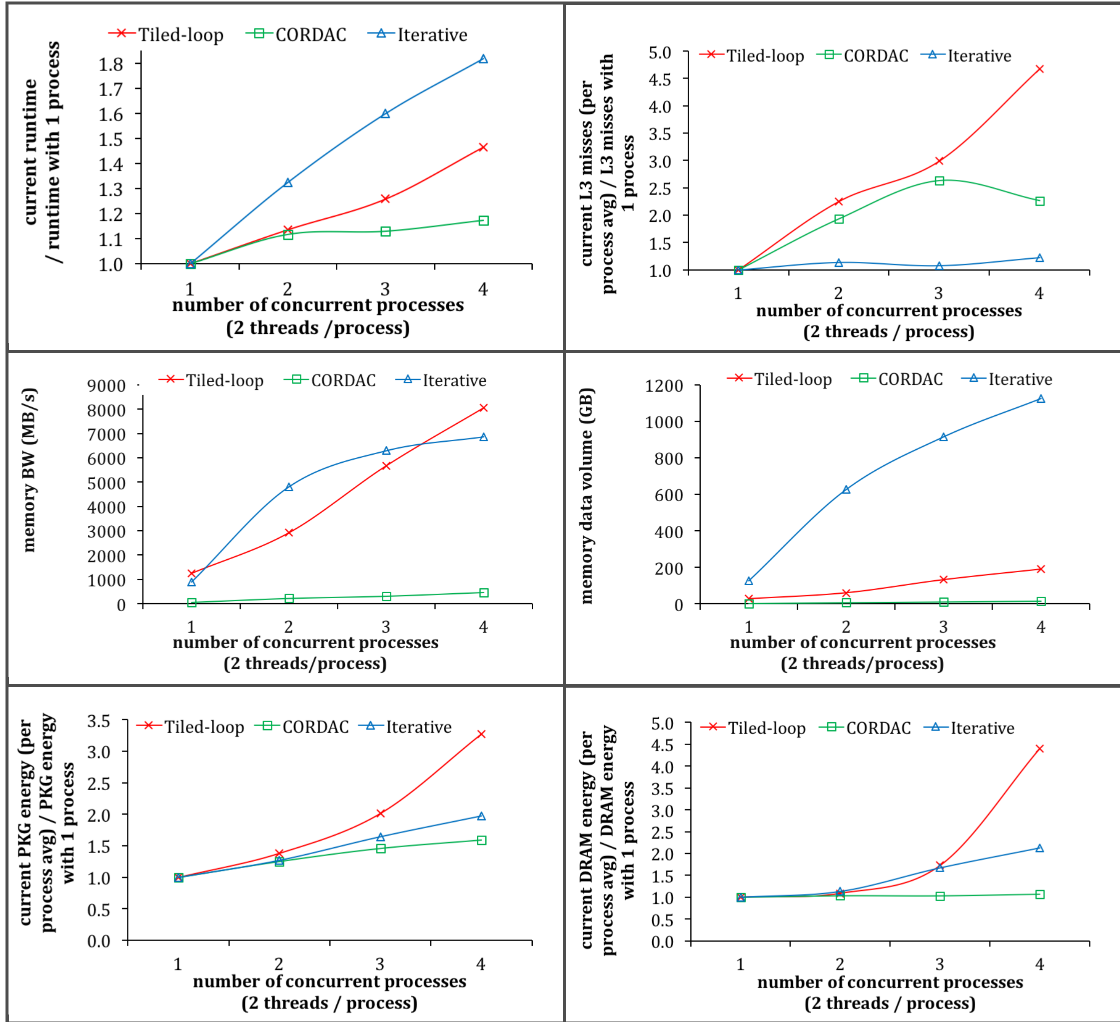


FIGURE 4.3: *The plots show how the performances of standard iterative, tiled-loop and recursive codes for solving the parenthesis problem (for $n = 2^{13}$) are affected when multiple instances of the same program are run on an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache.*

energy consumption. However, since for iterative implementation L3 misses did not change much with the increase in the number of processes, its package energy consumption increased at a slower rate compared to that of tiled.

Floyd-Warshall's APSP. Figure 4.4 shows the robustness of CORDAC compared to iterative and tiled-loop implementations for Floyd-Warshall's APSP. The auto-generated tiled-loop code by PLuTo [29] had only one parallel loop with no temporal locality. Therefore, we wrote our own tiled version that had two parallel loops for these experiments. However, this tiled implementation also did not have any temporal locality. In fact, it is non-trivial to generate efficient tiled code with temporal locality for FW-APSP.

For FW-APSP, with 4 concurrent processes both iterative and tiled implementations slowed down by over 3×, but CORDAC implementation lost only 16% of its performance. L3 misses incurred by CORDAC implementation increased by less than a factor of 3. However, since tiled-loop and iterative implementations do not have any temporal locality, loss of cache space did

not significantly change the number of L3 misses they incurred. The iterative implementation incurred 780× and tiled-loop incurred 176× more L3 misses than CORDAC implementation, and with 4 such concurrent processes the burden on the DRAM bandwidth increased considerably causing significant slowdown of these two programs. Bandwidth consumed per second increased by 36× for both iterative and tiled implementations, whereas for CORDAC implementation the increase was 16× with 4 concurrent processes. Total data volume consumed by iterative implementation was 182 GB, for tiled-loop it was 76 GB and for CORDAC, the required data volume was only 14 GB.

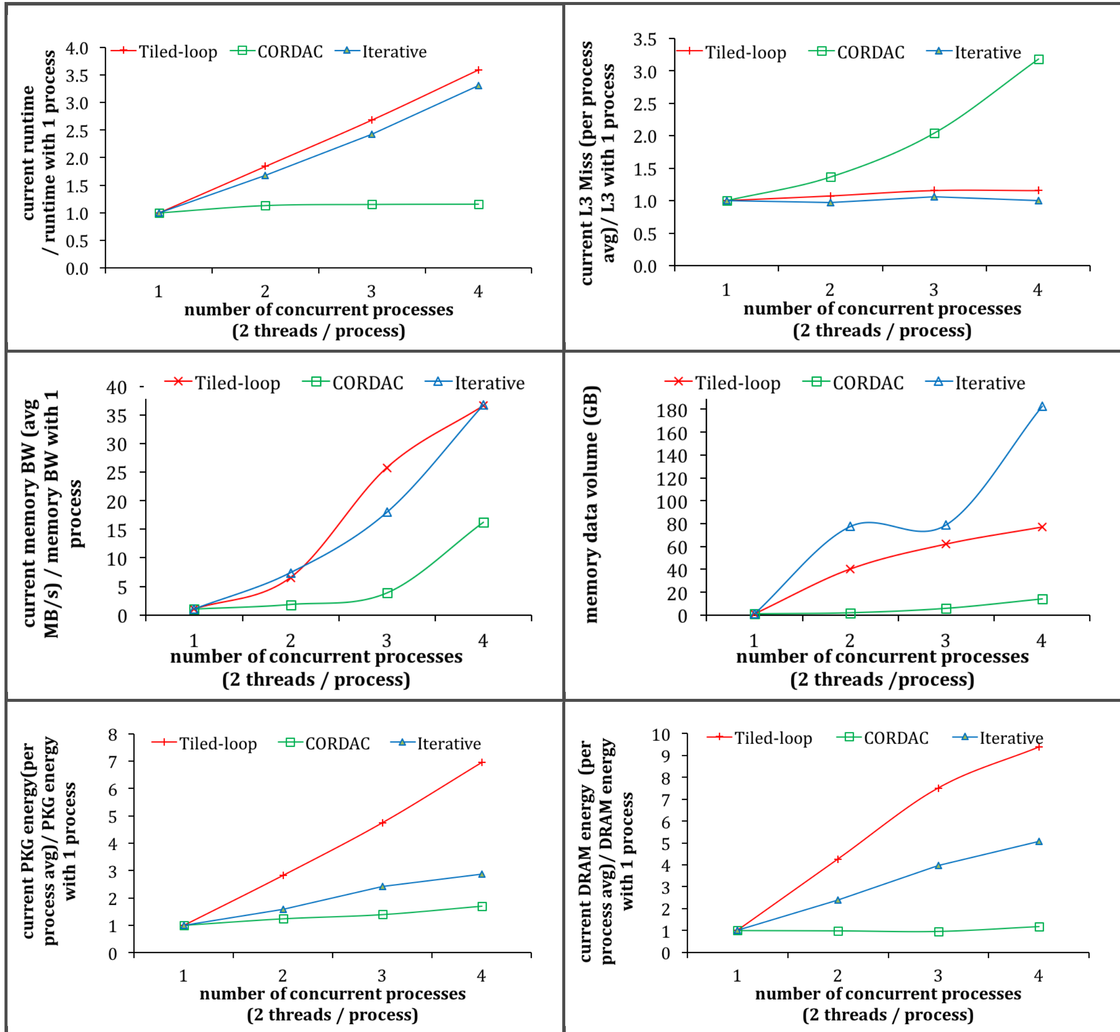


FIGURE 4.4: *The plots show how the performances of standard-iterative, tiled-loop and recursive codes for solving the Floyd-Warshall APSP problem (for $n = 2^{12}$) are affected as multiple instances of the same program are run on an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache.*

Gap Problem. For the Gap problem (see Figure 4.5) with 4 concurrent processes, the iterative implementation slowed down by 1.4×, tiled code slowed down by 1.5×, and CORDAC implementation lost 16% of its performance. L3 misses for tiled code increased by a factor of 140. On the other hand, L3 misses for CORDAC implementation increased by a factor of 2. Since iterative implementation does not have any temporal locality, loss of cache space did not

significantly change the number of L3 misses it incurred. The package energy for tiled code increased by 1.67×, for iterative by 2.22×, and for CORDAC by 1.69×. DRAM energy of CORDAC increased by 7%, whereas for tiled it increased by 12%, although the actual DRAM energy consumption of tiled-loop was already 2× more than the CORDAC implementation. DRAM energy of iterative implementation increased by 42%. The memory bandwidth requirement per second increased by 37× for the tiled-loop implementation, whereas for CORDAC and iterative, they increased by 21×. Total data volume (for all 4 instances) transferred by iterative was 2000 GB with 4 concurrent processes, for tiled-loop it was 22 GB, whereas, for CORDAC, it was 15 GB.

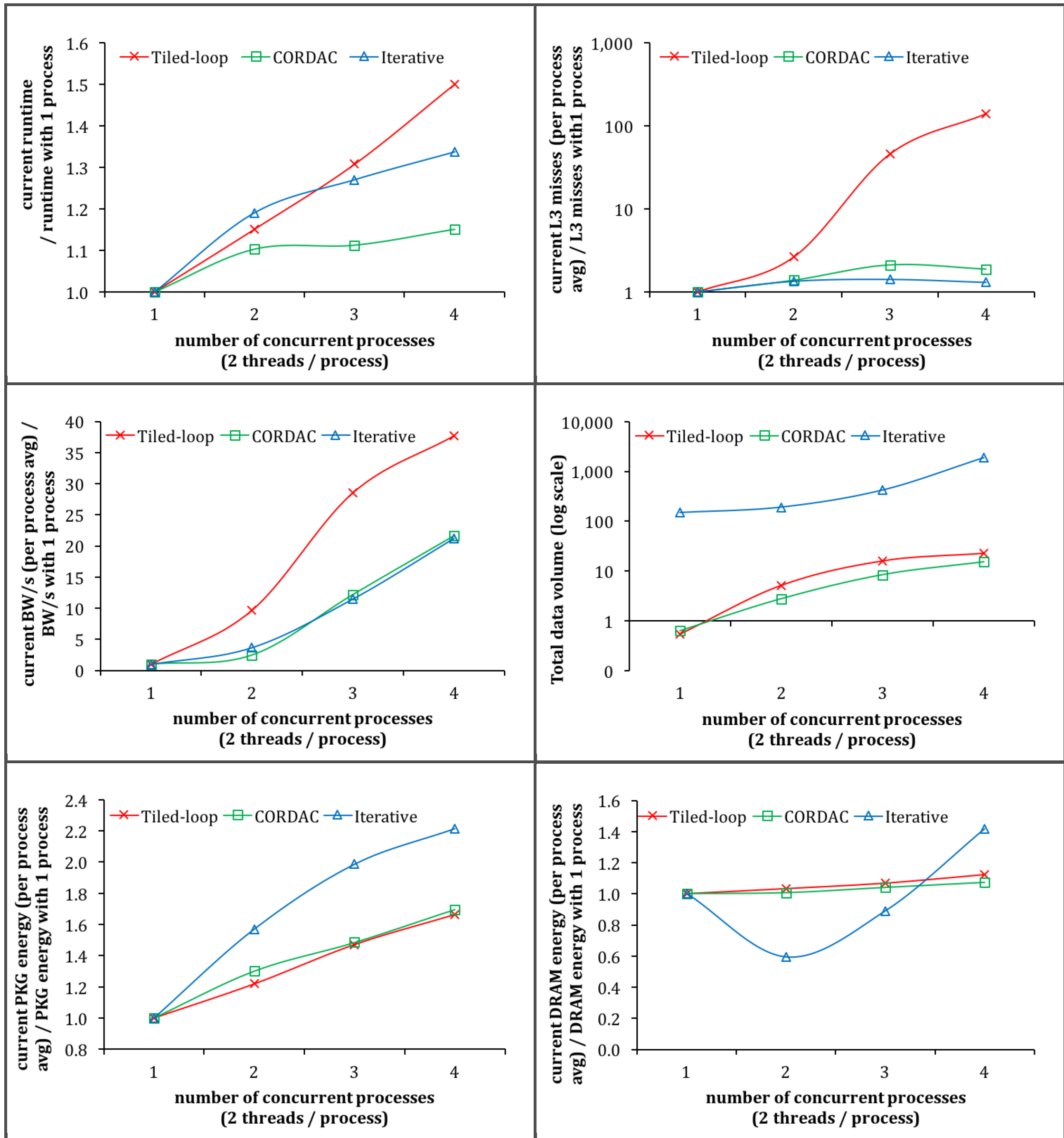


FIGURE 4.5: *The plots show how the performances of standard iterative, tiled-loop and recursive codes for solving the Gap problem (for $n = 2^{13}$) are affected as multiple instances of the same program are run on an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache.*

All these results show how robust and adaptive CORDAC algorithms are compared to the tiled

and iterative algorithms. Therefore, we can conclude that overall degradation in performance in response to dynamic fluctuations of resources is the least in CORDAC, making them the most preferable choice in a multiprogramming environment.

4.4.2 Cache-adaptivity

We performed another set of experiments to measure cache-adaptivity of CORDAC and the corresponding tiled implementations. For that we measured the effect on the running times and L3 cache misses of serial CORDAC and tiled code[1] when the available shared L3 cache space is reduced. The `Cache Pirate` tool [1, 70] was used to steal cache space[2].

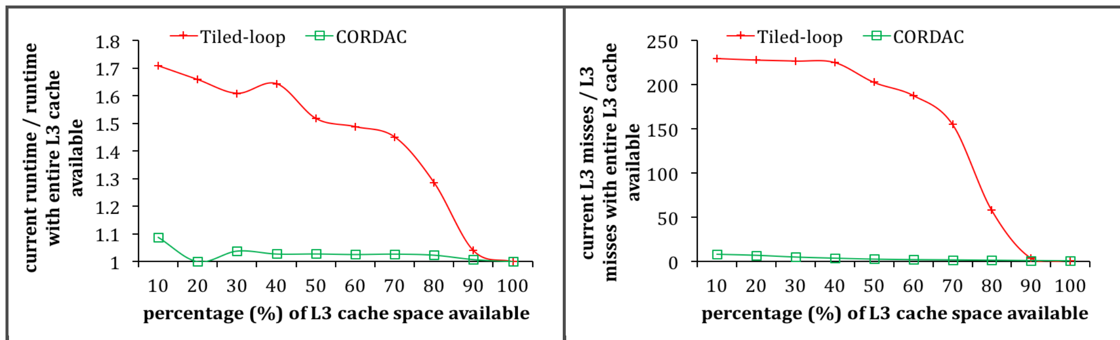


FIGURE 4.6: *The plots show how changes in the available shared L3 cache space affect the serial running time and the number of L3 cache misses of tiled-loop and recursive (CORDAC) algorithms for solving the parenthesis problem when $n = 2^{13}$. The code under test was run on a single core of an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. A multithreaded* `Cache Pirate` *[1, 70] was run on the remaining 7 cores to steal L3 cache space.*

Figure 4.6 shows that when the available cache space was reduced to 50%, the number of L3 misses incurred by the tiled code increased by a factor of 22, but for CORDAC, the increase was only 17%. As a result, the tiled code slowed down by almost 50% while for CORDAC the slowdown was less than 3%. Thus CORDAC automatically adapts to cache sharing [21], but the tiled code does not.

4.5 Conclusion

We show that the benefits of cache-oblivious recursive divide-and-conquer algorithms are two-fold: they are high-performing, and their performances remain relatively stable (robustness property) in a multiprogramming environment. Based on the adaptivity and robustness results of CORDAC algorithms, we can hope that these recursive divide-and-conquer algorithms are going to be the most preferable choice for implementing DP algorithms in near future instead of the current trends of using tiled-loop or iterative algorithms, especially in a multiprogramming environment, such as the standard and mobile OS, database, virtual machine systems and cloud settings. Even in case of a single programming environment (where only one program runs

[1] with tile size optimized for best serial performance
[2] `Cache Pirate` allows only a single (serial) program to run, and does not reduce bandwidth.

at a time), if the OS is allowed to dynamically change resources (such as turning on power boost, turning off cores and caches to save energy, or migrate threads or even data), CORDAC algorithms should be a better choice due to their robustness and adaptivity.

Questions that still need to be answered about adaptivity and performance tradeoff are: a) how the cache-adaptivity changes based on overall parallelism, space usage and cache-complexity of a program? b) what are the relationships among energy consumption, cache-miss, bandwidth and running time? Can they be represented as simple equations? One way to answer these questions is to conduct rigorous experiments, and then use the results to conduct statistical analyses to find those relationships. However, results of such analyses are going to be biased toward the machines and parallel platforms being used. Nevertheless, we want to have answers for these questions in future.

Chapter 5

Provably Efficient Scheduling of Cache-Oblivious Recursive Wavefront Algorithms

5.1 Abstract

In the previous chapters we have shown that cache-oblivious recursive divide-and-conquer (CORDAC) algorithms for solving dynamic programming (DP) problems are high-performing in a single programming environment (i.e., an execution environment where only one program is run at a time on the entire system), and their performance also remain relatively stable, robust and adaptive in a multiprogramming environment (i.e., where multiple programs run concurrently and share system's resources). In this chapter, we show that it is possible to improve parallelism of those CORDAC algorithms even further without loosing their cache-optimality, by transforming them to cache-oblivious wavefront algorithms. Our results demonstrate that CORDAC algorithms are not only high-performing on today's state-of-the-art (multicore) architectures, but also, can be made high-performing on future architectures with many more cores.

Iterative wavefront algorithms for evaluating dynamic programming recurrences exploit optimal parallelism but show poor cache performance. Tiled-loop wavefront algorithms can achieve optimal cache performance and high parallelism but are cache-aware and hence are not portable and not cache-adaptive. On the other hand, standard CORDAC algorithms have optimal serial cache complexity but often have low parallelism due to *artificial dependencies* among subtasks introduced by the recursive structure of the algorithm. Very recently we introduced cache-oblivious recursive wavefront (COW) algorithms that do not have any artificial dependencies, but are too complicated to develop, analyze, implement and generalize. Good theoretical performance guarantees of those COW algorithms often do not translate into good practical performance due to high overhead in implementation.

In this work[1], we show how to systematically transform standard cache-oblivious recursive divide-and-conquer algorithms into recursive wavefront algorithms (i.e., CORDAC to COW) to achieve optimal parallel cache complexity and high parallelism with negligible implementation overhead. We use closed-form formulas to compute at what time each divide-and-conquer function must be launched in order to achieve high parallelism without losing cache performance. The resulting implementations are arguably much simpler than implementations of known COW algorithms. We present theoretical analyses and experimental performance and scalability results showing the superiority of these new algorithms over the existing ones.

5.2 Introduction

Dynamic programming (DP) is a popular algorithm design technique to solve optimization problems that exhibit the overlapping subproblems and optimal substructure properties. The process involves dividing a problem into smaller subproblems, solving them, storing their results in a DP table to avoid recomputations, and combining those solutions. DP is used in many real-world application areas, and extensively in computational biology [16, 67, 97, 189]. This motivates us to develop a general framework for high-performance and scalable algorithms to solve many DP problems.

For good performance on a modern multicore machine with a cache hierarchy, algorithms must have ample parallelism and should be able to use the caches efficiently at the same time. Iterative wavefront algorithms for solving DP problems have optimal parallelism but often suffer due to bad cache performance. On the other hand, though standard cache-oblivious [81] recursive divide-and-conquer (CORDAC) DP algorithms have optimal serial cache complexity, they often have non-optimal parallelism. The tiled-loop wavefront algorithms achieve optimality in cache complexity and can achieve high parallelism but are cache-aware, and hence are not portable and do not adapt well when available cache space fluctuates during execution in a multiprogramming environment (see Chapter 4)). Very recently, the *cache-oblivious wavefront* algorithms have been proposed that have optimal parallelism and optimal serial cache complexity, but no bound on parallel cache complexity has been proved [173]. Those algorithms are also very difficult to implement and analyze since they require hacking into a parallel runtime system and use atomic locks. Extensive use of atomic locks causes too much overhead for very large and higher dimensional DPs.

In this chapter, we present a provably efficient method for scheduling cache-oblivious recursive divide-and-conquer wavefront algorithms on a multicore machine which optimizes parallel cache complexity and achieves high (near optimal) parallelism. Our algorithms are also arguably simpler to implement and analyze.

Performance of a parallel program on multicores. We analyze the performance of a parallel program run on a shared-memory multicore machine using the *work-span* model [56]. The *work* of a multithreaded program, denoted by $T_1(n)$, where n is the input parameter, is the total number of CPU operations performed when run on a single processor[2]. The *span* (a.k.a. critical-path length or depth), denoted by $T_\infty(n)$, is the maximum number of operations performed on

[1] Jesmin Jahan Tithi and Pramod Ganapathi are equal contributors in this work.

[2] unless specified otherwise, we will use "processor" and "processing core" synonymously

any processor when the program is run on an infinite number of processors. The *parallel running time* $T_p(n)$ of a program when scheduled by a greedy scheduler [30] on p processors is given by $T_p(n) = \mathcal{O}\left(\frac{T_1(n)}{p} + T_\infty(n)\right)$. The *parallelism*, computed by the ratio of $T_1(n)$ and $T_\infty(n)$, is the average amount of work done per step of the critical path.

Cache complexity is a performance metric that counts number of block transfers (or cache misses or I/O transfers or page faults) triggered by a program between adjacent levels of caches in a memory hierarchy. By Q_p we denote the total number of cache misses on a p-processor machine. So Q_1 is the *serial cache complexity.* We say that an algorithm has *spatial locality* provided each cache block it brings into a cache contains as much useful data as possible. We say that it has *temporal locality* provided it performs as much useful work as possible on each cache block it brings into a cache before the block gets evicted from the cache.

Iterative algorithms. Traditionally, DP algorithms are implemented using a series of (nested) loops and they can be parallelized easily. These algorithms often have good spatial locality, no temporal locality, and standard implementations may not have optimal parallelism as well. For example, an iterative algorithm for the parenthesis problem (explained in Section 5.3) has $T_\infty(n) = \Theta\left(n^2\right)$ and $Q_1(n) = \Theta\left(n^3\right)$.

Iterative algorithms are also implemented as tiled loops, in which case the entire DP table is blocked or tiled and the tiles are executed iteratively. For example, for a tiled iterative algorithm for the parenthesis problem with $r \times r$ tile size, where $r \in [2, n]$, we have $T_\infty(n) = \Theta\left((n/r)^2\right) \cdot \Theta\left(r^2\right) = \Theta\left(n^2\right)$, and $Q_1(n, r) = (n/r)^3 \cdot \mathcal{O}\left(r^2/B + r\right)$
$= \mathcal{O}\left(n^3/(rB) + n^3/r^2\right)$.

Fastest iterative DP implementations have the following wavefront-like property. Let a single update on a cell x in the DP table needs to be done by reading from the cells $\langle y_1, y_2, \ldots, y_s\rangle$. When the cells $y_1, y_2, \ldots, y_s$ are completely updated, then the cell x can immediately get updated, either partially or fully. For example, for the parenthesis problem, the fastest iterative wavefront algorithm has a span $T_\infty(n) = \Theta(n \log n)$, but the worst possible cache complexity $Q_1(n) = \mathcal{O}\left(n^3\right)$.

Recursive algorithms. Cache-oblivious recursive divide-and-conquer (CORDAC) parallel DP algorithms can overcome many of the limitations of their iterative counterparts. While iterative algorithms often have poor or no temporal locality, recursive algorithms have excellent and often optimal temporal locality. One problem with recursive divide-and-conquer algorithms is that they trade off parallelism for cache optimality, and thus may end up with suboptimal parallelism.

For example, a 2-way CORDAC algorithm (where, each dimension of the subtask will be half the dimension of its parent task) for the parenthesis problem has $T_\infty(n) = \Theta\left(n^{\log_2 3}\right)$ and $Q_1(n) = \Theta\left(n^3/(B\sqrt{M})\right)$, that is, it has optimal serial cache complexity but non-optimal span. For n-way CORDAC algorithm, $T_\infty(n) = \Theta(n \log n)$ and $Q_1(n) = \mathcal{O}\left(n^3\right)$. This time, the algorithm has optimal span but worse serial cache complexity. Ideally we want to have a balance between cache complexity and span by choosing r-way CORDAC algorithm in which case both the span and the parallel cache complexity will be non-optimal, however will have best practical

performance.

Source of suboptimal parallelism in recursive algorithms. The suboptimal parallelism in 2-way CORDAC algorithms results from artificial dependencies among subproblems that are not implied by the underlying DP recurrence [173]. A 2-way CORDAC algorithm for the LCS problem splits the DP table X into four equal quadrants: X_{11} (top-left), X_{12} (top-right), X_{21} (bottom-left), and X_{22} (bottom-right). It then recursively computes the quadrants in the following order: X_{11} first, then X_{12} and X_{21} in parallel, and finally X_{22}. As per the recursive structure, the top-left quadrant of X_{12} and X_{21} i.e., $X_{12,11}$ and $X_{21,11}$, respectively, can only start executing when the execution of the bottom-right quadrant of X_{11} i.e., $X_{11,22}$ completes. This dependency between subproblems or subtasks is not defined by the underlying DP recurrence but defined by the recursive structure of the algorithm. Such dependencies in a recursive algorithm are called *artificial dependencies*. There are artificial dependencies at several different granularities. Most often, these artificial dependencies asymptotically increase the span thereby reduce parallelism.

Recursive wavefront algorithms. By removing artificial dependencies from the recursive (CORDAC) algorithms, it is possible to develop algorithms that simultaneously achieve parallel cache-optimality, near-optimal parallelism, and cache-obliviousness. Such algorithms are called *recursive wavefront* (or *cache-oblivious wavefront*) *algorithms*.

We introduced recursive wavefront algorithms in [173]. However, those algorithms (also called `COW` algorithms) are too complicated to develop, analyze, implement, and generalize. Atomic instructions were used extensively to identify and launch ready tasks, and implementations required hacking into Cilk$^{\mathrm{TM}}$'s runtime system. No bounds on parallel cache complexities of those algorithms are known.

In this chapter, we present a generic method to schedule recursive wavefront algorithms based on timing functions. These algorithms have a structure similar to the standard recursive divide-and-conquer (CORDAC) algorithms, but each recursive function call is annotated with start-time and end-time hints that are passed to the scheduler. The task scheduler will make sure that the algorithms are executed in a wavefront fashion using the timing functions. Indeed, the transformation the scheduler is expected to do based on the timing functions is straightforward, and a programmer may choose to do that herself and use a scheduler that do not accept hints (e.g., cilk's work-stealing scheduler). The transformed code is still purely based on fork-join parallelism, and the performance bounds (e.g., parallel running time and parallel cache complexity) guaranteed by any scheduler supporting fork-join parallelism apply. The recursive wavefront algorithm for the parenthesis problem has $T_\infty(n) = \Theta(n \log n)$ and $Q_1(n) = \mathcal{O}\left(n^3/(B\sqrt{M})\right)$. The bounds on T_p and Q_p can be obtained from the scheduler guarantees.

Related work. The tiled-loop algorithms [60, 90, 142, 154, 157, 191] have been studied extensively as tiling is the traditional way of implementing dynamic programming and other matrix algorithms. There are several frameworks to automatically produce tiled codes such as PLuTo [29], Polly [95], and PoCC [148]. However, these softwares are not designed to generate correct parallel tiled code for non-trivial DP recurrences. The major concerns with tiled programs are

that they are cache-aware and sometimes processor-aware that sacrifices portability across machines. Another disadvantage of being cache-aware is that the algorithms are not cache-adaptive [21], i.e., the algorithms do not adapt to changes in available cache/memory space during execution and hence may run slower when multiple programs run concurrently in a shared-memory environment [14] (Chapter4). Several existing systems such as Bellman's GAP compiler [86], semi-automatic synthesizer [149], EasyPDP [170], EasyHPS [63], pattern-based system [123], and parallelizing plugins [153] can be used to generate iterative and tiled-loop programs. Parallel task graph execution systems such as Nabbit [10] and BDDT [185] execute the DP tasks during runtime using unrolling. Due to this they might lose cache efficiency.

The classic 2-way recursive divide-and-conquer (CORDAC) algorithms with optimal serial cache complexity and good (but, not always optimal) parallelism have been developed, analyzed, and implemented in [47, 48], [182] (see Chapter 3). Hybrid r-way algorithms have considered in [47] but they are either cache- or processor-aware, and complicated to program. Pochoir [171] is used to generate cache-oblivious implementations for stencils. However, the recursive algorithms often have low parallelism due to artificial dependencies among subtasks. Recently Aga et al. in [8] proposed a speculation approach to alleviate the concurrency constraints imposed by the artificial dependencies in standard parallel recursive divide-and-conquer programs, and reported a speedup up to 1.6× on 30 cores over their baseline.

The recursive wavefront algorithms were introduced in [173] but they are too complicated to develop, analyze, implement, and generalize. They make extensive use of atomic instructions, and standard analysis model of fork-join parallelism does not apply. In this work we try to address these issues.

Our contributions Our major contributions in this work are as follows:

(1) We show how to systematically transform recursive divide-and-conquer to cache-oblivious wavefront. We present a generic method to develop and schedule recursive wavefront algorithms based on timing functions.

(2) We present two approaches for scheduling a recursive wavefront algorithm: (*i*) the algorithm passes timing functions and space usage info to a hint-accepting space-bounded scheduler, (*ii*) the programmer appropriately transforms the algorithm to use the timing functions, and uses a standard randomized work-stealing scheduler to run the program.

(3) We present performance and scalability results of the presented algorithms on state-of-the art multicore machines and show a comparative analysis with standard 2-way CORDAC and the original cache-oblivious wavefront (COW) algorithms from [173].

5.3 Deriving recursive wavefront algorithms

In this section, we describe how to transform a standard cache-oblivious recursive divide-and-conquer (CORDAC) DP algorithm into a recursive wavefront algorithm. The method involves augmenting all recursive function calls with timing functions to launch them as early as possible without violating any dependency constraints implied by the DP recurrence. The timing functions are derived analytically, and do not employ locks or atomic instructions.

```
A(X, X, X)

1. if X is a cell then A_cell(X, X, X)
2. else
3.     par A(X11, X11, X11),  A(X22, X22, X22)
4.     B(X12, X11, X22)
```

```
B(X, U, V)

1. if X is a cell then B_cell(X, U, V)
2. else
3.     B(X21, U22, V11)
4.     par C(X11, U12, V21), C(X22, X21, V12)
5.     par B(X11, U11, V11), B(X22, X22, V22)
6.     C(X12, U12, X22)
7.     C(X12, X11, V12)
8.     B(X12, U11, V22)
```

```
C(X, U, V)

1. if X is a cell then C_cell(X, U, V)
2. else
3.     par C(X11, U11, V11), C(X12, U11, V12),
           C(X21, U21, V11), C(X22, U21, V12)
4.     par C(X11, U12, V21), C(X12, U12, V22),
           C(X21, U22, V21), C(X22, U22, V22)
```

Programmer computes the timing functions

```
S_A (X, X, X)

1. return 𝔈(x_r, x_c)
```

```
S_B (X, U, V)

1. return 𝔈(x_r + n − 1, x_c)
```

```
S_C (X, U, V)
  //here, x_r = u_r, x_c = v_c, and u_c = v_r

1. m ← (x_r + n − 1 + x_c)/2; û ← u_c + n − 1
2. if u_c > m then
3.     return max {𝔈(u_r + n − 1, u_c), 𝔈(v_r, v_c)} + 1
4. elif û < m then
       return max {𝔈(u_r + n − 1, û), 𝔈(û, v_c)} + 1
5. else return
       (max {𝔈(u_r + n − 1, m), 𝔈(m, v_c)} + 1).[u_c > (x_r + x_c)/2]
```

```
E_A (X, X, X)

1. return 𝔈(x_r, x_c + n − 1)
```

```
E_B (X, U, V)

1. return 𝔈(x_r, x_c + n − 1)
```

```
E_C (X, U, V)

1. lval ← max {𝔈(u_r, u_c), 𝔈(v_r, v_c + n − 1)}
2. rval ← max{𝔈(u_r, u_c + n − 1), 𝔈(v_r + n − 1, v_c + n − 1)}
3. return (max {lval, rval} + 1) .[u_c > (x_r + x_c)/2]
```

```
𝔈(i, j)

1. if (j − i) ≤ 1 then return (j − i) else return 2 × (j − i) − 1
```

Transformation by the scheduler/programmer

```
RecursiveWavefront-Parenthesis()

1. w ← 0
2. while w < ∞ do w ← A(G, G, G, w)
```

```
A(X, X, X, w)

1. v_i ← ∞ for all i ∈ [1, 3] //number of function calls=3
2. if X is an n' × n' matrix then
3.     if w = S_A (X, X, X) then A_non−wave(X, X, X)
4. else
5.     F_1..3 ← {A, A, B}
6.     arg_1..3 ← {(X11, X11, X11), (X22, X22, X22),
       (X12, X11, X22)}
7.     par for i ← 1 to 3 do
8.         if w < S_Fi(arg_i) then v_i ← S_Fi(arg_i)
9.         elif w ≤ E_Fi(arg_i) then v_i ← F_i(arg_i, w)
10. sync
11. return min v_i for all i ∈ [1, 3]
```

```
B(X, U, V, w)

1. v_i ← ∞ for all i ∈ [1, 8] //number of function calls=8
2. if X is an n' × n' matrix then
3.     if w = S_B (X, U, V) then B_non−wave(X, U, V)
4. else
5.     F_1..8 ← {B, C, C, B, B, C, C, B}
6.     arg_1..8 ← {(X21, U22, V11), (X11, U12, V21),
                   (X22, X21, V12), (X11, U11, V11),
                   (X22, X22, V22), (X12, U12, X22),
                   (X12, X11, V12), (X12, U11, V22)}
7.     par for i ← 1 to 8 do
8.         if w < S_Fi(arg_i) then v_i ← S_Fi(arg_i)
9.         elif w ≤ E_Fi(arg_i) then v_i ← F_i(arg_i, w)
10. sync
11. return min v_i for all i ∈ [1, 8]
```

```
C(X, U, V, w)

1. v_i ← ∞ for all i ∈ [1, 8] //number of function calls=8
2. if X is an n' × n' matrix then
3.     if w = S_C (X, U, V) then C_non−wave(X, U, V)
4. else
5.     F_1..8 ← {C, C, C, C, C, C, C, C}
6.     arg_1..8 ← {(X11, U11, V11), (X12, U11, V12),
                   (X21, U21, V11), (X22, U21, V12),
                   (X11, U12, V21), (X12, U12, V22),
                   (X21, U22, V21), (X22, U22, V22)}
7.     par for i ← 1 to 8 do
8.         if w < S_Fi(arg_i) then v_i ← S_Fi(arg_i)
9.         elif w ≤ E_Fi(arg_i) then v_i ← F_i(arg_i, w)
10. sync
11. return min v_i for all i ∈ [1, 8]
```

Figure 5.1: *Left: The programmer derives the timing functions from a given standard 2-way recursive divide-and-conquer DP algorithm for the parenthesis problem. A matrix region Z has its top-left corner at (z_r, z_c) and is of size $n \times n$. Right: A recursive divide-and-conquer wavefront algorithm is generated for the parenthesis problem. The programmer derives the algorithm if work-stealing scheduler is used, and the scheduler derives the algorithm if modified hint-accepting space-bounded scheduler (Section 5.5) is used. The algorithm makes use of the timing functions derived by the programmer.*

Our transformation allows the updates to the DP table proceed in an order close to iterative wavefront, but from within the structure of a recursive divide-and-conquer algorithm. The goal is to reach the higher parallelism of an iterative wavefront algorithm while retaining the better cache performance (i.e., efficiency and adaptivity) and portability (i.e., cache- and processor-obliviousness) of a recursive algorithm.

Let us first define the *wavefront order* of applying updates to a DP table. Each update writes to one DP table cell by reading values from other cells. We say that a cell is *fully updated* provided it is never updated in the future. An update becomes *ready* when all cells it reads from are fully updated. We assume that only ready updates can be applied and each such update can only be applied once. A wavefront order of updates proceeds in discrete timesteps. In each step all ready updates to distinct cells are applied in parallel. However, if a cell has multiple ready updates only one of them is applied, and the rest are retained for future to avoid race conditions. A wavefront order does not have any artificial dependencies.

Transformation. It is completed in three major steps:

(1) [**Construct completion-time function.**] A closed-form formula is derived based on the original DP recurrence that gives the timestep at which each DP cell is fully updated in wavefront order.

(2) [**Construct start- and end-time functions.**] Cell completion times are used to derive closed-form formulas that give the timesteps in wavefront order at which each recursive function call should start and end execution.

(3) [**Derive the recursive wavefront algorithm.**] Each recursive function call in the standard CORDAC algorithm is augmented with its start- and end-time functions so that the algorithm can be used to apply only the updates in any given timestep in wavefront order. We then use a variant of iterative deepening on top of this recursive algorithm to execute all timesteps efficiently in non-decreasing wavefront order.

We describe our transformation for arbitrary d-dimensional ($d \geq 1$) DP in which each dimension of the DP table is of the same length and is a power of 2.

Example of transformation. We explain our approach by applying it on a recursive algorithm for the parenthesis problem [37], which is defined as follows. Let $C[i,j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then the 2D DP table $C[0:n, 0:n]$ is filled up using the following recurrence:

$$C[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ v_j & \text{if } 0 \leq i = j-1 < n, \\ \min\limits_{i \leq k \leq j} \left\{ C[i,k] + C[k,j] + w(i,k,j) \right\} & \text{if } 0 \leq i < j-1 < n; \end{cases} \tag{5.1}$$

where the v_j's and function $w(\cdot,\cdot,\cdot)$ are given. The recurrence is evaluated by the recursive algorithm [182] given at the top of Figure 5.1 which is same as the algorithm presented in Chapter 3 Figure 3.5. In the rest of the section, we show how a recursive wavefront algorithm (shown in Figure 5.1) can be derived from the given CORDAC algorithm.

Consider the standard 2-way CORDAC algorithm for the parenthesis problem given in the top-left corner of Figure 5.1. It has three functions that update the DP table. Initially, function $\mathcal{A}(C,C,C)$ is called, where C is the entire DP table. Then the computation progresses by recursively breaking the table into quadrants, and calling functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ on these smaller

(a) top: update timesteps

$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_3\mathcal{B}_4$	$\mathcal{C}_5\mathcal{C}_6\mathcal{B}_7$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{19}\mathcal{C}_{20}\mathcal{B}_{21}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{27}\mathcal{B}_{28}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{29}\mathcal{C}_{30}\mathcal{B}_{31}$
–	$\mathcal{A}_0$	$\mathcal{B}_2$	$\mathcal{C}_3\mathcal{B}_4$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{B}_{16}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{B}_{26}$	$\mathcal{C}_{22}\mathcal{C}_{23}\mathcal{C}_{24}\mathcal{C}_{25}\mathcal{C}_{27}\mathcal{B}_{28}$
–	–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{11}\mathcal{C}_{12}\mathcal{B}_{13}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{19}\mathcal{C}_{20}\mathcal{B}_{21}$
–	–	–	$\mathcal{A}_0$	$\mathcal{B}_8$	$\mathcal{C}_9\mathcal{B}_{10}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{B}_{16}$	$\mathcal{C}_{14}\mathcal{C}_{15}\mathcal{C}_{17}\mathcal{B}_{18}$
–	–	–	–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_3\mathcal{B}_4$	$\mathcal{C}_5\mathcal{C}_6\mathcal{B}_7$
–	–	–	–	–	$\mathcal{A}_0$	$\mathcal{B}_2$	$\mathcal{C}_3\mathcal{B}_4$
–	–	–	–	–	–	$\mathcal{A}_0$	$\mathcal{B}_1$
–	–	–	–	–	–	–	$\mathcal{A}_0$

(a) top: completion timesteps

0	1	4	7	18	21	28	31
–	0	2	4	16	18	26	28
–	–	0	1	10	13	18	21
–	–	–	0	8	10	16	18
–	–	–	–	0	1	4	7
–	–	–	–	–	0	2	4
–	–	–	–	–	–	0	1
–	–	–	–	–	–	–	0

(b) bottom: update timesteps

$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{B}_{12}$	$\mathcal{C}_7\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$	$\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{C}_{16}\mathcal{C}_{17}\mathcal{B}_{18}$
–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{B}_{12}$	$\mathcal{C}_7\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{C}_{13}\mathcal{C}_{14}\mathcal{B}_{15}$
–	–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$	$\mathcal{C}_7\mathcal{C}_8\mathcal{C}_{10}\mathcal{C}_{11}\mathcal{B}_{12}$
–	–	–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$	$\mathcal{C}_4\mathcal{C}_7\mathcal{C}_8\mathcal{B}_9$
–	–	–	–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$	$\mathcal{C}_4\mathcal{C}_5\mathcal{B}_6$
–	–	–	–	–	$\mathcal{A}_0$	$\mathcal{B}_1$	$\mathcal{C}_2\mathcal{B}_3$
–	–	–	–	–	–	$\mathcal{A}_0$	$\mathcal{B}_1$
–	–	–	–	–	–	–	$\mathcal{A}_0$

(b) bottom: completion timesteps

0	1	3	6	9	12	15	18
–	0	1	3	6	9	12	15
–	–	0	1	3	6	9	12
–	–	–	0	1	3	6	9
–	–	–	–	0	1	3	6
–	–	–	–	–	0	1	3
–	–	–	–	–	–	0	1
–	–	–	–	–	–	–	0

TABLE 5.1: *Timesteps at which each DP table cell is updated ($\mathcal{F}_t$ means function $\mathcal{F}$ updates at timestep t) and the timestep at which each cell becomes fully updated (on the right) for the parenthesis problem on a DP table of size 8×8 using (a) top: standard 2-way recursive algorithm, and (b) bottom: recursive/iterative wavefront algorithm. Both recursive algorithms use a 1×1 base case. We assume that the number of processors is infinite.*

regions of C. At the base case (i.e., a 1×1 region of C), each function updates a cell. When x is a cell, function $\mathcal{A}(x, x, x)$ updates x by reading x itself which corresponds to the case $i = k = j$ in the recurrence. Similarly, function $\mathcal{B}(x, u, v)$ updates cell x by reading x itself and two other cells u and v which correspond to cases $i = k \neq j$ and $i \neq k = j$. Finally, function $\mathcal{C}(x, u, v)$ updates the cell x by reading the two cells u and v which corresponds to $i \neq k \neq j$.

The top part of Table 5.1 shows how the standard 2-way CORDAC algorithm with 1×1 base case updates $C[1 : n, 1 : n]$ when $n = 8$. We use $\mathcal{F}_t$ in a cell to denote that function $\mathcal{F}$ updates the cell at timestep t, where $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$. Using an infinite number of processors, the standard CORDAC algorithm updates the entire table in 31 timesteps. In contrast, the bottom part of Table 5.1 shows that the fastest iterative wavefront algorithm will update C in only 18 timesteps. With a 1×1 base case our recursive wavefront algorithm (shown on the right hand side of Figure 5.1) will perform the updates in exactly the same order as the iterative wavefront algorithm, and terminate in 18 steps.

5.3.1 Constructing completion-time function

In this section, we define completion-time, and show how to compute it in $\mathcal{O}(1)$ time for any cell. There are two different ways to define completion time of a cell. The completion time of a cell is the time after which it will not be updated/written any more. In other words, if in a dynamic programming problem, a cell x is updated/written k times, completion time of x is the latest time when x would be written/updated. It is also possible to define completion time based on the completion time of the input cells as well. In this case, if in a DP problem a cell x needs to be updated based on values from k other cells y_1, y_2, ..., y_k at different updates, completion time of x can be computed by taking maximum completion time of all cells that x depends on (i.e., y_1, y_2, ..., y_k), plus 1. If for a DP problem, there are m input cells with the maximum completion time, then we need to wait at least m time steps before we can start updating x. Furthermore, if a cell also depends on its own value, we also need to add an extra 1 to the completion time to consider the self update time.

A formal definition of completion time follows:

Definition 5.1 (**Completion-time**). The completion-time for a particular cell x, denoted by $\mathfrak{C}(x)$, is the timestep in wavefront order at which x is fully updated. More formally,

$$\mathfrak{C}(x) = \max\ t \mid \text{for all } \mathcal{F}_t(x, \ldots); \tag{5.2}$$

where $\mathcal{F}_t(x, \ldots)$ means that cell x is updated by function $\mathcal{F}$ at timestep t.

Completion-time of a cell computed from the given DP recurrence as follows.

$$\mathfrak{C}(x) = \begin{cases} \text{initial values} & \text{initial conditions,} \\ rmax(x) + m + su(x) & \text{otherwise;} \end{cases} \tag{5.3}$$

where $rmax(x)$ is the maximum completion time of the cells on which x directly depends, i.e., $rmax(x) = \max_{\mathcal{F}(x,\ldots,y,\ldots)} \mathfrak{C}(y)$, m is the number of input cells (on which x directly depends on) with that maximum completion time, and the term $su(x)$ is 1 if there is a self-update function that reads from itself; and 0, otherwise.

The bottom part of Table 5.1 shows completion-times for the parenthesis problem for all cells of an 8×8 DP table. The completion-time for any cell (i, j) in the DP table can be found as follows.

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \mathfrak{C}(i, j-1) + 1 + 1 & \text{if } i = j - 1; \\ \mathfrak{C}(i, j-1) + 2 + 1 & \text{if } i < j - 1; \end{cases} \tag{5.4}$$

because $rmax(i,j) = \mathfrak{C}(i, j-1) = \mathfrak{C}(i+1, j)$ (Note that $m = 1$ when $i = j - 1$ and $m = 2$ when $i < j - 1$), and $su(i,j) = 1$ as the self-update function $\mathcal{B}$ updates the cell (i, j) reading from itself. Solving the recurrence (assuming that race will be avoided by fractional timing as explained in the following section, and therefore considering $m = 1$ always), we get the following.

$$\mathfrak{C}(i,j) = \begin{cases} j - i & \text{if } (j-i) = 0, \\ 2(j-i) - 1 & \text{if } (j-i) \geq 1. \end{cases} \tag{5.5}$$

We also need to know start-time and end-time of each recursive function call in the wavefront order so that we can execute them appropriately. Both start-time and end-time are defined for each function and they depend on the function type and their input and output parameters. Ideally the start-time of a function in a recursive wavefront algorithm can be computed recursively by taking the smallest start-time of any sub-function called by the original function on any of the quadrants (often its a quadrant that has earliest completion time in the wavefront order). On the other hand, end-time can be computed by recursively taking the maximum end-time of any sub-function on any quadrant (often its a quadrant that has latest completion time in the wavefront order). We also need to take care of any possible race conditions into account which may happen if two different function calls with the same write cell/region become ready at the same time. In such case, both start-time and end-time need to be adjusted to avoid any potential race.

Definition 5.2 (**Start-time and end-time**). The start-time (resp. end-time) of a recursive function call in a recursive wavefront algorithm is the earliest (resp. latest) timestep in wavefront

order at which one of the updates to be applied by that function call (either directly or through a recursive function call) becomes ready.

Let $\mathcal{F}(X, Y_1, \ldots, Y_s)$ be a function call that writes to a region X by reading from regions $Y_1, \ldots, Y_s$ of the DP table. Its start- and end-times, denoted by $\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)$ and $\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)$, respectively, are computed as follows.

$$\underbrace{\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \in \{Y_1, \ldots, Y_s\}} = \begin{cases} (\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\ \min\ \mathcal{S}_{\mathcal{F}'}(X', Y_1', \ldots, Y_s') & \text{otherwise;} \end{cases}$$

$$\underbrace{\mathcal{S}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \notin \{Y_1, \ldots, Y_s\}} = \begin{cases} (\min_{1 \le i \le s}\{\mathfrak{C}(Y_i)\} + 1).ra(X) & \text{if } X \text{ is a cell,} \\ \min\ \mathcal{S}_{\mathcal{F}'}(X', Y_1', \ldots, Y_s') & \text{otherwise;} \end{cases}$$

$$\underbrace{\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \in \{Y_1, \ldots, Y_s\}} = \begin{cases} (\mathfrak{C}(X)).0 & \text{if } X \text{ is a cell,} \\ \max\ \mathcal{E}_{\mathcal{F}'}(X', Y_1', \ldots, Y_s') & \text{otherwise;} \end{cases}$$

$$\underbrace{\mathcal{E}_{\mathcal{F}}(X, Y_1, \ldots, Y_s)}_{X \notin \{Y_1, \ldots, Y_s\}} = \begin{cases} (\max_{1 \le i \le s}\{\mathfrak{C}(Y_i)\} + 1).ra(X) & \text{if } X \text{ is a cell,} \\ \max\ \mathcal{E}_{\mathcal{F}'}(X', Y_1', \ldots, Y_s') & \text{otherwise;} \end{cases}$$

where, in the non-cellular case minimization/maximization is taken over all functions $\mathcal{F}'(X', Y_1', \ldots, Y_s')$ recursively called by $\mathcal{F}(X, Y_1, \ldots, Y_s)$. Here, $ra(X)$ is the problem-specific race avoidance condition used when two functions write to the same region. Though we use real-valued timesteps for simplicity, the total number of distinct timesteps remains exactly the same as that in the iterative wavefront algorithm.

For the parenthesis problem, the start-times for the three functions $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ are computed as below. Let $(x_r, x_c), (u_r, u_c)$, and (v_r, v_c) denote the positions of the top-left cells of regions X, U and V, respectively. Note that for parenthesis problem, $x_r = u_r$, $x_c = v_c$, and $u_c = v_r$. Then,

$$\mathcal{S}_{\mathcal{A}}(X, X, X) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{A}}(X_{11}, X_{11}, X_{11}) & \text{otherwise;} \end{cases}$$

$$\mathcal{S}_{\mathcal{B}}(X, U, V) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{S}_{\mathcal{B}}(X_{21}, U_{22}, V_{11}) & \text{otherwise;} \end{cases}$$

$$\mathcal{S}_{\mathcal{C}}(X, U, V) = \begin{cases} \left(\max\left\{\mathfrak{C}(U), \mathfrak{C}(V)\right\} + 1\right).[u_c > \frac{x_r + x_c}{2}] & \text{if } X \text{ is a cell,} \\ \min\left\{\begin{array}{l} \mathcal{S}_{\mathcal{C}}(X_{21}, U_{21}, V_{11}), \\ \mathcal{S}_{\mathcal{C}}(X_{21}, U_{22}, V_{21}) \end{array}\right\} & \text{otherwise;} \end{cases}$$

where [ ] is the Iversion bracket which denotes 1 if the condition in the bracket is true and 0 otherwise. The quadrant X_{21} appears in the start time because, for those cases, the bottom-left cell of the X_{12} quadrant is the cell which always gets updated first in the wavefront order.

Both $\mathcal{A}$ and $\mathcal{B}$ read from and write to X, and hence their start-times follow directly from the first recurrence in Definition 5.2. In case of $\mathcal{B}$, X is updated by reading from pair $\langle U, X \rangle$ and also from $\langle X, V \rangle$. Function $\mathcal{C}$ follows the second recurrence from the definition, since for $\mathcal{C}$ there is no overlap between the read and write regions. As $\mathcal{C}$ writes to the same region twice, there is a race and to avoid it we use the condition $[u_c > (x_r + x_c)/2]$ derived manually. An intuition behind this condition is that, we delay the function calls for which $w_u - w_x < w_x - w_v$, where w_u is the wavefront where u lies on, and let the other functions to run so that no race happens.

Similarly, the end-times are as follows.

$$\mathcal{E}_{\mathcal{A}}(X,X,X) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{E}_{\mathcal{B}}(X_{12}, X_{11}, X_{22}) & \text{otherwise;} \end{cases}$$

$$\mathcal{E}_{\mathcal{B}}(X,U,V) = \begin{cases} \mathfrak{C}(X).0 & \text{if } X \text{ is a cell,} \\ \mathcal{E}_{\mathcal{B}}(X_{12}, U_{11}, V_{22}) & \text{otherwise;} \end{cases}$$

$$\mathcal{E}_{\mathcal{C}}(X,U,V) = \begin{cases} \left(\max\left\{\mathfrak{C}(U), \mathfrak{C}(V)\right\} + 1\right).[u_c > \frac{x_r + x_c}{2}] & \text{if } X \text{ is a cell,} \\ \max\begin{Bmatrix} \mathcal{E}_{\mathcal{C}}(X_{12}, U_{11}, V_{12}), \\ \mathcal{E}_{\mathcal{C}}(X_{12}, U_{12}, V_{22}) \end{Bmatrix} & \text{otherwise.} \end{cases}$$

The quadrant X_{12} appears in the end-time recurrence because, the top-right cell of the X_{12} quadrant is the cell which always gets updated the last in the wavefront order. Note that the start time and end time of a cell are always the same, since we assume that it takes only a constant amount of time to update a cell. Solving the recurrences for the start-times and end-times above, we obtain the timing functions shown in Figure 5.1.

5.3.2 Deriving a recursive wavefront algorithm

In this section, we describe how to use timing functions to derive a recursive wavefront algorithm from a given standard recursive divide-and-conquer (CORDAC) DP algorithm. We use the parenthesis problem as an example.

A standard CORDAC algorithm for the parenthesis problem is shown in the top-left corner of Figure 5.1. We modify it as follows, and the modified algorithm is shown on the right hand side of the same figure.

First, we modify each function $\mathcal{F}$ to include a switching point $n' \geq 1$, and switch to the original non-wavefront recursive algorithm by calling $\mathcal{F}_{non-wave}$ when the size of each input submatrix drops to $n' \times n'$ or below.

We augment each function to accept a timestep parameter w. We remove all serialization among recursive function calls by making sure that all functions that are called are launched in parallel. However, we do not launch a function unless w lies between its start-time and end-time which means that a function is not invoked if we know that it does not have an update to apply at timestep w in wavefront order. Observe that the function $\mathcal{F}_{non-wave}$ at switching does not accept a timestep parameter, but if we reach it we know that it has an update to apply at timestep w. However, once we enter that function we do not stop until we apply all updates that function can apply at all timesteps $\geq w$.

Each function is also modified to return the smallest timestep above w for which it may have at least one update that is yet to be applied. It finds that timestep by checking the start-time of each function that was not launched because the start-time was larger than w, and the timestep returned by each recursive function that was launched, and taking the smallest of all of them.

Finally, we add a loop (see RecursiveWavefront-Parenthesis in Figure 5.1) to execute all timesteps of the wavefront in non-decreasing fashion using the modified functions. We start with timestep $w = 0$, and invoke the main function $\mathcal{A}(C, C, C, w)$ which applies all updates at

timestep w and depending on the value chosen for n' possibly some updates above timestep w, and returns the smallest timestep above w for which there may still be some updates that are yet to be applied. We next call function $\mathcal{A}$ with that new timestep value, and keep iterating in the same fashion until we are able to exhaust all timesteps.

5.4 Applications

In this section, we present the recursive wavefront algorithms for three other dynamic programming problems: LCS, Floyd-Warshall's APSP, and gap problem. All of these DP problems have many applications in bioinformatics. LCS and gap problem have applications in sequence alignement and other types of biological sequence analyses, FW-APSP has application in phylogeny analysis and parenthesis problem is used in RNA secondary structure predictions. In this section, we will only give the timing functions and not the entire recursive wavefront algorithm. We give references to the papers that present the standard (non-wavefront) CORDAC algorithms from which recursive wavefront algorithms can easily be derived by plugging in the timing functions as per Section 5.3.2.

Longest common subsequence (LCS). The LCS problem [46, 103] asks one to find the longest of all common subsequences [56] between two strings. Here, we are interested in finding only the length of the LCS. In LCS DP, a cell depends on its three adjacent cells and has the same dependency structure as the standard edit distance problem (see Chapter 2).

We build on the recursive algorithm given in [46] which has only one function $\mathcal{A}$ (i.e., named LCS-Output-Boundary in [46]). The timing functions are as follows.

$$\begin{aligned}
\mathfrak{C}(i,j) &= i+j,\\
\mathcal{S}_{\mathcal{A}}(X) &= \mathfrak{C}(x_r, x_c),\\
\mathcal{E}_{\mathcal{A}}(X) &= \mathfrak{C}(x_r+n-1, x_c+n-1);
\end{aligned}$$

where, (x_r, x_c) is the top-left corner of X. There is no self dependency or race in the function calls and therefore we omitted fractional timestamps. An intuition behind why these timesteps are correct is that, for LCS any cell on diagonal/wavefront w gets fully updated at timestep w, and a cell with index (x_r, x_c) lies on diagonal (x_r+x_c). A cell is updated only once. Furthermore, for any quadrant/region with top-left corner at (x_r, x_c) and size $n \times n$, the top-left cell, (x_r, x_c) is the first cell to be computed and the bottom-right cell, (x_r+n-1, x_c+n-1) is the last cell to be computed in the wavefront order, and the start and end times directly follow from this observation.

Gap problem. Sequence alignment with general gap penalty [83, 84, 182, 189] is a generalization of the edit distance problem. We build on the recursive algorithm given in [182] (see Chapter 3). The timing functions are as follows.

$$\begin{aligned}
\mathfrak{C}(i,j) &= 2(i+j),\\
\mathcal{S}_{\mathcal{F}}(X,Y) &= (\mathfrak{C}(y_r, y_c) + [X \neq Y]).ra(X),\\
\mathcal{E}_{\mathcal{F}}(X,Y) &= (\mathfrak{C}(y_r+n-1, y_c+n-1) + [X \neq Y]).ra(X);
\end{aligned}$$

where, when $\mathcal{F}$ is $\mathcal{A}$ (resp. $\mathcal{B}, \mathcal{C}$), then Y is X (resp. U, V), (y_r, y_c) is the top-left corner of region Y; and $ra(X) = [x_c \geq n']$ for function $\mathcal{C}$, and $ra(X) = 0$, otherwise. The $[X \neq Y]$ condition has been added to take care of the self update time. The $ra(X)$ condition is added to avoid race condition, since in gap problem, function B and C can be applied in parallel for all cells (resp. basecase/block) except the cells (resp. basecases/blocks) in the first row and the first column.

Floyd-Warshall's all-pairs shortest path (FW-APSP). For Floyd-Warshall's APSP [77, 188] we build on the recursive algorithm given in [48]. However, that algorithm violates our assumption that cells can only be updated using values from fully updated cells. That violation can be removed by performing the computation in cubic space instead of quadratic space as explained in [14, 56]. We find the timing functions in cubic space which remain valid for the DP using quadratic space.

Completion-time is given by $\mathfrak{C}(i, j, k) = 3k + [i \neq k] + [j \neq k]$, where, [ ] is the Iversion bracket.

Let (x_r, x_c, x_h) be the cell with the smallest coordinates in X. Then for each $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}\}$,

$$\mathcal{S}_{\mathcal{F}}(X, \ldots) = \mathfrak{C}(x_r, x_c, x_h), \text{ and}$$
$$\mathcal{E}_{\mathcal{F}}(X, \ldots) = \max \left\{ \begin{matrix} \mathfrak{C}(x_r, x_c, x_h + n - 1), \mathfrak{C}(x_r, x_c + n - 1, x_h + n - 1), \\ \mathfrak{C}(x_r + n - 1, x_c, x_h + n - 1), \\ \mathfrak{C}(x_r + n - 1, x_c + n - 1, x_h + n - 1) \end{matrix} \right\}.$$

Here, end time is basically computed by taking the maximum end time for four corner cells in the DP table, since in case of FW-APSP, the wavefront moves both in forward and in backward directions.

5.5 Scheduling recursive wavefront algorithms

In this section, we show how to schedule recursive wavefront algorithms to achieve provably good bounds (optimal or near-optimal) for both parallelism and cache performance.

Recall that our recursive wavefront algorithm switches to the original non-wavefront CORDAC algorithm when the input parameter n drops to a value $\leq n'$. While both recursive (wavefront and non-wavefront) algorithms have the same serial work T_1, and same serial cache complexity Q_1 (as they reduce to the same serial algorithm), their spans are different. We use $T_\infty^R(n')$ to denote the span of the non-wavefront algorithm for a problem of size n'.

Scheduling using work-stealing (WS) scheduler. As explained before (see Figure 5.1), it is possible to implement these recursive wavefront algorithms by annotating each function call with their start and end time and then use a standard fork-join scheduler such as the randomized work-stealing (WS) scheduler [27]. In that case, all the scheduler bounds will hold for the program. For example if $N_\infty(n/n')$ denotes the number of distinct wavefronts in wavefront order for a recursive wavefront algorithm on an infinite number of processors, the span of this algorithm will be $T_\infty(n) = \mathcal{O}\left(N_\infty(n/n')\left(\log(n/n') + T_\infty^R(n')\right)\right)$, since it takes $O(\log n/n')$ time to reach a sub-problem of size n' recursively after starting from a size of n. For LCS the serial work is $T_1(n) = \Theta(n^2)$, and for other problems described in this chapter $T_1(n) = \Theta(n^3)$. Then, the running time on p cores, $T_p(n) = \mathcal{O}(T_1(n)/p + T_\infty(n))$ (w.h.p. in n) and parallel cache

complexity, $Q_p(n) = \mathcal{O}(Q_1(n) + p(M/B)T_\infty(n))$ (w.h.p. in n). The later bounds directly follow from the bound provided by cilk's work-stealing scheduler. Here, we assume that $T_1(n') = \Omega(\log n)$. The extra space used, $S_p(n) = \mathcal{O}(p \log n)$, since any path from the root to the leaf of the recursion can be of at most of length $\Omega(\log n)$ and there can be at most p active branches. More detailed formal proofs for all other bounds can be found in [44].

Scheduling using a modified space-bounded (W-SB) scheduler. In this section, we show how to modify a space-bounded scheduler [51] so that it can execute a recursive wavefront algorithm cache-optimally with near-optimal parallelism.

For each recursive function call, our W-SB scheduler accepts three hints: start-time, end-time and working set size (i.e., total size of all regions in the DP table accessed by the function call). Given an implementation of a standard recursive/CORDAC algorithm with each function call annotated with those three hints, the W-SB can automatically generate a recursive wavefront implementation (similar to the one on the right hand side of Figure 5.1). From the given start-times, the scheduler determines the lowest start-time and executes the tasks that can be executed at that lowest start-time. Since the scheduler knows all the cache sizes, as soon as the working set size of any function executing on a processor under a cache fits into that cache, the scheduler anchors the function to that cache in the sense that all recursive function calls made by that function and its descendants will only be executed by the processors under that anchored cache. This approach of limiting migration of tasks ensures cache-optimality [26, 51]. All bounds for runtime complexity shown for the work-stealing scheduler remain the same for space-bounded scheduler. However, the parallel cache complexity changes to $Q_p(n) = \mathcal{O}(Q_1(n))$ which happens due to the restriction on task migration. Some formal proofs for these bounds can be found in [44].

5.6 Experimental results

In this section we present experimental results showing performance of recursive wavefront algorithms for the parenthesis, LCS and 2D FW-APSP problems.

We compare performance of those algorithms with the corresponding standard 2-way recursive divide-and-conquer (CORDAC) and the original cache-oblivious wavefront (COW) algorithms [173]. We used C++ with Intel®Cilk™Plus extension to implement all algorithms presented in this section. Therefore, all implementations basically used the work-stealing scheduler provided by Cilk™ runtime system. All programs were compiled with `-O3 -ip -parallel -AVX -xhost` optimization parameters. The codes were not hand-optimized. To measure cache performance, we used PAPI-5.3 [4]. Table 5.2 lists the systems on which we ran our experiments.

Model	E5-2680	E5-4650	E5-2680v3
Cluster	Stampede [5]	Stampede [5]	Comet [2]
#Cores	2x8	4x8	2x12
Frequency	2.70GHz	2.70GHz	2.50GHz
L1	32K	32K	32K
L2	256K	256K	256K
L3	20480K	20480K	30720K
Cache-line size	64B	64B	64B
Memory	64GB	1TB	64GB
Compiler	15.0.2	15.0.2	15.2.164
OS	CentOS 6.6	CentOS 6.6	CentOS 6.6

TABLE 5.2: *System specifications.*

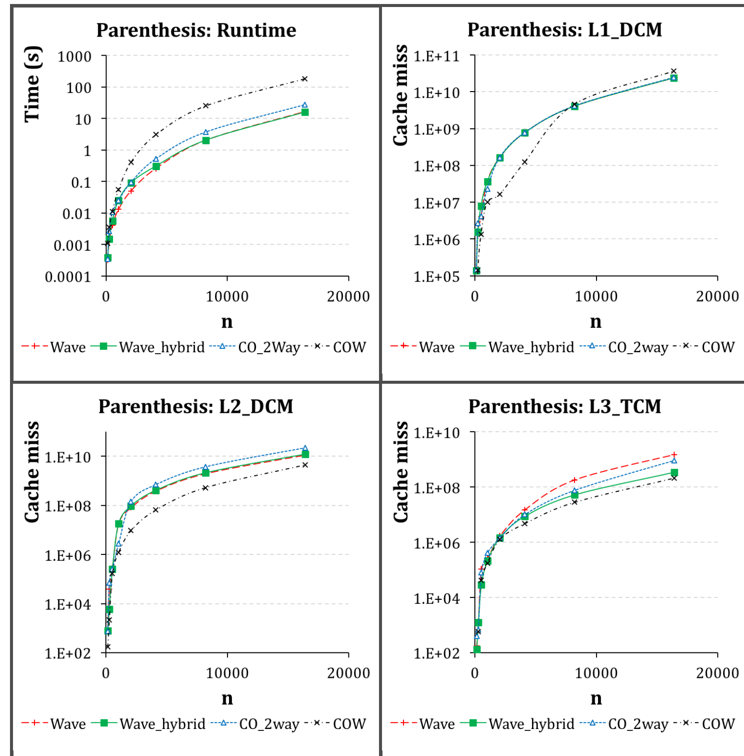


FIGURE 5.2: *Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the Parenthesis Problem. All programs were run on* 16 *core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler.*

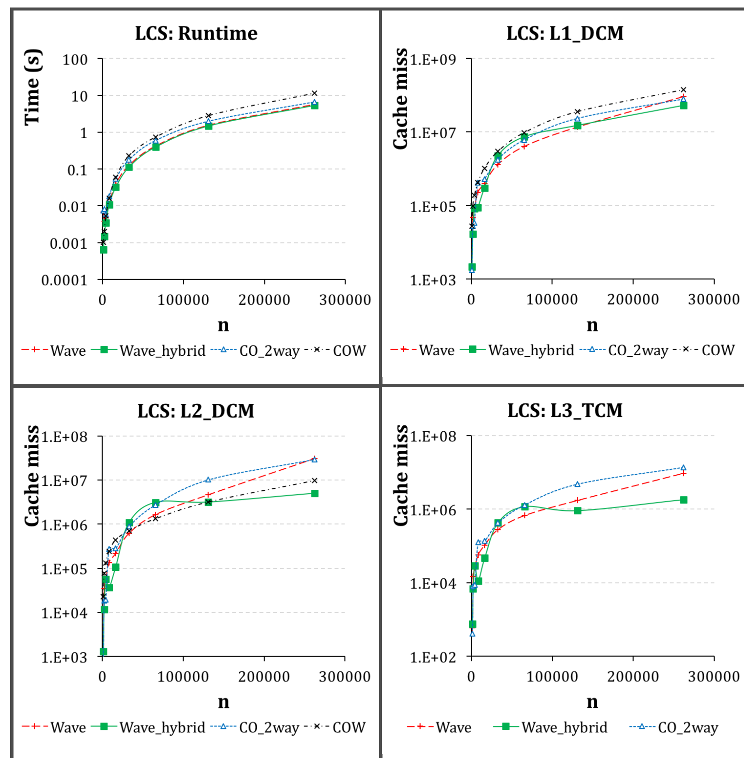


FIGURE 5.3: *Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the LCS Problem. All programs were run on* 16 *core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler.*

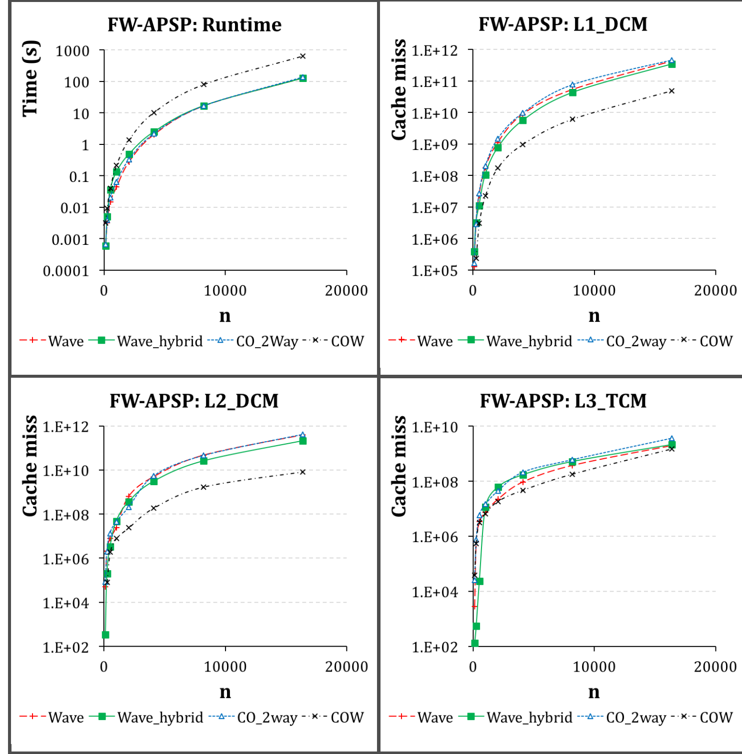


FIGURE 5.4: *Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the 2D FW-APSP Problem. All programs were run on* 16 *core machines in Stampede. All implementations used Cilk Plus's work-stealing scheduler.*

Figure 5.2, 5.3, and 5.4 show performance of the following on a 16-core Sandy Bridge machine: (i) recursive wavefront algorithm that does not switch to the 2-way non-wavefront recursive algorithm and instead directly uses an iterative basecase (`wave`), (ii) recursive wavefront algorithm that switches to the 2-way CORDAC at some point (`wave-hybrid`), (iii) standard 2-way CORDAC algorithm (`CO_2Way`), and (iv) our original cache-oblivious wavefront (COW) algorithms with atomic locks from [173]. For `wave-hybrid`, we have used an n' in the range of $128 - 2048$ ideally based on the best empirical running time. However a good guess of what this number will be is $n' = \max\left\{256, \text{power of 2 closest to } n^{2/3}\right\}$. In order to reduce overhead of recursion and to take advantage of vectorization we switch to an iterative kernel when n becomes sufficiently small (e.g., 64 for `wave`, `wave-hybrid` and `CO_2Way`). It is clear from the figures that `wave` and `wave-hybrid` algorithms perform better than the `CO_2Way` and COW algorithms for all cases. Almost always, the speedup numbers are better when input size n is small, since for those cases, improvement in parallelism matters the most.

On the Stampede 16-core machines, for parenthesis problem (Figure 5.2), `wave` is 2.6×, and `wave-hybrid` is 2× faster than `CO_2Way`. Similarly, number of cache misses of `CO_2Way` is slightly higher than that of both `wave` and `wave-hybrid`. For LCS (Figure 5.3) `wave` is 1.5×, and `wave-hybrid` is 1.7× faster than `CO_2Way`. We see similar trends for cache misses as well. For Floyd-Warshall's APSP (Figure 5.4), `wave` is 18%, and `wave-hybrid` is 10% faster than `CO_2Way`. Therefore, even with 16 cores, the impact of improvements in parallelism and cache-misses is visible on the running time. On the other hand, though COW algorithms have excellent theoretical parallelism, their implementations use a separate scheduler that heavily uses atomic

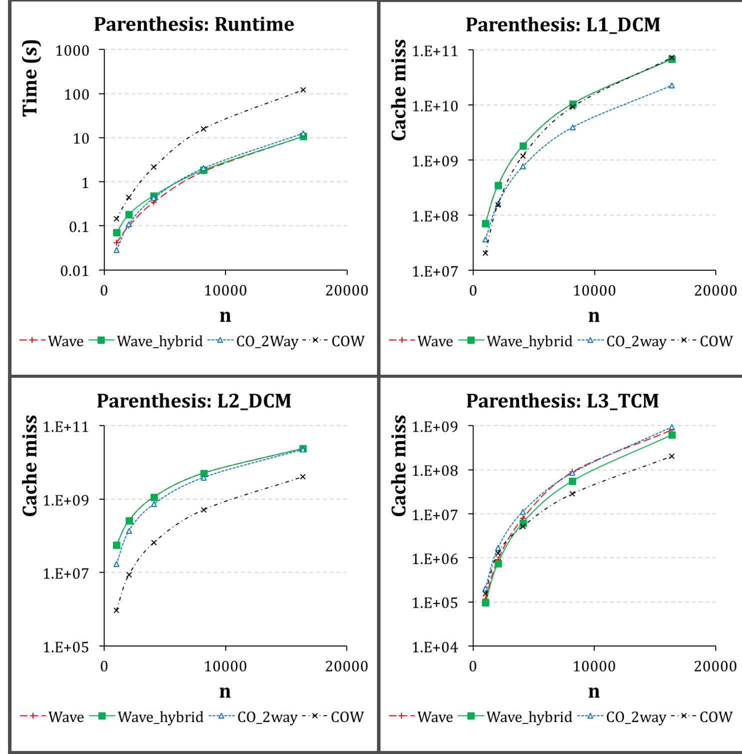


FIGURE 5.5: *Runtime and cache misses in three levels of caches for 2-way CO (CORDAC), COW and recursive wavefront algorithms for the Parenthesis problem. All programs were run on 24 core machines in Comet. All implementations used Cilk Plus's work-stealing scheduler.*

locks, which may have impacted their performance negatively for large n and for DP with dimension $d > 1$.

Figures 5.5, 5.6, and 5.7 show performance results obtained on a 24-core Haswell machine. Value of n' and size of the iterative kernel were determined in the same way as we did on Stampede. For FW-APSP (Figure 5.7), `wave` is 15% and `wave-hybrid` is 10% faster than `CO_2Way`. Although we see improvements in L1 and L2 cache misses, number of L3 misses is worse probably due to the increased parallelism. For parenthesis problem (Figure 5.5), `wave` is 16% and `wave-hybrid` is 18% faster than `CO_2Way`, and we see only improvement in the L3 cache misses.

On a 32-core Sandy Bridge machine, LCS `wave` runs 2.28× faster and `wave-hybrid` runs 2.21× faster than `CO_2Way`. Similarly, in all three levels of caches, `wave` algorithms incur fewer cache misses.

Algorithm	LCS	Parenthesis	FW_APSP
`wave`	509.9	1911.0	1404.2
`wave-hybrid`	152.3	821.8	276.7
`CO-2way`	17.8	22.5	147.7

TABLE 5.3: *Projected scalability of the new recursive wavefront algorithms computed by the Cilkview^{TM}Scalability Analyzer. The numbers denote till how many cores the implementation should scale linearly. The input size for LCS was 262144 and for both Parenthesis and FW-APSP, n was 16384.*

For the parenthesis problem both `wave` and `wave-hybrid` are 2.1× faster than `CO_2Way`. On the other hand, `wave` for FW-APSP runs 73%, and `wave-hybrid` runs 69% faster than `CO_2Way`. For FW-APSP the improvement is less, because the measured parallelism of the `CO_2Way` is > 140 (see Table 5.3), which is more than the number of cores we were using for these experiments.

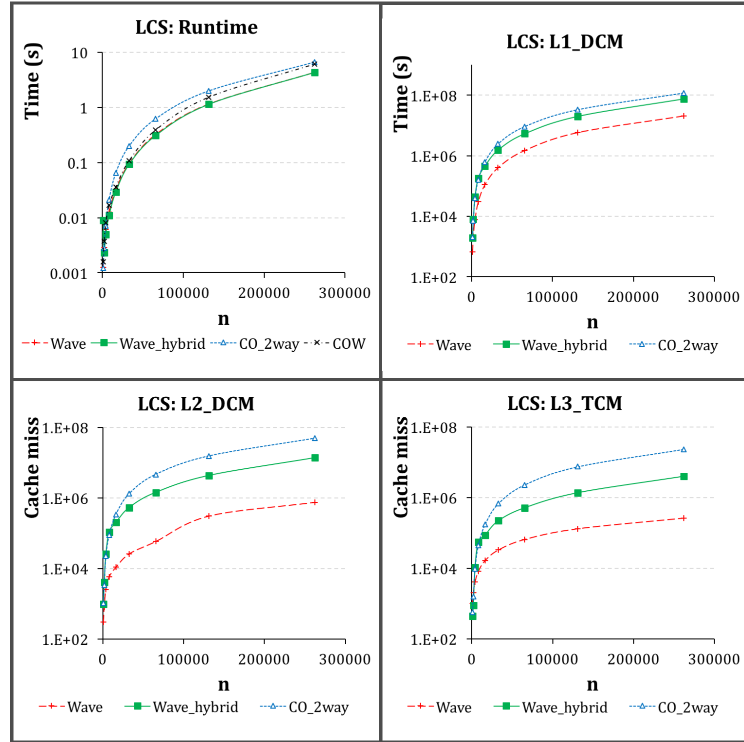


FIGURE 5.6: *Runtime and cache misses in three levels of caches for 2-way CO (COR-DAC), COW and recursive wavefront algorithms for the LCS Problem. All programs were run on 24 core machines in Comet. All implementations used Cilk Plus's work-stealing scheduler.*

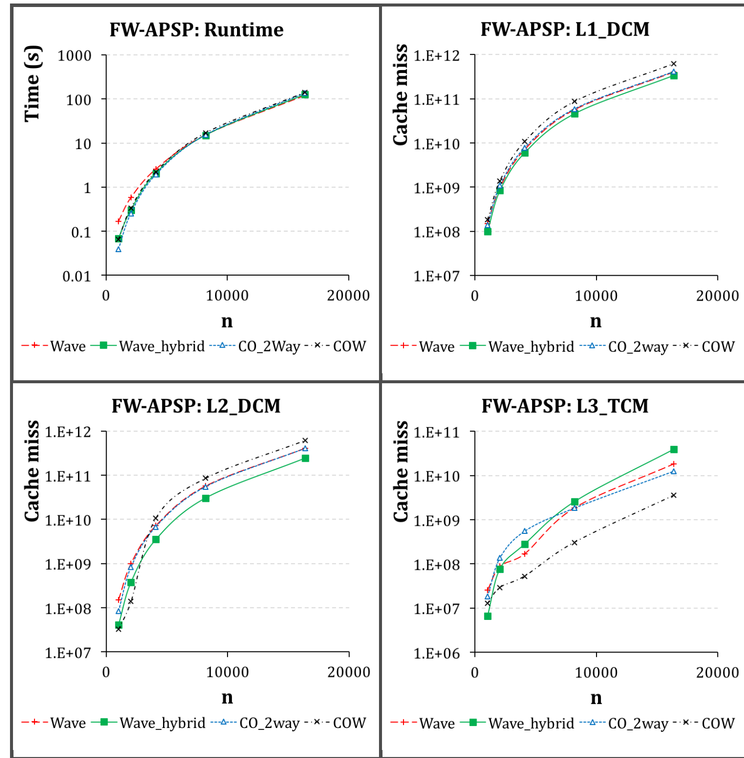


FIGURE 5.7: *Runtime and cache misses in three levels of caches for 2-way CO (COR-DAC), COW and recursive wavefront algorithms for the 2D FW-APSP Problem. All programs were run on 24 core machines in Comet. All implementations used Cilk Plus's work-stealing scheduler.*

Projected parallelism. We have used the Intel© Cilkview™scalability analyzer to compute the ideal parallelism and burdened span of our implementations. Table 5.3 shows the scalability results reported by Cilkview for all problems. These parallelism numbers show that recursive wavefront algorithms scale much better than standard 2-way recursive divide-and-conquer algorithms, making them a better choice for future multicores/manycores machines with thousands of cores.

5.7 Future Research

The next step in this research is to find out a way to automatically generate the complete-time, start and end timestamps. Then we can use them to develop an *autowave* system that can automatically convert a standard 2-way CORDAC algorithm to a cache-oblivious wavefront algorithm while guaranteeing cache-optimality with improved parallelism.

Chapter 6

An Efficient Cache-oblivious Viterbi Algorithm

6.1 Abstract

The dynamic programming (DP) algorithms that we have considered so far have very regular dependency structures (i.e., a cell in the DP table depends on some other cells deterministically). However, there are dynamic programming algorithms with important applications in bioinformatics for which the dependency structures are irregular (e.g., input data dependent). In this chapter we consider one such algorithm, the Viterbi algorithm and present an efficient cache-oblivious recursive divide-and-conquer technique to solve the underlying problem.

The Viterbi algorithm is used to find the most likely path through a hidden Markov model (HMM) given an observed sequence of events. This algorithm has numerous applications in bioinformatics including multiple sequence alignment, gene finding, CG island and conserved elements detection, protein secondary structure prediction and nanopore ionic flow blockades analysis. Due to its importance and computational complexity, several algorithmic strategies have been engineered to parallelize this algorithm on different parallel architectures. However, none of the existing algorithms for the Viterbi decoding problem is cache-efficient while being cache-oblivious (i.e., does not use cache-parameters in the algorithmic description). Being oblivious of machine resources (e.g., cache and processors) while also being efficient promotes portability and adaptivity. However, achieving better parallelism and cache-efficiency for the Viterbi algorithm is challenging due to the its irregular dependency pattern and no opportunity for data reuse.

In this chapter, we present an efficient cache- and processor-oblivious recursive divide-and-conquer Viterbi algorithm that uses the rank convergence property of matrix operations to achieve cache-efficiency [1]. The algorithm builds upon the parallel Viterbi algorithm of Maleki et al. (PPoPP 2014). We provide empirical analysis of our algorithm showing that our algorithm outperforms the prior best performing algorithm in terms of cache-performance, running time and energy efficiency.

[1] Jesmin Jahan Tithi and Pramod Ganapathi are major contributors in this work.

6.2 Introduction

The Viterbi algorithm [78, 186, 187] proposed by Andrew J. Viterbi in 1967 is a dynamic programming algorithm that finds the most probable sequence of hidden states, called "Viterbi path" for a given sequence of observed events in the context of a hidden Markov model (HMM). The hidden Marov model consists of three phases: the forward evaluation phase, the backward decoding phase and the learning phase. The Viterbi algorithm is used in the decoding phase to find the most probable path through a probabilistic HMM model and is also one of the most compute intensive kernels among these three.

The Viterbi algorithm is an example of a dynamic programming problem where the cell dependencies have irregular pattern and are data dependent. Furthermore, it requires to scan $\Theta(n^2)$ data per $\Theta(n^2)$ computations, leaving no scope for data-reuse. Therefore, it is not clear how to get temporal locality while solving Viterbi problem, since to get temporal locality we need a data read/write to computation ratio of $\omega(1)$.

Motivation. The Viterbi algorithm has numerous real world applications. Although it was originally used for speech recognition in CDMA technology [74, 87, 111, 151, 163], from the year of 1990, it had been heavily used in computational biology and bioinformatics to find the coding and non-coding regions of an unlabeled string of DNA nucleotides (i.e., gene finding) [32], prediction of protein-coding regions in genome sequences modeling families of related DNA or protein sequences and prediction of secondary structure elements in proteins [115], CpG island [67], promoter [140] and conserved elements detection [160]. Using the Viterbi algorithm, we can compute the most likely alignment of a sequence against a sequence family. Viterbi algorithm can also be used to build multiple alignments to compute an optimal alignment among a group of sequences [145].

For the sequence alignment application Satchmo [69], typically the last two phases of HMM are needed and out of those two phases, Viterbi algorithm has been found to consume 80% of the overall computational time [64]. Furthermore, biological sequences tend to be longer (consider annotating all 250 million symbols of the human chromosome 1 with a gene finding HMM which consists of hundred of states). Therefore, it is very important to develop fast and efficient Viterbi algorithm that can handle large number of states and timesteps targeting modern parallel architectures (e.g., multicores and manycores).

Problem Specifiction. A formal definition of the problem that the Viterbi algorithm solves is as follows: we are given an observation space $O = \{o_1, o_2, \ldots, o_m\}$, state space $S = \{s_1, s_2, \ldots, s_n\}$, observations $Y = \{y_1, y_2, \ldots, y_t\}$, transition matrix A of size $n \times n$, where $A[i, j]$ is the transition probability of transiting from s_i to s_j, emission matrix B of size $n \times m$, where $B[i, j]$ is the probability of observing o_j at state s_i, and initial probability vector (or initial solution vector) I, where $I[i]$ is the probability that the initial state, x_1 is s_i. Let $X = \{x_1, x_2, \ldots, x_t\}$ be a sequence of hidden states that generates $Y = \{y_1, y_2, \ldots, y_t\}$. Then the matrices P and P' are of size $n \times t$, where $P[i, j]$ is the probability of the most likely path of getting to state s_i at observation y_j, and $P'[i, j]$ stores the hidden state of the most likely path. Using the Viterbi algorithm, we are interested in computing values for P and P'. Form now on, we will call this problem as the *Viterbi problem*.

An example of the Viterbi problem can be this: a human wearing sunglass, taking umbrella or wearing hand-gloves can be considered as observation states (O) and cloudy, raining, sunny can be considered as hidden states (S). Then emission matrix B gives probability of "wearing sunglass" or "taking umbrella" given the state is "raining". Transition matrix A gives probability of going to state "rainy" from state "cloudy", "cloudy" to "sunny", etc. The initial probability vector gives the probably of getting states "cloudy", "rainy" or "sunny". Now if the observation Y is equal to "wearing hand-glove", using Viterbi algorithm, we will find the most probable sequence of states that led to this observation "wearing hand-glove" given the initial probability of each state.

The $P[i,j]$ and $P'[i,j]$ matrices can be computed using the following recurrences:

$$P[i,j] = \begin{cases} I[i] \cdot B[i,y_1] & \text{if } j = 1, \\ \max\limits_{k \in [1,n]} (P[k,j\text{-}1] \cdot A[k,i] \cdot B[i,y_j]) & \text{if } j > 1. \end{cases}$$

$$P'[i,j] = \begin{cases} 0 & \text{if } j = 1, \\ \arg\max\limits_{k \in [1,n]} (P[k,j\text{-}1] \cdot A[k,i] \cdot B[i,y_j]) & \text{if } j > 1. \end{cases}$$

A simple iterative solution implementing this recurrences will take $\Theta(n^2 t)$ time and in general, will be cache inefficient. In this chapter we show how to design an efficient cache-oblivious recursive divide-and-conquer algorithm to solve this Viterbi problem. In the following sections we discuss some prior work followed by our algorithm to solve the Viterbi Problem.

```
VITERBI-RANK(P[0..t − 1], A, B)
1.  p ← #processors
    // Forward phase
2.  parallel for i ← 1 to p do
3.      l_i ← t(i − 1)/p; r_i ← ti/p
4.      if i > 1 then P[l_i] ← random vector
5.      for j ← l_i to r_i − 1 do
6.          P[j + 1] ← VITERBI(P[j], A, B[.., y_{j+1}])
    // Fixup phase
7.  converged ← false
8.  while !converged do
9.      parallel for i ← 2 to p do
10.         conv_i ← false; s ← P[l_i]
11.         for j ← l_i to r_i − 1 do
12.             s ← VITERBI(s, A, B[.., y_{j+1}])
13.             if s is parallel to P[j + 1] then
14.                 conv_i ← true; break
15.             P[j + 1] ← s
16.     converged ← ∧_i conv_i
```

FIGURE 6.1: *Processor-aware parallel Viterbi algorithm using rank convergence as given in Maleki et al. paper [126]. This algorithm is not cache-efficient.*

6.3 Viterbi algorithm using rank convergence

In this section we describe Maleki et al.'s algorithm [126] for solving the Viterbi problem which uses rank-convergence properties of matrices over a closed tropical semiring where the multiply symbol ($\times$) is replaced by a plus ($+$) symbol and the sum ($\sum$) is replaced by a *max* in the standard matrix multiplication. Almost all research before Maleki et al.'s work used a serial execution order across the timesteps due to the presence of strict dependencies among them, parallelizing only inside a time-step using different methods. Therefore, the parallelization opportunity was quite restricted in those cases. Maleki et al. [126] used the rank convergence property of a sequence of matrix operations over a closed tropical semiring to extract parallelism across different time-steps in the Viterbi algorithm. They generalized this

idea to a class of dynamic programming problems called LTDP (linear tropical dynamic programming problems) which includes LCS, Smith-Waterman and Needleman-Wunsch as well.

Viterbi recurrence using log-likelihood. Almost all practical implementations of Viterbi algorithm use log-probabilities (i.e., logarithm of all probability values) instead of the original probability, and additions instead of the multiplications in the DP recurrence. This makes the computations faster as well as more accurate as multiplication on probability values are lossy due to fixed point precisions. In that case the Viterbi recurrence looks like the following:

$$P[i,j] = \begin{cases} I[i] + B[i, y_1] & \text{if } j = 1, \\ \max_{k \in [1,n]} (P[k, j\text{-}1] + A[k,i] + B[i, y_j]) & \text{if } j > 1. \end{cases}$$

The above recurrence can be rewritten as a chain of matrix multiplications as follows:

$$P[t-1] = P[0] \odot AB_1 \odot AB_2 \odot \cdots \odot AB_{t-1}$$

where $P[j]$ is the jth solution vector (and the column vector $P[.., j]$) of matrix P, the $n \times n$ matrix AB_i is a suitable combination of A and B, and $\odot$ is a matrix operation defined between two matrices $R_{n\times n}$ and $S_{n\times n}$ as

$$(R \odot S)[i,j] = \max_{k \in [1,n]} (R[i,k] + S[k,j])$$

which is the same as the tropical semiring product operation as mentioned at the beginning of this section.

Before describing the algorithm, we will provide a few important definitions from the Maleki et al.'s paper which are needed to understand the algorithm. The *rank* of a matrix $A_{m\times n}$ is r, if r is the smallest number such that A can be written as a product of two matrices $C_{m\times r}$ and $R_{r\times n}$, i.e., $A_{m\times n} = C_{m\times r} \odot R_{r\times n}$. Two vectors v_1 are v_2 are *parallel*, if v_1 and v_2 differ by a constant offset. For example, [1,2,3,4] and [5,6,7,8] are parallel vectors, because they differ by a constant offset 4.

A property of the product operation under the tropical semiring is that, the rank of the product of two matrices is always smaller or equal to the rank of the individual matrix (almost always the rank reduces), i.e., $rank(AB) \leq rank(A) \odot rank(B)$. As a result, the rank of a sequence of matrix operations is likely to become 1 eventually, which is called rank-convergence. In a rank 1 matrix, all the column vectors are parallel to each other. Similarly, all row vectors are also parallel to each other. A nice property of a rank 1 matrix is that, all non-zero random vectors multiplied ($\odot$) by a rank 1 matrix will produce parallel vectors (it is similar to mapping all vectors to parallel lines).

6.3.1 Maleki et. al.'s algorithm using rank-convergence

Maleki's algorithm (see Figure 6.1) consists of two phases: (i) parallel forward phase, and (ii) parallel fixup phase. In the forward phase, the t stages (total timesteps) are divided into p segments, where p is the number of processors, each segment having $\lceil t/p \rceil$ stages (except possibly

the last stage). The stages in the ith segment consists of columns l_i (exclusive) to r_i (inclusive) from matrix P. The initial vector of the first segment is the initial vector for the entire DP problem, which is known. The initial solution vectors of all other segments are initialized to non-zero random valid probability values. A sequential Viterbi algorithm is run in all the segments in parallel. A timestep i is said to converge if the computed solution vector s_i is parallel to the actual solution vector $P[i]$. A segment i is converged if rank of $(AB_{l_i} \odot AB_{l_i+1} \odot \cdots \odot AB_j)$ is 1 for $j \in [l_i, r_i - 1]$. After the forward pass, possibly only the first segment will have correct log-probability values, since all other segments started with random initial values.

In the fixup phase, as in the forward phase a sequential Viterbi algorithm is executed for all segments simultaneously except the first segment, taking the solution vector from the prior segment computed in the previous phase (forward/fixup phase) in a temporary storage. After that, the newly computed solution vectors are compared with the solution vectors available in the original DP table. If they are parallel for all segments, the program terminates as the solutions have converged. Otherwise, the non-converged old values in the DP table are replaced with the newly computed values, and a new fixup phase starts.

Though solution vectors computed in different segments might be wrong, eventually they become parallel to the actual solution vectors either after the $p-1$ fixup phases or earlier if *rank convergence* occurs at an earlier fixup phase. Note that the $(i+1)^{th}$ segment always gets fixed in the ith fixup phase. In the worst case, which rarely happens in practice, the fixup phase executes $p - 1$ times. Please refer to Maleki et al.'s paper [126] for a proof of why the method works.

```
VITERBI-RANK-IMPROVED(s[0..t-1], A, B)
1.  n ← 2^k; t ← 2^(k+k'); c ← 2^8
    // Forward phase
2.  size ← c; q ← t/size
3.  parallel for i ← 0 to q − 1 do
4.      l_i ← i × size, r_i ← l_i + size − 1
5.      if i > 0 then s[l_i] ← random vector
6.      for j ← l_i to r_i do
7.          s[j + 1] ← VITERBI(s[j], A, B[.., y_{j+1}])
    // Fixup phase
8.  u[0..t − 1] ← s[0..t − 1]; converged ← false
9.  for (j ← log c to (log t) − 1) and !converged do
10.     size ← 2^j; q ← t/(2 × size)
11.     parallel for i ← 0 to q − 1 do
12.         l_i ← (2i+1) × size − 1; r_i ← l_i + size; conv_i ←
            false
13.         for j ← l_i to r_i do
14.             u[j + 1] ← VITERBI(u[j], A, B[.., y_{j+1}])
15.             if u[j + 1] is parallel to s[j + 1] then
16.                 conv_i ← true; break
17.             s[j + 1] ← u[j + 1]
18.     for i ← 0 to q − 1 do
19.         converged ← converged ∧ conv_i
20.     if converged = true then break
```

FIGURE 6.2: *Processor-oblivious parallel cache-inefficient Viterbi algorithm using rank convergence.*

6.3.2 An improved processor-oblivious algorithm

The algorithm described in Section 6.3.1 is processor-aware and can be made processor-oblivious by setting p to some constant. We can chose a suitable segment size c (say, 256) that is large enough, then use a parallel for loop to solve those t/c segments simultaneously. Unlike Maleki et al.'s algorithm, we need to make sure that the segments are non-overlapping at their boundaries and then adjust the fixup phase accordingly as shown in Figure 6.2.

```
VITERBI-MI(P_1, P_2, ..., P_q, A, B, t) //Cache-oblivious Divide and Conquer Based Viterbi-Multi-Instance
1. for j ← 2 to t do
2.     X ← [P_1[.., j], P_2[.., j], ..., P_q[.., j]]
3.     U ← [P_1[.., j − 1], P_2[.., j − 1], ..., P_q[.., j − 1]]
4.     V ← A
5.     W ← [B[.., y_1j], B[.., y_2j], ..., B[.., y_qj]]
6.     A_vit(X, U, V, W)
-----------------------------------------------------------------
A_vit(X_{n×q}, U_{n×q}, V_{n×n}, W_{n×q})
1. if X and V are small matrices do
2.     A_loop−vit(X, U, V, W)
3. else if q > n do
4.     parallel A_vit(X_L, U_L, V, W_L), A_vit(X_R, U_R, V, W_R)
5. else if q < n do
6.     parallel A_vit(X_T, U_T, V_11, W_T), A_vit(X_B, U_T, V_12, W_B)
7.     parallel A_vit(X_T, U_B, V_21, W_T), A_vit(X_B, U_B, V_22, W_B)
8. else
9.     parallel A_vit(X_11, U_11, V_11, W_11), A_vit(X_12, U_12, V_11, W_12), A_vit(X_21, U_11, V_12, W_21), A_vit(X_22, U_12, V_12, W_22)
10.    parallel A_vit(X_11, U_21, V_21, W_11), A_vit(X_12, U_22, V_21, W_12), A_vit(X_21, U_21, V_22, W_21), A_vit(X_22, U_22, V_22, W_22)
```

FIGURE 6.3: *Cache-efficient parallel recursive divide-and-conquer multi-instance Viterbi algorithm.*

Here is how the algorithm works. Let the initial segment size be c (i.e., c consecutive timesteps). For convenience we chose $c = 2^i$, where $i \in [\log c, \log t]$. We divide t timesteps into t/c independent segments each of size c. Similar to Maleki et. al.'s algorithm, the first solution vectors of all except the first segment are initialized to non-zero valid random probability values. Then in the forward phase we run serial viterbi algorithm on all of the t/c segments of size c simultaneously. At the end of the forward phase solution vectors till the cth column (i.e., all columns in the first segment) will have correct log-likelihood values. Other segments will have values computed from the random values chosen initially which may or may not be parallel to the expected values.

In the fixup phase, we start fixing from the second segment as the original Maleki's algorithm. However, in each fixup phase, we work on alternative segments always leaving the first segment of the prior fixup phase. After each fixup phase, the size of each segment being considered is doubled and therefore, the number of segments becomes half with respect to the previous phase. At the end of each fixup phase, we check whether the computed solution vectors are parallel to that of the prior phase (forward or fixup phase), and if the answer is yes for all segments under consideration, the program terminates. Otherwise, the next fixup phase starts. In the worst case, the fixup phase is executed for $\max(1, \log(t/c))$ times after which all results are guaranteed to be correct since by that time the result from the original input has been propagated till the end. Hence, in the worst case, the program is like a serial Viterbi algorithm with a $\log(t/c)$ factor overhead.

6.4 Cache-efficient multi-instance Viterbi algorithm

In this section, we discuss a cache-efficient Viterbi algorithm that can compute the most probable path for multiple instances of the problem at once. Since in Viterbi algorithm we need to scan $\Theta(n^2)$ data to compute $\Theta(n^2 t)$ times, the only way we can get temporal locality is from the time dimension t. However, the time dimension has a strict sequential dependency. Another way to get temporal locality is to solve multiple instances of the problem that have the same A and B

matrices (but different observation vectors) at once, so that the $\Theta(n^2)$ reads can be reused. For example, if we solve n instances of the problem, by scanning the transition matrix A only once, a particular column of matrix P can be computed for all n instances of the problem at once, thus efficiently amortizing the cost of data loading.

Two problems that have the same transition matrix A and emission matrix B can be considered as two instances of the same problem. An example of the multi-instance Viterbi problem is the problem of multiple sequence alignment. Different biological sequences can be considered as separate observations Ys of the same problem, with the same transition and emission matrices A and B (in fact values in the B matrices for different instance can be different, provided the states are the same). Species with similar biological properties can fit in this form. Another possible usecase would be finding the most probable state sequence for many cancer genes taken from the same human body.

Figure 6.3 gives a cache-efficient and cache- and processor-oblivious recursive divide-and-conquer (CORDAC) Viterbi algorithm that can be used to solve q instances of the problem at once. To exploit temporal cache locality, it solves q instances of the problem simultaneously, where $q = \Omega(n^x)$ and $x > 0$, which increases computational work to $O(n^2 q)$ in the two innermost loops. Total space required for this problem is $O(n^2 + qt + qn)$.

In function $\mathcal{A}_{vit}(X, U, V, W)$ (see Figure 6.3), the matrix U is an $n \times q$ matrix obtained by concatenating $(j-1)^{th}$ columns of q matrices $P_1, P_2, \ldots, P_q$, where P_i is the most likely path probability matrix of problem instance i. The algorithm computes X, which is a concatenation of jth columns of the q problem instances. Each problem instance i has a different observations vector $Y_i = \{y_{i1}, y_{i2}, \ldots, y_{it}\}$. The matrix W is a concatenation of y_j^{th} columns of matrix B obtained from different observations i.e., $B[y_{1j}], B[y_{2j}], \ldots, B[y_{qj}]$. We use X_T, X_B, X_L, and X_R to represent the top half, bottom half, left half, and right half of the matrix X, respectively. Executing the divide-and-conquer algorithm once computes the second column of all matrices P_1 to P_q. Executing the algorithm again computes the third column of the q matrices. Executing the algorithm $t-1$ times will fill the last column of all problem instances with the final log-likelihood values. Note that for each timestep, the matrix W should be constructed again and again.

It is important to note that the structure of the function $\mathcal{A}_{vit}$ is similar to the recursive divide-and-conquer-based in-place matrix multiplication algorithm (MM-kernel). When $q = n$, both algorithms have 8 recursive function calls in two parallel steps. Therefore, the complexity analysis of the multi-instance Viterbi algorithm will be similar to that of the matrix multiplication algorithm, i.e., $\mathcal{O}\left(\frac{n^2 q}{B\sqrt{M}}\right) t$. The $\sqrt{M}$ factor in the denominator tells us that the algorithm has temporal locality. Note that since the matrix multiplication kernel is flexible and highly optimizable (see Chapter 3), the algorithm presented in Figure 6.3 also has similar properties.

In the next section we show how to use this cache-efficient multi-instance Viterbi algorithm to solve the original single instance Viterbi problem cache-efficiently by leveraging the rank-convergence property of a sequence of matrix operations.

6.5 Cache-efficient Viterbi algorithm

In this section we present a cache-efficient cache- and processor-oblivious recursive divide-and-conquer parallel Viterbi algorithm that uses the rank-convergence property of a sequence of matrix operations to compute values for multiple timesteps in parallel. This algorithm uses the cache-efficient multi-instance Viterbi algorithm as a sub routine which leads to cache-optimality. Here is how we apply the multi-instance Viterbi algorithm to solve the standard (i.e., single instance) Viterbi problem.

We divide t timesteps into t/c independent segments each of size c (i.e., c consecutive timesteps). Let the initial segment size is c. For convenience we chose $c = 2^i$, where $i \in [\log c, \log t]$. Similar to the algorithm presented in Figure 6.2, the first solution vectors of all except the first segment are initialized to non-zero random valid probability values. As each segment is independent, we can assume that these segments are different instances of the same problem who also have the same A and B matrices. Therefore, we can use the cache-efficient multi-instance Viterbi algorithm, Viterbi-MI to solve all these t/c instances at once.

In the forward phase, a multi-instance Viterbi algorithm (see Figure 6.2) is run assuming each of the t/c segments as an independent problem instance. The jth columns of all segments are considered as the input, and at the end of this phase the solution vectors ($(j+1)^{th}$ columns) for each of those segments are produced as output, where $0 \leq j < c$. The fixup phase is similar to the improved Maleki's algorithm shown in Figure 6.2, expect that now we use cache-efficient multi-instance Viterbi algorithm to compute the next solution vectors of all segments at once instead of using a serial iterative Viterbi algorithm to compute the entire segment independently. As before, we start fixing from the second segment since the first segment is already fixed after the forward phase. In each fixup phase, we work on alternative segments always leaving out the first segment of the prior phase (already fixed by this time). After each fixup phase, the size of each segment being considered is doubled (i.e., from 2^i it becomes 2^{i+1}) and as a result, the number of segments becomes half (i.e., from $t/(2^i)$ to $t/(2^{i+1})$) with respect to the previous phase. For each step, we use cache-efficient multi-instance Viterbi algorithm (Viterbi-MI) to compute the solution vectors for all segments at once. At the end of each fixup phase, we

```
Viterbi-Cache-Efficient(s[0..t − 1], A, B)
1.  n ← 2^k; t ← 2^{k+k'}; c ← 2^8
    // Forward phase
2.  size ← c; q ← t/size
3.  parallel for i ← 0 to q − 1 do
4.      l_i ← i × size, r_i ← l_i + size − 1
5.      if i > 0 then s[l_i] ← random vector
6.  Viterbi-MI(s[l_0..r_0], s[l_1..r_1], .., s[l_{q-1}..r_{q-1}], A, B, c)
    // Fixup phase
7.  u[0..t − 1] ← s[0..t − 1];
8.  for (j ← log c to (log t) − 1) and !converged do
9.      size ← 2^j, q ← t/(2 × size)
10.     parallel for i ← 0 to q − 1 do
11.         l_i ← (2i + 1) × size − 1; r_i ← l_i + size; conv_i ← false
12.     Viterbi-MI(u[l_0..r_0], u[l_1..r_1], .., u[l_{q-1}..r_{q-1}], A, B, size+1)
13.     parallel for i ← 0 to q − 1 do
14.         r_i ← 2(i + 1) × size − 1
15.         if u[r_i] is parallel to s[r_i] then conv_i ← true
16.         else s[r_i] ← u[r_i]
17.     for i ← 0 to q − 1 do
18.         converged ← converged ∧ conv_i
19.     if converged = true then break
```

Figure 6.4: *An efficient cache- and processor-oblivious parallel Viterbi algorithm using rank convergence.* Viterbi-MI *refers to* Viterbi-MI *algorithm presented in Section 6.4.*

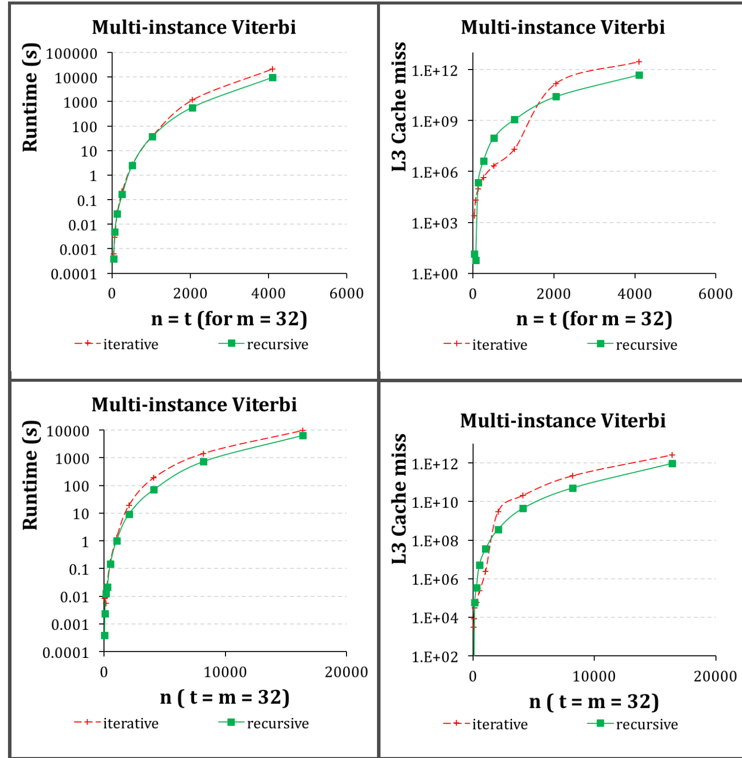


FIGURE 6.5: *Running time and L3 miss of our cache-efficient multi-instance Viterbi algorithm and the multi-instance iterative Viterbi algorithm.*

check whether the computed solution vectors are parallel to those found in the prior phase, and if the answer is "yes" for all segments under consideration, the program terminates. Otherwise, the next fixup phase starts. In the worst case, the fixup phase is executed $\max(1, \log(t/c))$ times, after which all results are guaranteed to be correct.

6.6 Experimental results

In this section, we briefly describe our implementation details and performance results. We implemented all algorithms presented in this chapter in C++ with Intel® Cilk™Plus [3] extension and compiled them using Intel® C++ Compiler v13.0. We used PAPI 5.2 [4] to count the cache misses and LIKWID [184] to measure energy and power consumption of the program. We used a hybrid recursive divide-and-conquer (CORDAC) algorithm where the recursive implementation switched to an iterative kernel when the problem size became smaller than a predefined basecase size (e.g., 64×64) to amortize the overhead of recursion and allow vectorization. All programs were compiled with `-O3 -parallel -AVX -ansi-alias -opt-subscript-in-range` optimization parameters and were auto vectorized by the compiler.

We used a dual socket 16-core (= 2×8-cores) 2.7 GHz Intel Sandy Bridge machine to run all experiments presented in the paper. Each core of this machine was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All the cores in a socket shared a 20 MB 10-way L3 cache, and the machine had 32 GB RAM shared by all cores.

The matrices A, B and I were initialized to random valid probabilities. We used log-probabilities in all implementations and hence used addition instead of multiplication in the Viterbi recurrence.

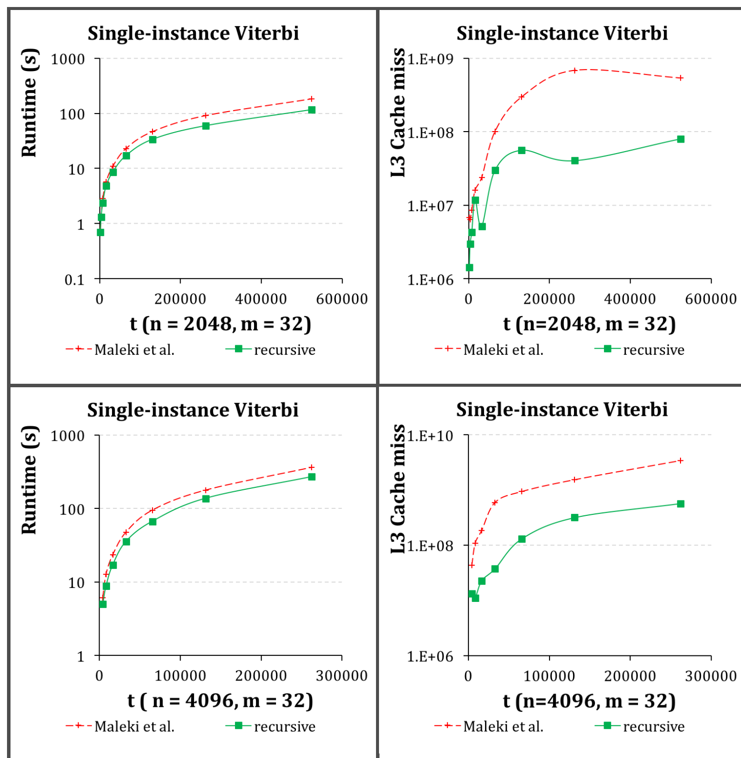


FIGURE 6.6: *Running time and L3 miss of our cache-efficient Viterbi algorithm and comparison with Maleki et. al.'s algorithm.*

All matrices were stored in column-major order. We performed two sets of experiments to compare our cache-efficient algorithms with the iterative and the fastest known [126] Viterbi algorithms. We discuss the results in the following section.

6.6.1 Multi-instance Viterbi: Iterative vs. Recursive

We compared our cache-efficient multi-instance recursive Viterbi algorithm with the multi-instance iterative Viterbi algorithm. Both algorithms were moderately optimized. To construct matrix $W_{n \times q}$, we used a list of pointers to the respective columns and avoided direct copying which saved frequent copy overhead. Wherever possible, we used pointer swapping to interchange previous solution vector (or matrix) with the current solution vector (or matrix).

The running time and the L3 cache misses for the two algorithms are plotted in Figure 6.5. The number of states (n), number of timesteps (t) and number of instances (q) for those experiments were the same (hence, the overall complexity is $O(n^4)$), and varied from 32 to 4096. The variable m was fixed to 32. Although in the cache-efficient multi-instance Viterbi algorithm, the number of stages does not need to be the same as the number of instances, we used $n = q$ for convenience. Note that, for biological sequence matching problems, t can be in the order of billions, and n can be in the order of thousands.

The cache-efficient recursive algorithm ran faster than the multi-instance iterative algorithm in most of the cases and at data point $n = q = t = 2048$, our recursive algorithm ran around $3\times$ faster than the parallel iterative algorithm.

6.6.2 Single-instance Viterbi: Efficient recursive vs. Maleki et. al.'s

We compared our cache-efficient parallel Viterbi algorithm with Maleki et al.'s parallel Viterbi algorithm. Both implementations were optimized similarly and the reported statistics are average of 4 independent runs. In all experiments, the number of processors p was set to 16. Figure 6.6 shows the running time and L3 cache misses for the two algorithms when $n = 2048$ and 4096.

When $n = 2048$, the number of timesteps t was varied from 2^{11} to 2^{19} and m was set to 32. Our algorithm ran faster than Maleki et al.'s original rank convergence algorithm throughout, and for $t = 2^{19}$ our algorithm ran approximately 57% faster. Our algorithm's L3 cache misses were also lower by a significant amount, and for $t = 2^{19}$, Maleki et al.'s algorithm incurred 6.7 times more cache misses than ours.

Similarly, when $n = 4096$, the trends remained similar. The number of timesteps t was increased from 2^{12} to 2^{18}. At $t = 2^{18}$, our algorithm ran 33% faster, and incurred a factor of 6 fewer L3 misses than Maleki et al.'s rank convergence algorithm.

Energy consumption. We also ran experiments to analyze the energy consumption (taking average over three runs) of our cache-efficient recursive and Maleki et. al.'s algorithm. We used the LIKWID tool to measure CPU, Power Plane 0 (PP0), DRAM energy, and DRAM power consumption during the execution of each program. The energy measurements were end-to-end, i.e., included all costs during the entire program execution.

Note that the DRAM energy consumption is somewhat related to the L3 cache miss of a program as each L3 cache miss results in a DRAM access. Similarly, since Package/CPU energy gives the energy consumed by the entire package (all cores, on chip caches, registers and their interconnections), it is related to a program's running time. PP0 is basically a subset of CPU energy since it captures energy consumed by only the cores and their private caches.

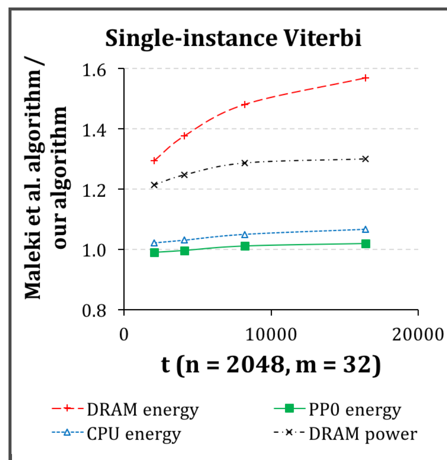


Figure 6.7: *Energy / power consumption of our and Maleki et al.'s algorithms.*

For $n = 2048$, the timesteps was increased from 2048 to 16384 keeping $m = 32$. Figure 6.7 gives the ratio of the energy and power consumption of Maleki et al.'s algorithm with that of ours for all three types of energy and power. The DRAM energy as well as power consumption of our algorithm were significantly less because of the reduced L3 cache misses. When $t = 16384$, Maleki et al.'s algorithm consumed 60% more DRAM energy and 30% more DRAM power than ours. The reduction in package/CPU energy was 6%.

6.7 Conclusions and Future Research

Solving the Viterbi decoding problem efficiently is very important for bioinformatics. In this chapter we proposed the first processor- and cache-oblivious efficient Viterbi algorithm. The

algorithm combines the ideas of the cache-efficient multi-instance Viterbi algorithm with the rank-convergence used by Maleki et al.'s parallel Viterbi algorithm. The significance of our algorithm lies mainly in its improved cache complexity, and cache- and processor-obliviousness, which translate to better runtime, energy and power metrics. It will be interesting to find out the cache-adaptivity and bandwidth-performance of the presented algorithms. Extending this algorithm to manycores and distributed-memory settings is also interesting. Another possible extension is to solve other irregular dynamic programming problems such as the knapsack problem using a similar algorithmic technique that we have used here.

Part II

Algorithms on Graphs

Chapter 7

Optimistic Parallelization: Avoiding Locks and atomic instructions in Shared-memory Parallel BFS

7.1 Abstract

Breadth-first search (BFS) has numerous applications in bioinformatics including analysis of biological interaction networks, metabolic pathway search, finding minimum gene subsets, betweenness centrality, searching in tries, and so on. In this research we show how to use optimistic parallelization to avoid the use of locks and atomic instructions during dynamic load-balancing in level-synchronous parallel breadth-first search algorithms.

Dynamic load-balancing in parallel algorithms typically requires locks and/or atomic instructions for correctness. We show that sometimes an *optimistic parallelization* technique can be used to avoid the use of locks and atomic instructions during dynamic load-balancing which results in improvement in scalability and performance. In *optimistic parallelization* one allows potentially conflicting operations to run in parallel with the hope that everything will run without conflict, and if any occasional inconsistencies arise due to conflicts, one will be able to fix them without hampering the overall correctness of the program. We use this approach to implement two new types of high-performance lockfree parallel BFS algorithms and their variants based on *centralized job queues* and *distributed randomized work-stealing*, respectively. All of these algorithms are cache-oblivious and one of them uses recursive divide-and-conquer for dynamic load-balancing.

We derive theoretical performance bounds and prove correctness of our algorithms. We also present experimental results showing scalability of our algorithms on state-of-the-art multicore and manycore (Xeon Phi) machines, using several parallel programming platforms (cilk++, cilk plus, OpenMP) and on various kinds of graphs. Our implementations generally run faster than parallel BFS algorithms by Hong et. al. (PACT, 2011) and Leiserson and Schardl (SPAA, 2010)

where the latter paper is also free of locks and atomic instructions but does not use optimistic parallelization.

7.2 Introduction

Optimistic parallelization is an approach where we allow parallel execution of potentially conflicting code blocks provided we know how to handle conflicts if they actually arise [136]. In this approach, threads modify shared data optimistically and try to detect conflicts, and if conflicts arise they undo the modifications or take recovery steps. Such parallelization is specifically useful for irregular problems where it is hard to exploit fine-grained parallelism [116].

Different variations of optimistic parallelization approaches have been used for solving different problems including Delaunay mesh refinement, image segmentation using graphcuts, agglomerative clustering, and Delaunay triangulation [116], [136] and so on. However, all of them typically use locks or atomic instructions while recovering from conflicts. We show that sometimes problem-specific properties can be used to avoid the use of locks and atomic instructions even during the handling of inconsistencies in the data structure arising from the unprotected concurrent updates. We demonstrate how to use optimistic parallelization technique to avoid locks and atomic instructions during dynamic load-balancing on shared-memory multicores and manycores. For this we use *breadth-first search* – a popular graph traversal problem known to be hard to parallelize efficiently because of its irregularity. Our experimental results show that optimistic parallelization often leads to better overall load-balancing.

Breadth-first Search (BFS) is one of the basic graph search algorithms in which we explore a graph systematically level by level from a source vertex. Graphs are often used as the fundamental representational tool for solving problems in a wide range of application areas such as analyzing social networks [133], consumer-product web analysis, computational biology [129], intelligent analysis, robotics, network analysis, and even in image processing [167]. All these applications require handling of massive data and traditionally demand longer processing time and other computational resources. Therefore, efficient parallelization of graph processing algorithms such as BFS is of utmost importance in many of these application areas. BFS is used as a building block for several other important algorithms such as finding shortest paths and connected components, graph clustering [19], community structure discovery, max-flow computation and the betweeness centrality problem [80]. High performance BFS is also used in pathminer [129], for pattern searching in DNA/RNA strings using Trie [76], and for the execution of range queries of an MVP-index [127]. BFS is being used as a graph benchmark application for ranking supercomputers ([15], [13]), too.

BFS belongs to the class of parallel algorithms where the memory accesses and work distribution are both irregular and data-dependent [131]. In the standard serial BFS algorithm, a FIFO queue is used to sequentially explore the vertices of a graph level by level from a source vertex, and find the levels (or unweighted shortest distances) of other vertices from the source. In a parallel BFS algorithm, typically all vertices of a given level and all neighbors of a given vertex can be explored in parallel which gives the best theoretical bounds of $O(\frac{m+n}{p} + D \log p)$ where m denotes the number of edges, n is the number of vertices, D is the diameter of the graph, and

p denotes the number of cores used to run the BFS. Parallel BFS algorithms are typically level-synchronous, i.e., a synchronization barrier is used after each level of the breadth-first search. This also means that there is a synchronization overhead of $O(\log p)$ after each BFS level which increases linearly with the diameter D of the graph. Furthermore, the amount of parallelism achievable at each BFS level is constrained by the number of nodes/vertices at that particular level making performance of a BFS algorithm highly dependent on the structure of the graph itself.

Various techniques have been engineered so far to parallelize BFS algorithms, such as use of distributed-memory parallelism [167, 194], shared-memory parallelism [9, 15, 18, 105, 119], centralized queues, distributed queues [11], and complicated concurrent data structures [119]. Several GPU-based implementations of BFS have also been proposed [105, 131]. Each year a number of new approaches are proposed on efficient and scalable parallel BFS algorithms [31, 80], [38, 158]. However, none of them is completely free of locks and atomic instructions and does dynamic load-balancing while using simple data structures at the same time. Most of the earlier algorithms either use locks (fine-grained or coarse-grained), or atomic instructions or complicated data structures to parallelize BFS. Some earlier endeavor of lockfree graph algorithms can be found in [54, 55], however, those algorithms use atomic instructions for correctness. Note that although lock or atomic instructions based resource protection is very common, it has many disadvantages including non-scalability and inefficiency. In [38], the authors proposed a lock and atomic-instruction free VIS data structure (bit array) to keep track of visited vertices in BFS. They used static load-balancing to divide the vertices, adjacency lists, and the VIS data structure, and because of static load-balancing, no lock was required. On the contrary, here we present shared-memory parallel BFS algorithms that use lock- and atomic instruction-free simple data structure with optimistic parallelization while doing dynamic load-balancing among threads at the same time.

Although different optimistic parallelization techniques have been used for efficient parallelization of several irregular problems, most of them use locks and/or atomic instructions for resolving conflicts. Cledat et al. [52] has proposed another type of optimistic parallelization in which one traces the data dependency and readability to decide whether two operations can be executed in parallel, and demonstrated its use on graph coloring problem. However, the type of optimistic parallelization technique we use for BFS differs from all of these. In our approach, threads update global shared data structures without any protection. However, they can detect inconsistencies on the fly and use the perceived values to explore a segment of vertices from a queue only when they find it to be safe. In case of inconsistencies, threads retry to get consistent values. We exploit problem-specific properties to maintain correctness of the algorithm.

In this work, we use a lock- and atomic instruction-free optimistic parallelization approach to implement two types of shared-memory parallel BFS algorithms based on centralized queue and distributed randomized work-stealing for dynamic load-balancing, and show that these algorithms outperform their lock-based counterparts. We present proof of correctness of these algorithms, analyze theoretical complexity and demonstrate that the proved theoretical bounds actually hold in practice using experimental results. We also demonstrate the performance and scalability of these algorithms on different modern multicore architectures including Intel Westmere, Sandy Bridge, AMD Magny-Cours as well as on Intel Xeon Phi Many-Integrated-Cores.

To the best of our knowledge, there is only one other shared-memory parallel BFS implementation [119] that performs dynamic load-balancing without using locks and atomic instructions, and our implementations outperform that one, too. However, implementation from [119] uses a complicated data structure (called a *bag*) instead of optimistic parallelization to achieve the goal, whereas we use simple array-based data structures with optimistic parallelization technique.

7.3 Prior Work

In this section, we summarize some of the prior work on shared-memory parallel BFS. Leiserson et al. [119] have implemented a work-efficient parallel BFS algorithm using `cilk++` which does not use locks and atomic instructions. However, it uses a specialized data structure called a *bag*, which is composed of reducers (a concurrent hyper-object provided by `cilk++`) [82] instead of standard FIFO queues. In [18], authors have proposed a hybrid of top-down (parent to child) and bottom-up (child to parent) exploration of edges during BFS, and used atomic instructions to ensure mutually exclusive writes. In [80], a NUMA (Non-uniform Memory Architecture) aware graph traversal technique has been proposed which uses a work-stealing approach and an idle thread steals from other neighboring threads running on the same socket to improve cache efficiency. This algorithm also uses atomic instructions for correctness. In [158], the authors have proposed a *block-accessed shared queue* data structure to implement a layered (level-synchronous) BFS and used atomic *fetch_and_add* to change the queue index pointer. Chhugani et al. [38] have proposed a lock and atomic-instruction free update of VIS data structure that keeps track of visited vertices during BFS. They have used static load-balancing to distribute the vertices, adjacency lists, and VIS data structure among the threads, and static work division typically does not require locks or atomic instructions. Lastly, in [105], the authors have proposed a hybrid approach which chooses an appropriate version of BFS algorithm from a) a serial version, b) two different multicore versions, and c) a GPU version, mainly based on the number of vertices in the current and the next BFS level. They have presented two level-synchronous parallel BFS algorithms for multicores which use a read-based method (random arrays instead of queues) and a read+queue based approach with and without using a visited array bitmap, respectively. Their work has adapted the queue based BFS algorithm from [9], which was claimed to be the best performing state-of-the-art parallel BFS at its time of publication. They have also used atomic instructions in their BFS for updating the visited vertices bitmap.

7.4 Our Contributions

None of the known parallel BFS algorithms based on dynamic load-balancing is free of locks (e.g. [11] uses `locks`), atomic instructions (e.g. [105] uses atomic `case_and_set`) and complicated data structures (e.g. [119] uses `bags of reducers`) at the same time as opposed to ours. Similarly, none of them has considered lock- and atomic instructions-free work-stealing for dynamic load-balancing which we use in our algorithms.

Our major contributions in this work are summarized below:

▷ **Novel algorithms using optimistic parallelization:** We present two types of level-synchronous parallel BFS algorithms for multicores along with their variants based on:

- *Centralized queues:* single and multiple queues
- *Distributed queues using* ***explicit*** *randomized work-stealing*: with/without special considerations for scalefree graphs (i.e., power law graphs).
- *Distributed queues using* ***implicit*** *randomized work-stealing provided by* `Cilk`*'s work-stealing scheduler*: uses a fork-join type recursive divide-and-conquer technique. This algorithm is also free of locks and atomic instructions.

We show how to use *optimistic parallelization to avoid the use of locks and atomic instructions* during dynamic load-balancing. We allow concurrent threads to update shared variables (queue indices and distance values) without locks and atomic instructions. However, we make sure that the results are still correct and the overhead of duplicate exploration (a consequence of unprotected update of queue indices) is negligibly small.

▷ **Theoretical analysis:** We prove correctness of the lockfree algorithms, and theoretical bounds of the corresponding lock-based algorithms.

▷ **Experimental analysis:** We implement these algorithms using Intel® `Cilk++`™ [118], `Cilk`™`Plus` [3] and `OpenMP` [59] and present performance and scalability results on different architectures. Major conclusions that we have reached are as follows:

- Lockfree versions are generally faster than the corresponding lock-based versions.
- Work-stealing based algorithms are more scalable than centralized queue based algorithms, and scale till 244 threads on the `Intel Xeon Phi` architecture even with hyper-threading, where the actual number of cores in the machine is 61 with 4 hardware threads running per core.
- For explicit work-stealing, the `OpenMP` implementation is slightly faster than its `Cilk++` and `Cilk Plus` counterpart. However, `OpenMP` implementation is slower than the divide and conquer based lock and atomic instructions free `Cilk Plus` BFS implementation that uses work-stealing provided by the `Cilk`'s runtime scheduler.

▷ **Comparison with prior results:** We compare the performance of our algorithms with the following two publicly available level-synchronous parallel BFS algorithms.

- Baseline1 [119]: To the best of our knowledge this is the only known state-of-the-art BFS algorithm that avoids the use of locks and atomic instructions during dynamic load-balancing; however, unlike our algorithms, it uses a complicated reducer-based recursive data structure (called a *bag*) provided by `Cilk++`. Furthermore, it does not use optimistic parallelization.
- Baseline2 [105]: This algorithm uses atomic instructions and performs better than the multicore algorithm presented in [9].

7.5 Our Parallel BFS Algorithms for Multicores

In this section, we present sketches of our parallel BFS algorithms. In all our algorithms we use two arrays of queues (these queues are basically randomly accessible arrays) $Q_{in}[0..p-1]$

and $Q_{out}[0..p-1]$ to store vertices in the current level and the next level (assuming there are p threads in the system) of BFS, respectively.

We start from a designated source vertex, s and put s in $Q_{in}[0]$, visit all neighbors, $\{v\}$ of s and put them in the next level queue $Q_{out}[i]$, where i is the id of the thread that discovered those neighbor vertices. Then we swap Q_{in} and Q_{out} and the next level of exploration starts. We keep exploring any unexplored vertices in Q_{in} this way until there is any unexplored vertex remaining in the queue. Vertices are explored in parallel by all threads and threads put newly discovered vertices in their private output queues, $Q_{out}[thead_id]$. No queue is protected by locks or atomic instructions. We use the term `segment` to denote a contiguous part of an input queue. Typically, a thread explores a segment of vertices from an input queue. We always **add a sentinel (0) at the end of each queue** which helps in ensuring correctness of the lockfree algorithms.

We use optimistic parallelization in our algorithms in the following way: we allow multiple threads to fetch or steal a `segment` from the shared distributed/centralized queues without any locks and atomic instructions, assuming that nothing will go wrong. However, because of the absence of locks and atomic instructions during the update of shared queue indices, a thread can pick either

- ▷ an *invalid* segment (i.e., at least one endpoint of the segment falls outside the actual queue range), or
- ▷ an *overlapping* segment (i.e., segment is valid but overlaps with other thread's current segment) or
- ▷ a *stale* segment (i.e., segment is valid but already explored by other threads).

While invalid segments may produce wrong results, or even cause the program to crash, overlapping or stale segments can only cause duplicate explorations. In our algorithms, threads check for invalid segments while stealing or fetching a segment from the queues, and in a case of failure (i.e., actually picked an invalid segment), they retry to get a valid segment. The fact that for BFS duplicate exploration does not hamper correctness helps us to use optimistic parallelization. Furthermore, several tricks can be used to eliminate duplicate explorations almost entirely. Note that because of duplicate exploration, some extra overhead may be added to the system. On the other hand, we are completely removing the overhead of locks[1] and atomic instructions which are known to cause serialization and create bottlenecks when the number of threads increases. So, here the challenge is to reduce the cost of inconsistency/conflict detection and duplicate exploration to such an extent that the total overhead does not negate the total savings resulting from the avoidance of locks, atomic instructions, and complicated data structures. Similar techniques can also be used for other types of algorithms where repeated work does not introduce inaccuracy in results (e.g., DFS, IDA*, A*, and applications that pick minimum or maximum from a bunch of values, etc).

To name our algorithms, we use the convention shown in Table 7.1. Table 7.2 shows the acronyms of the presented algorithms. We have explained the lockfree algorithms in the following section. Each algorithm has a corresponding lock-based version where threads use locks to change the shared variables (queue indices, segment pointers) instead of using optimistic parallelization.

[1]On a typical PC, locks are known to be more than 20 times slower than standard CPU operations [139].

Subscript	Meaning
C	Centralized
D	Decentralized
L	Lockfree
W	Work-stealing
S	Scalefree

TABLE 7.1: *Naming convention.*

Acronym	Full Name	Acronym	Full Name
BFS_C	Centralized (with locks)	BFS_{WS}	Work-stealing + Scalefree (with locks)
BFS_{CL}	Centralized + Lockfree	BFS_{WSL}	Work-stealing + Scalefree + Lockfree
BFS_{DL}	Decentralized + Lockfree	BFS_{WSLDQ}	Divide-and-conquer Based, Implicit Work-stealing
BFS_W	Work-stealing (with locks)	Baseline1	Implementations from [119]
BFS_{WL}	Work-stealing + Lockfree	Baseline2	Implementations from [105]

TABLE 7.2: *Program acronyms.*

7.5.1 Based on Centralized Queues

BFS_{CL} **(Centralized + Lockfree).** In this algorithm, we maintain a global queue pointer q with the invariant that all vertices in the queues to the left of $Q_{in}[q]$ have already been explored. Each queue, $Q_{in}[k]$, has a front pointer $Q_{in}[k].f$ initialized to 0, and we maintain the invariant that all vertices to the left of $Q_{in}[k].f$ in that queue have already been visited. Whenever a thread needs to fetch a segment, it first stores q in a local variable k. It then keeps incrementing k (if needed and as long as necessary) to find the leftmost queue with $f' < Q_{in}[k].r$ where r is $Q_{in}[k]$'s rear pointer and f' is a local variable that holds the value of $Q_{in}[k].f$. As soon as it finds such a k, it updates q to k, and $Q_{in}[k].f$ to $f' + s$ where s is the length of a segment. Observe that in the case of two or more threads changing q at the same time, a thread may end up updating q to a point to the left of where it should actually be, which can result in a subsequent thread receiving a segment with vertices that are already visited. The $Q_{in}[q].f$ pointer can also get updated backward in a similar way, which may cause two threads receiving the same segment for exploration.

However, as mentioned before, this type of duplicate exploration does not hamper the correctness of the algorithm. Nevertheless, to reduce the possibilities of duplicate exploration, we use the following trick: whenever a thread reads a new vertex from the queue for exploration, it empties that location by setting it to 0. Whenever a thread sees a 0 in the queue, it concludes that the current segment has already been explored or is under exploration by some other thread, or it has reached the end of the queue segment. So, it simply stops at that point and tries to get a new segment from the queues. Note that there is no possibility of creating a gap in the queues because a thread only stops when it sees a 0 (rather than stopping by checking a rear pointer) and a 0 can only appear either at the end of the queue or if the element has already been explored. While exploring the vertices, each thread puts the newly discovered vertices in their own private output queue $Q_{out}[i]$ where i denotes the thread id, and after finishing the exploration of all vertices from all queues in the current level, we swap Q_{in} and Q_{out}, and exploration starts again for the next level.

BFS_{DL} **(Decentralized + Lockfree).** This algorithm builds on BFS_{CL}, however, rather than having one centralized queue, we now have j centralized queues for some $j \in [1, p]$, where each centralized queue consists of either $\lceil p/j \rceil$ or $\lfloor p/j \rfloor$ queues from Q_{in}. Note that $j = 1$ means it is a purely centralized approach like BFS_{CL}, whereas $j = p$ means purely distributed. At the beginning of each BFS level, each thread picks a random centralized queue, and whenever the thread becomes idle, it fetches the next available segment from that centralized queue and explores vertices from that segment. However, if there is no more segment left in its chosen

centralized queue, it randomly tries at most $cj \log j$ times (where $c > 1$ is a constant) to get a new nonempty centralized queue, and if it succeeds in finding such a queue, it explores vertices from that in the same way as before. It can be proved using the *balls and bins model* [135] that w.h.p. it takes no more than $cj \log j$ tries to check each centralized queue at least once for work provided $c > 1$. This process continues until all queues become empty and then, the next level of BFS starts.

7.5.2 Based on Distributed Randomized Work-stealing

BFS_{WL} **(Work-stealing + Lockfree).** In this algorithm, we assume that at the beginning of a BFS level, $Q_{in}[q].r$ holds the rear pointer of $Q_{in}[q]$ for every q, and this variable remains unchanged throughout that level. Every thread maintains three variables, namely q, f and r to keep track of the queue id, front pointer, and rear pointer, respectively, of the segment of vertices it is currently working on. Initially, thread $t \in [0, p)$ gets the entire $Q_{in}[t]$ as a single segment. As it explores the segment, it keeps updating its own f pointer accordingly. In order to reduce the chances of duplicate exploration of the same vertex by other threads, a thread clears (i.e., sets to 0) every location of the queue segment as soon as it reads the vertex stored in that location for exploration. A thread aborts working on a segment as soon as it encounters a 0 value (i.e., a cleared value) in the segment. Whenever the thread runs out of work, it chooses a random thread with enough work and tries to steal half of its work (i.e., the right half of its unexplored segment of vertices). The thief first saves the queue id q, front pointer f and rear pointer r of the victim's segment to local variables q', f' and r', respectively. It then performs the following sanity check: $f' < r' \leq Q_{in}[q'].r$. If the check fails (means the victim has possibly moved to another queue, and the retrieved segment is invalid), the thief aborts this steal and tries another random victim. Otherwise it updates its own q, f and r pointers to q', $f' + \frac{1}{2}(r' - f')$ and r', respectively, and the victim's r pointer to $f' + \frac{1}{2}(r' - f')$. It does not change the victim's q and f pointers. Observe that as no thread checks its own rear pointer while exploring, any invalid change to the rear pointer of the segment (which may happen due to not using locks and atomic instructions) does not hamper correctness. If a thief changes a rear segment pointer of the victim to any invalid location, no other thread will be able to steal from that particular victim for some time until either the victim itself becomes a thief and changes its own rear pointer or exploration of a new level starts. On the contrary, if a thief gets an invalid segment from a victim, using the sanity checks as described above, it safely avoids that segment and retries for a valid segment.

BFS_{WSL} **(Work-stealing + Scalefree + Lockfree).** The BFS_{WSL} algorithm is optimized for scalefree graphs (whose degree distribution follow power law) that appear in real-world very frequently (e.g., many biological interaction networks, social networks, Wikipedia and so on). This algorithm uses an approach similar to that used in BFS_{WL}. However, the vertices of each level are explored in two phases. In the first phase, the threads only explore the low-degree vertices using explicit work-stealing as before and push the higher degree vertices into a separate queue, Q_s (the definition of high degree can be changed using a threshold variable). At the end of this phase, we divide the adjacency list of each vertex from Q_s into p chunks and for each $i \in [1, p]$, the ith thread explores the ith chunk of the adjacency list of that vertex (phase 2). No work-stealing happens in this phase. We have also experimented with another variant of BFS_{WSL} which uses work stealing also in the second phase, and a thread is allowed to steal

half of the remaining unexplored adjacency list of a vertex if there is only one vertex left in the queue. However, this approach does not perform as well as the first approach.

BFS_{WSLDQ}: BFS$_{WSL}$ using `Cilk`'s Work-Stealing Scheduler. We change our BFS_{WSL} algorithm to use parallel recursive divide and conquer so that we can use the benefit of `Cilk's` work-stealing scheduler and avoid doing work stealing explicitly. Again, we do not use locks, atomic instructions, and complicated data structures. Each thread starts with its own queue as before. But before going for actual exploration, it recursively divides the work into two halves and spawns two threads to work on those two parts. Each thread keeps doing this until it reaches a `basecase` size. After reaching the pre-specified basecase size, it explores the vertices in two phases as done in BFS_{WSL}. Note that this algorithm is similar to the algorithm presented in [119] except that we use simple arrays instead of the bags (a reducer hyper-object). It is possible to avoid the use of bags because each thread can use its thread *id* returned by `Cilk's` runtime system to identify its own output queue to write to [2], and thus avoiding any potential race on the output queue. Each thread separates the high degree vertices and explores them in the second phase by dividing their adjacency lists evenly among the threads. Note that by using divide and conquer and spawning on disjoint subproblems, threads put big chunks of work in the `Cilk's` double-ended-queues (deque). `Cilk's` runtime work-stealing scheduler allows idle threads to steal those work from the deque automatically. So in this case, the programmer does not need to implement work stealing; the work-stealing scheduler of `Cilk` will handle the stealing instead.

7.6 Extension to NUMA

It is not difficult to optimize our algorithms for NUMA (Non-Uniform-Memory-Architecture) machines. For example, for the decentralized algorithm (BFS_{DL}), we can make sure that all threads that are initially assigned to the same centralized queue are launched on the cores of the same socket.[3] When a group of threads finishes exploring the vertices from their centralized queue pool, each of them can migrate to another random queue allocated on the same socket or in a case of no such queue available on the same socket, they explore from queues allocated on other sockets. This can also be done by assigning higher priorities to centralized queues allocated on the same socket and lower priorities to others. For the work-stealing based algorithms, we can use the following approach: while stealing, a thread randomly chooses a thread running on the same socket with higher priority. A NUMA-aware work-stealing approach for the betweenness centrality problem has been used in [80] which can also be adopted.

7.7 Discussion: Further Improvements

In parallel BFS algorithms, if one does not maintain any visited bitmap to keep track of visited vertices, it is possible to explore the same vertex multiple times by multiple threads. However, as explained before, this does not hamper correctness of the algorithm. Note that none of our algorithms has used any technique to remove duplicate vertices from the queues. One can use locks, try_locks or atomic instructions and/or bitmaps of visited vertices as used in [105] to

[2]The use of thread id makes this algorithm processor-aware, whereas [119] is processor-oblivious

[3]`Cilk++` does not allow setting thread affinities, and so, we can use OpenMP instead.

remove duplicate vertices (or to prevent duplicate exploration by different threads) from queues. It is also possible to depend on arbitrary concurrent write property to record only one parent of a vertex (since a vertex can have multiple parents) as used in [23]. However, we plan to use the following method to reduce duplicate exploration of vertices even further. Each thread will store the `queue id` (or parent id) of a vertex in a global array during discovery (using arbitrary concurrent write), and it will also check the `queue id` (or parent id) before exploring a vertex; if that matches with the previously stored value (which means that this thread discovered the vertex in the previous level), it explores the vertex, otherwise it skips that vertex. Note that this approach does not require any locking or atomic instructions. Avoiding duplicate explorations can be beneficial for dense and low diameter graphs where the number of duplicate vertices can be huge. Lock and atomic-instruction free approach for tracking visited vertices have already been proposed in [38] which can also be followed. It is also not difficult to incorporate the direction-optimized-BFS [18] in our algorithm to get an additional performance boost as reported by others.

7.8 Correctness

In this section, we argue the correctness of BFS_{WL}. The correctness proofs of other lockfree variants (BFS_{CL}, BFS_{DL}, and BFS_{WSL}) are based on similar arguments, and hence have been omitted. We argue that in each BFS level:

1. The vertex in each non-empty location of the input queues is explored by at least one thread (*safety property*), and
2. At any time step, if there are unexplored queue locations in the system, at least one thread is exploring a queue location it has not explored before (*progress property*).

Recall that in BFS_{WL} whenever a thread picks a vertex from a queue for exploration, it sets that queue position to 0 (zero). We call each of these zeros a *wall*. Each queue starts with a sentinel zero, that is, initially, there is a wall at the end of each queue. Then new walls start to appear at various locations of the queues as vertices are picked for exploration by threads. In each of our lockfree algorithms once a thread starts exploring a queue from any location (and always moving to the right), it does not stop until it hits a wall. Once a thread hits a wall, it tries to steal a new queue segment from a random busy thread (if any).

We say that an unexplored queue location j of any given queue q (i.e., $q[j] \neq 0$) is visible to a given thread τ provided τ is exploring (or about to start exploring) the same queue q, the next location of q to be accessed by τ is $q[i]$ for some $i \leq j$, and no wall exists between $q[i]$ and $q[j]$, i.e., $q[k] \neq 0$ for $i \leq k \leq j$.

It is not difficult to see that in every BFS level BFS_{WL} maintains the following invariant.

Invariant 7.8.1. At any given time, every unexplored input queue location is visible to at least one thread.

Since at the start of a given level each thread points to the first (leftmost) location of its own input queue, and each queue terminates at a wall, the invariant holds initially. It is also easy to see that every steal maintains the invariant. Since once a thread starts exploring a queue

segment it does not stop until it hits a wall, and attempt to steal only if it runs out of visible vertices, the invariant implies that each queue entry will be explored by at least one thread.

Invariant 7.8.1 also implies that at any given time if there are unexplored queue locations in the system, at least one thread is doing useful work, i.e., exploring a queue location it did not explore before. This property ensures that progress is always made by the algorithm.

Note that our correctness proof works under the sequential consistency model [4]. However, we believe that modifying our algorithms and/or proofs for some of the weaker models should not be too difficult. For example, our algorithms work under a relaxed model in which a core can read the value of its own previous write before the write is made visible to other cores. As all the races are benign, the correctness of our BFS algorithm should hold on any weaker consistency models as well.

7.9 Complexity Analyses

In this section, we analyze the parallel running times of the locked versions of our algorithms on p processing cores assuming that the input graph G has n vertices, m edges, maximum degree Δ and diameter D. We show the analysis of lock-based version instead of lockfree versions, because it is difficult to guarantee any bound on the lockfree algorithms. However, we experimentally show that the lockfree algorithms follow the bounds proved for the lock-based algorithms.

Each algorithm starts by setting the BFS level of each vertex to $+\infty$ except that of the source vertex whose distance is set to 0. For ease of analysis (as in [119]) we assume that when a thread is exploring the adjacency list of a vertex u, for every $v \in adj[u]$ it tries to lock v using a waitfree try_lock(v), and if it succeeds in doing so, it inserts v into its output queue provided the BFS level of v is still $+\infty$. The thread then updates v's BFS level to the correct value and releases the lock on v. This try_lock() approach ensures that no vertex appears more than once in the input queues (of next level) without incurring the contention overhead of a standard lock. In our actual implementations we do not lock v before checking it for insertion into the queue, and as in [119], we found that even without locks, duplicate insertions happen very rarely in practice.

7.9.1 BFS_C (Centralize + Lock)

At any given BFS level l, each thread either explores vertices or waits on a lock to acquire the next segment from the centralized queue. A thread waits $O(p)$ time for each access to the queue. Let us assume that at level l, there are q_l vertices in the input queues, and q'_l vertices are inserted into the output queues. Therefore, $q'_l = q_{(l+1)}$. We divide the q_l vertices into segments of size s, then the function to get the next segment from the centralized queue will be called $\left(\frac{q_l}{s} + p\right)$ times in total since each thread tries one additional time to be sure that the queue is empty. Therefore, the total wait time and cost of getting the next segment will be $O\left(\left(\frac{q_l}{s} + p\right)p\right)$. Also, observe that a level cannot end until each thread has completed exploring the vertices in its last segment even if there is no more work left in the centralized queue. Overall, the total

[4]The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

amount of work for a given level l can be computed by summing the work for exploration, for acquiring the next segment including the waiting for lock and for termination as follows: $O\left((q_l + q_l') + \left(\frac{q_l}{s} + p\right)p + ps\Delta\right)$.

Let w_l be the amount of work done by any thread at level l, and therefore, total work by all p threads at level l is $w_l p$. Hence, we get $w_l p = \Theta\left((q_l + q_l') + \left(\frac{q_l}{s} + p + s\Delta\right)p\right)$ $\Rightarrow w_l{=}\Theta\left(\frac{1}{p}q_l + \frac{1}{p}q_l' + \frac{q_l}{s} + p + s\Delta\right)$. Therefore, total work by all p threads across all BFS levels is $W{=}p\sum_{l=0}^{D} w_l$, and the parallel running time, $T_p{=}\ W/p{=}\Theta\left(\sum_{l=0}^{D}\left(\frac{1}{p}q_l + \frac{1}{p}q_l' + \frac{q_l}{s} + p + s\Delta\right)\right)$. Note that $\sum_{l=0}^{D}(q_l + q_l'){=}O(n+m)$. We consider three different cases based on the relative sizes of q_l, p and Δ, and choose the value of s accordingly. For each case $i \in [1, 3]$, we compute the parallel running time ${T_p}^{(i)}$ of the algorithm assuming that only case i arises in each BFS level. Then ${T_p}^{(1)} + {T_p}^{(2)} + {T_p}^{(3)}$ will be an upper bound on the overall parallel running time T_p of the algorithm.

Case 1 ($1 \le q_l < \Delta p$): We set $s = 1$, and obtain ${T_p}^{(1)}{=}O\left(\frac{m+n}{p} + D\left(q_l + p + \Delta\right)\right) {=}O\left(\frac{m+n}{p} + D\Delta p\right)$.

Case 2 $\left(\Delta p \le q_l < \Delta p^2\right)$: We set $s{=}\left\lceil \frac{q_l}{\Delta p} \right\rceil$, and get ${T_p}^{(2)}{=}O\left(\frac{m+n}{p} + D\left(\Delta p + p\right) + \sum_{l=0}^{D}\frac{q_l}{p}\right) = O\left(\frac{m+n}{p} + D\Delta p\right)$.

Case 3 $\left(q_l \ge \Delta p^2\right)$: We choose $s = \left\lceil\sqrt{\frac{q_l}{\Delta}}\right\rceil$, leading to ${T_p}^{(3)}{=}O\left(\frac{m+n}{p} + \sum_{l=0}^{D}\left(\sqrt{q_l\Delta} + p\right)\right) = O\left(\frac{m+n}{p} + \sum_{l=0}^{D}\left(\frac{q_l}{p} + p\right)\right) = O\left(\frac{m+n}{p} + Dp\right)$.

Therefore, $T_p \le {T_p}^{(1)} + {T_p}^{(2)} + {T_p}^{(3)} = O\left(\frac{m+n}{p} + D\Delta p\right)$. Observe that the algorithm is work-efficient, i.e., $pT_p = O(m + n)$ provided $(m + n) = \Omega(D\Delta p^2)$.

Theorem 7.1. *For a graph of diameter D with n vertices, m edges and maximum degree Δ, BFS_C takes $O\left(\frac{m+n}{p} + D\Delta p\right)$ time when run on p cores, and it is work-efficient provided $(m + n) = \Omega(D\Delta p^2)$.*

7.9.2 BFS_W (Work-stealing + Lock)

Let Min-Steal-Size denote the number of vertices in a queue required to allow a steal, and Max-Steal-Attempts denote the maximum number of steal attempts made by a thread before exiting a level. Let us consider a specific BFS level l. Suppose there are q_l vertices in the input queues, and let ${q_l}'$ be the number of vertices to be inserted into the output queues. Therefore, $q_l' = q_{(l+1)}$. We use the following result from probability theory: for any $\alpha > 1$, if $\alpha p \log p$ balls are thrown uniformly at random into p bins, then with probability at least $1 - \frac{1}{p^{\alpha-1}}$ each bin will contain at least one ball. We model the processors as bins, and the steal attempts as balls. We will first assume that q_l and Min-Steal-Size are independent of p. Therefore, we can treat them as constants w.r.t. p. We will remove this assumption later.

We consider the steal attempts in the entire system sorted in nondecreasing order of time, and group them into rounds with each round containing $\alpha p \log p$ consecutive steal attempts in the sorted order. Hence, after each round of steal attempts each thread will be the victim of at least one steal attempt with probability at least $1 - \frac{1}{p^{\alpha-1}}$. Since each successful steal attempt splits the work of the victim thread into two halves, after the first round of steals, no thread will contain

more than $\frac{q_l}{2}$ entries in it's input queue with probability at least $1 - \frac{1}{p^{\alpha-1}}$. In general, after $k = \log\left(\frac{q_l}{\text{Min-Steal-Size}}\right)$ rounds of steal attempts all input queues will have $\leq q_l/2^k =$ Min-Steal-Size entries with probability at least $1 - \frac{k}{p^{\alpha-1}}$. Since we assumed k to be a constant w.r.t. p, this means that w.h.p. in p, after $k(\alpha p \log p) = \Theta\left(p \log p \log\left(\frac{q_l}{\text{Min-Steal-Size}}\right)\right)$ steal attempts in the entire system, no steal attempt will succeed.

Now if we do not assume q_l and Min-Steal-Size to be constants (w.r.t. p), we must choose α carefully, so that $1 - \frac{k}{p^{\alpha-1}}$ remains a high probability in p. For example, we may choose to keep $1 - \frac{k}{p^{\alpha-1}} \geq 1 - \frac{1}{p}$ which leads to $\alpha \geq 2 + \frac{\log k}{\log p} = 2 + \frac{\log\log\left(\frac{q_l}{\text{Min-Steal-Size}}\right)}{\log p}$. In that case, w.h.p. in p, after $k(\alpha p \log p) = \Theta\left(p\left(\log p + \log\log\left(\frac{q_l}{\text{Min-Steal-Size}}\right)\right)\log\left(\frac{q_l}{\text{Min-Steal-Size}}\right)\right)$ steal attempts in the entire system, all steal attempts will fail.

Observe that we can compute the total time W_l spent by all threads in a given level by summing up the followings: a) the total time spent by all threads waiting to be launched, b) total time spent doing real work (i.e., exploring vertices), c) total time spent trying to steal, and d) total wait time at a ***sync*** point. For our implementation, the total time spent waiting to be launched by all threads is $\mathcal{O}\left(\sum_{i=1}^{p} i\right) = \mathcal{O}\left(p^2\right)$. Assuming that no duplicate exploration happened, total time spent doing useful work is clearly $\mathcal{O}\left(q_l + q_l'\right)$.

Recall that we divided the steal attempts into rounds of $\alpha p \log p$ steal attempts each, where $\alpha > 1$ is a parameter to be determined later to ensure that the final parallel running time holds w.h.p. in p. After $\log q_l$ rounds of steal attempts w.h.p. in p no thread will have any work worth stealing (assuming Min-Steal-Size $= 1$). Let's also assume for simplicity that Max-Steal-Attempts $= 3\alpha p \log p$. Thus the total number of steal attempts in level l will be $3\left(\log q_l + p\right)\alpha p \log p$ w.h.p. in p. The total time spent at the ***sync*** point can be determined by observing that in addition to the $\mathcal{O}\left(\log p\right)$ idle time introduced by the ***sync*** implementation itself, w.h.p. in p a thread needs to wait for $\mathcal{O}\left((\text{Min-Steal-Size})\Delta\right)$ time so that other working threads (if any) can complete exploring their last segments of size Min-Steal-Size. Hence, $W_l = O(p^2 + (q_l + q_l') + 3\left(\log q_l + p\right)\alpha p \log p + (\log p + \Delta)p)$ w.h.p. in p. Thus parallel time spent in this level is $\frac{W_l}{p} = O\left(\frac{q_l + q_l'}{p} + 3\left(\log q_l + p\right)\alpha \log p + \Delta\right) = \mathcal{O}\left(\frac{q_l + q_l'}{p} + \left(\log n + p\right)\alpha \log p + \Delta\right)$ w.h.p. in p, where we use the observation that $q_l \leq n$.

Therefore, $T_p = \sum_{i=1}^{D} \frac{W_l}{p} = O\left(\sum_{i=1}^{D}\left(\frac{q_l + q_l'}{p}\right) + D(\log n + p)\alpha \log p + D\Delta\right)$. Observing that $\sum_{i=1}^{D}\left(q_l + q_l'\right) = \mathcal{O}\left(m + n\right)$, we have, $T_p = \mathcal{O}\left(\frac{m+n}{p} + D(\log n + p)\alpha \log p + D\Delta\right)$.

Now we are in a position to determine the value of α. Let us first assume that m, n, D, Δ and the q_l's and q_l''s in all levels are independent of p (which is indeed the case), and thus they are constants w.r.t. p. Then we can simply set $\alpha = 2$, and the resulting $T_p = \mathcal{O}\left(\frac{m+n}{p} + D(\log n + p)\log p + D\Delta\right)$ will still hold w.h.p. in p. The algorithm is work-efficient provided $pT_p = \mathcal{O}\left(m + n\right) \Rightarrow m + n = \Omega\left(D\Delta' p\right)$, where, $\Delta' = \Delta + (p + \log n)\log p$.

If we drop the independence assumption from the last paragraph, a constant value of α will no longer guarantee that T_p holds w.h.p. in p. We find a value of α below that provides such a guarantee. A round of $\alpha p \log p$ steal attempts is successful (i.e., hits each thread at least once) with probability at least $1 - \frac{1}{p^{\alpha-1}}$, and if there are k such rounds, all of them are successful with probability $\geq 1 - \frac{k}{p^{\alpha-1}}$. The total number of such stealing rounds across all levels is $\sum_{i=1}^{D}\left(\log q_l + p\right) \leq D(p + \log n) \leq Dp \log n$. In order to make sure that the bound on T_p holds w.h.p. in p, we enforce $1 - \frac{Dp \log n}{p^{\alpha-1}} \geq 1 - \frac{1}{p} \Rightarrow \alpha \geq 3 + \frac{\log D + \log\log n}{\log p}$. Now putting

$\alpha = 3 + \frac{\log D + \log\log n}{\log p}$ in T_p, we obtain $T_p = O(\frac{m+n}{p} + D(p + \log n)(\log p + \log D + \log\log n) + D\Delta)$ w.h.p. in p. Note that both $\log p$ and $\log D$ is bounded by $\log n$. Therefore, our T_p boils down to $\mathcal{O}\left(\frac{m+n}{p} + D(p + \log n)(\log n) + D\Delta\right)$. The algorithm is work-efficient provided $pT_p = \mathcal{O}(m+n) \Rightarrow m + n = \Omega(D\Delta'' p)$, where, $\Delta'' = \Delta + (p + \log n)(\log n)$.

Theorem 7.2. *For a graph of diameter D with n vertices, m edges and maximum degree Δ, BFS_W takes $T_p = \mathcal{O}\left(\frac{m+n}{p} + D(p + \log n)(\log n) + D\Delta\right)$ time (w.h.p.) when run on p cores, and it is work-efficient provided, $pT_p = \mathcal{O}(m+n) \Rightarrow m+n = \Omega(D\Delta'' p)$, where, $\Delta'' = \Delta + (p + \log n)$ $(\log n)$.*

7.9.3 BFS_{WS}(Work-stealing + Scalefree + Lock)

We consider the degree of a vertex *high* provided its degree is at least $p\log p$. All high degree vertices generated in each BFS level are processed separately one by one by dividing the adjacency list of each such vertex evenly among all threads. Thus the $D\Delta$ term in the parallel time complexity of BFS_W (see Theorem 7.2) is replaced with $Dp\log p$ (i.e., Δ effectively drops to $p\log p$), and a new term $\frac{m}{p}$ is added which arises from the parallel processing of the adjacency lists of high degree vertices. Thus we have $T_p = \mathcal{O}\left(\frac{m+n}{p} + D(p + \log n)(\log n) + Dp\log p\right) = \mathcal{O}\left(\frac{m+n}{p} + D(p + \log n)(\log n)\right)$ since $Dp\log p$ is bounded by $Dp\log n$.

Theorem 7.3. *For a graph of diameter D with n vertices, m edges and maximum degree Δ, BFS_{WS} takes $\mathcal{O}\left(\frac{m+n}{p} + D(p + \log n)(\log n)\right)$ time (w.h.p.) when run on p cores, and it is work-efficient provided $pT_p = \mathcal{O}(m+n) \Rightarrow m + n = \Omega(D\Delta'')$, where, $\Delta'' = (p + \log n)$ $(\log n)$.*

7.10 Experimental Results

In this section, we analyze the performance of our BFS implementations and match the results with our proved theoretical performance bounds. We compare performance of our algorithms with Baseline1 and Baseline2 and demonstrate the scalability of our work-stealing BFS algorithms on different machine architectures including the new Xeon Phi architecture.

7.10.1 Simulation Environment and Input Graphs

Attribute	Lonestar	Stampede	Trestles
Processors	Two 3.33 GHz-Hexa-Core 64-bit Intel-Westmere	Two 2.7 GHz E5-2680 Intel Xeon (Sandy Bridge), one Intel Xeon Phi Co-processor	Four 8-core 2.4 GHz AMD Magny-Cours processor
Cores/node	12 (2×6)	16 (2×8)	32 (4×8)
RAM	24 GB, 177GB/s	32 GB	64 GB, 171GB/s
OS	Linux Centos 5.5	Linux Centos	Linux Centos 5.5
Cache	12MB shared $L3$, 256KB private $L2$, 64KB private $L1$	20MB shared $L3$, 256KB private $L2$, 64KB private $L1$	12MB shared $L3$, 512KB private $L2$, 128KB private $L1$

TABLE 7.3: *Simulation environment.*

To conduct our experiments, we have used machines from the Lonestar 4 and Stampede [5] supercomputing clusters located at the Texas Advanced Computing Center (TACC) [6], and

the Trestles cluster at San Diego Supercomputer Center (SDSC). We have tested our parallel BFS algorithms on real-world graphs such as *cage 15*, *cage 14*, *kkt-power*, *freescale*, *Wikipedia-2007* and *inline_1* collected from the Florida Sparse Matrix Collection [61]. We have also used synthetic random RMAT graphs generated using the Graph-500 RMAT generator[5] with millions of vertices and up to a billion of edges. All graphs were directed. Properties of the simulation environment and the input graphs are summarized in Tables 7.3 and 7.4, respectively.

Graph	Description	n	m	Diameter
Cage15	DNA electrophoresis, 15 monomers in polymer	5.2M	99.2M	53
Cage14	DNA electrophoresis, 14 monomers in polymer	15.1M	27.1M	42
Freescale	Large circuit, Freescale Semiconductor	3.4M	18.9M	141
Wikipedia	Gleich/Wikipedia-20070206	3.6M	45M	14
kkt-power	Optimal power flow, nonlinear optimization (KKT)	2M	8.1M	11
inline_1	Stiffness Matrix	3.5M	45M	222
RMAT100M	RMAT Graph generated using Graph-500 RMAT generator	10M	100M	12
RMAT1B	RMAT Graph generated using Graph-500 RMAT generator	10M	1B	5

Table 7.4: *Graphs and their properties. In this table, n and m denote the number of vertices and the number of edges of the graph, respectively. The diameters in the table show the maximum diameters explored by the BFS rather than the actual diameters of these graphs.*

7.10.2 Performance Analysis

In this section we show the performance comparison of all BFS implementations. We compare all

Graph	Baseline1	Baseline2			Centralized			Work-stealing		Serial BFS
		Read	Read + bitmap	Local queue +read + bitmap	BFS_C	BFS_{CL}	BFS_{DL}	BFS_{WS}	BFS_{WSL}	
Kkt-power	0.3	7.6	9.5	6.0	0.3	0.3	0.3	0.4	0.4	1.8
Freescale	37.6	189.6	221.6	105.0	38.4	36.9	33.9	44.0	42.6	241.8
Cage14	32.2	62.6	92.9	83.3	31.4	30.7	30.5	37.5	37.0	227.1
Wikipedia	83.8	180.2	321.5	259.8	77.9	73.0	76.5	78.0	73.4	518.5
Cage15	131.0	189.5	285.4	263.2	123.3	119.6	121.9	141.2	141.0	911.1
Graph-10M-100M	329.7	324.2	391.2	320.4	312.6	306.3	310.0	354.0	344.8	2576.0
Graph-10M-1B	1900.3	1755.4	1365.1	1143.2	1843.5	1844.6	1849.8	1975.0	1983.6	12278.6

(a) Running times (ms) on Lonestar (Intel Xeon with 12 cores).

Graph	Baseline1	Baseline2			Centralized			Work-stealing		Serial BFS
		Read	Read + bitmap	Local queue +read + bitmap	BFS_C	BFS_{CL}	BFS_{DL}	BFS_{WS}	BFS_{WSL}	
Kkt-power	0.8	7.5	11.8	8.6	0.6	1.0	1.0	1.2	0.8	5.3
Freescale	81.2	219.2	226.5	198.9	185.1	144.2	133.7	79.0	66.8	648.8
Cage14	64.0	78.3	88.0	91.5	107.2	93.2	58.9	55.0	53.9	633.5
Wikipedia	167.2	242.8	336.6	279.5	162.3	156.4	146.2	108.3	106.4	1843.2
Cage15	208.2	221.7	243.7	242.2	301.9	263.8	206.9	179.0	173.9	2432.0
Graph-10M-100M	410.6	366.4	385.2	314.9	394.7	401.0	358.0	370.5	359.1	7951.5
Graph-10M-1B	2313.5	1996.6	1158.3	1029.3	1869.3	1864.1	1752.6	1854.3	1832.6	32443.6

(b) Running times (ms) on Trestles (AMD Magny with 32 cores).

Table 7.5: *Running times of different algorithms (all times are shown in milliseconds). All programs were implemented using* `Cilk++`*. The minimum running time in each row is highlighted by color. If our algorithms do not achieve the minimum running time, we have highlighted the minimum time for our algorithms in addition to the absolute minimum time in a row.*

variants of our parallel BFS algorithms[6] mentioned in Section 7.5 with the BFS implementations of Baseline1 [119] and Baseline2 [105]. The algorithm presented in [105] has both CPU and

[5]parameters used: a=.45, b=.15 and c=.15.
[6]The decentralized algorithm was ran with 1 centralized queue.

GPU implementations, but we compare only with the CPU implementations. Moreover, the CPU implementation has several variants, and we compare with all of them. In [105] and [119], the authors have used `pthreads` and `Cilk++` concurrency platforms for parallelism, respectively. All programs (including codes from [119] and [105]) are compiled using the `-O3` optimization parameter. We run all programs with 1000 random non-zero degree source vertices and compute the average running time per source for each. The results show that our work-stealing algorithms optimized for scalefree graphs almost always perform better than the corresponding unoptimized variant even on general graphs. Hence, we do not report the results for those unoptimized work-stealing variants.

Running Time. Tables 7.5(a) and 7.5(b) show the running times of different algorithms run on a single compute node of Lonestar (Intel Westmere) and Trestles (AMD Magny-Cours), respectively. The shaded cells mark the smallest running times in each row. If our algorithms do not achieve the minimum running time, we have highlighted the minimum time for our algorithms in addition to the absolute minimum time in a row. Observe that the *lockfree versions generally run faster than the corresponding lock-based versions.* Also observe that on Lonestar (12-cores), BFS_{CL} generally outperforms both Baseline1 and Baseline2, while on Trestles (32-cores) BFS_{WSL} does. We explain why that should be the case in the following paragraphs.

Centralized vs. Distributed: Performance Benefits of Work-stealing. Table 7.5 shows that the centralized queue based BFS implementations perform better than the work-stealing based approaches on Lonestar (12-cores/node), whereas on Trestles (32-cores/node) work-stealing BFS algorithms show better performance. This change in the best-performing

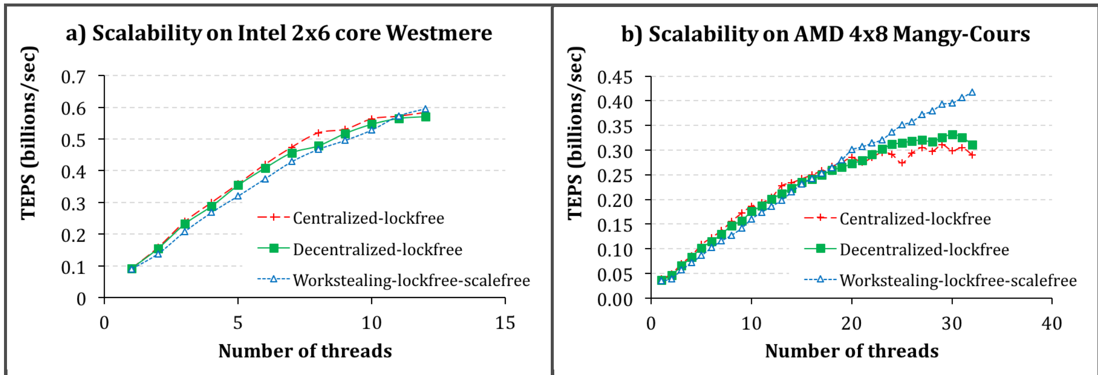


FIGURE 7.1: *Scalability of lockfree parallel BFS algorithms running on (a) Lonestar and (b) Trestles. All algorithms were run on the Wikipedia graph. All programs were implemented using Intel® `Cilk++`™.*

approach with the increase of the number of cores has been also demonstrated in Figures 7.1(a) and 7.1(b) by the strong scalability of different variants of our BFS algorithms on the scale-free Wikipedia graph when run on Lonestar and Trestles, respectively. It is apparent that the centralized queue based versions are not scalable beyond 20 cores while the work-stealing version remains scalable till the end (i.e., up to 32 cores). This observation matches with what the theory predicts (see Theorems 7.1, 7.2 and 7.3 in Section 7.9). In the work-stealing implementation, steal attempts are more or less equally distributed among all queues, and thus the maximum number of simultaneous accesses to each queue in that implementation is fewer than that to the single shared queue pool in the centralized queue based implementation. In our centralized queue based algorithms we tried to reduce the number of accesses to the centralized queue by controlling (i.e., increasing) the segment size s. However as s increases, the amount

of time threads must wait idly until the last working thread completes execution in any BFS level also increases (please see the proof of Theorem 7.1). This gives rise to the $D\Delta p$ term in the parallel time complexity of BFS_C while the largest p term for BFS_W and BFS_{WS} is only $Dp(\log p + \log D + \log\log n)$ (please see the proof of Theorem 7.2). Hence, with the increase of p, the performance of the centralized versions degrades faster than the distributed work-stealing versions. In addition to that, the fact that the work-stealing BFS_{WS} version is optimized for scalefree graphs may have also contributed to its scalability. Observe that unlike BFS_C and BFS_W, no Δ term appears in the complexity of BFS_{WS}.

As mentioned before, in lockfree versions, although we are removing the overhead related to locks and atomic instructions, it causes more frequent accesses to the shared queues and this frequency increases with the number of threads. As the number of threads in the system increases, there will be more simultaneous accesses to the same centralized queue causing more threads to grab duplicate segments, which in turn will result in more duplicate work, and thus, incur more overhead. In the work-stealing algorithms, different threads access different queues for input and steal from random victims. This distributes the simultaneous accesses to the shared queues somewhat evenly among all queues and causes fewer duplicate/overlapping segments and less associated overhead compared to the centralized queue versions.

Comparison with Baseline1 and Baseline2. Figure 7.2 shows performance comparison of our algorithms with that of Baseline1 and Baseline2. For all real-world graphs in our experiments,

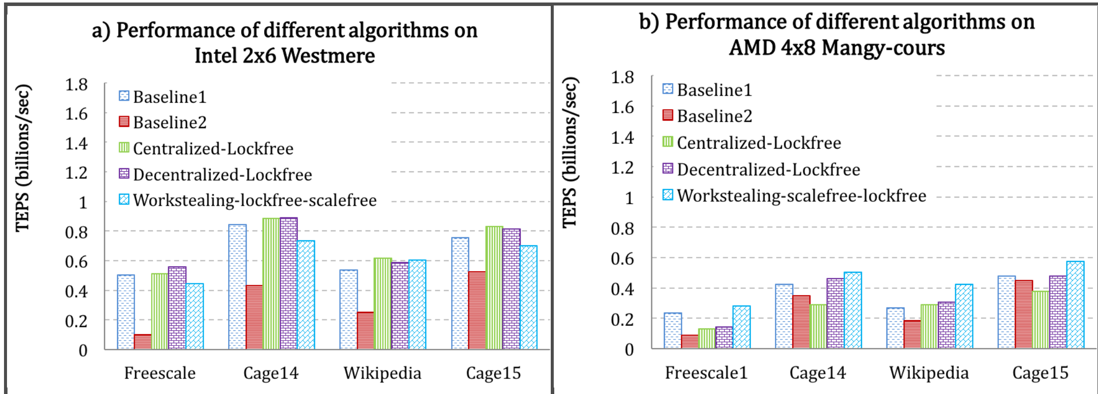


FIGURE 7.2: *Performance in terms of* Traversed Edges Per Second *(TEPS) when traversing real-word graphs on machines from (a) Lonestar (12 cores) and (b) Trestles (32 cores). All programs were implemented using Intel® `Cilk++™`.*

our best-performing BFS implementation is better than both the implementations from [119] and [105]. Our algorithms perform the best for scalefree graphs and sparse graphs. However, for large synthetic RMAT graphs (`graph-10M-100M` and `graph-10M-1B`) Baseline2 performs slightly better that both Baseline1 and ours. One possible reason for this is that it uses a bitmap to track visited vertices (using atomic `case_&_set` instruction), and thus avoids duplicate exploration. Note that in our algorithms although the same vertex can appear only once in a particular thread's output queue, it can appear in the output queue of multiple threads, if it had been discovered by those threads exactly at the same time. With 10M vertices and 1B edges, the `graph-10M-1B` being very dense resulted in a lot of duplicate explorations during the execution of our algorithms. On the contrary, Baseline2 runs faster by avoiding the bulk overhead of these unnecessary explorations.

7.10.3 Why Lockfree Does Better Load-balancing than Lock-based

Our experimental results show that the lockfree work-stealing ends up doing better load-balancing than the lock-based work-stealing algorithm for BFS. Here we explain why that happens based on some statistical data. Table 7.6 shows some statistics on the steal attempts made by threads in BFS_{WS} and BFS_{WSL}. Both implementations were run 5 times with 100 sources of the Wikipedia Graph on Intel® Westmere (12 cores), and the average values were computed for the number of successful steal attempts and for different types of failed steal attempts. Table 7.6 shows that, though the total number of steal attempts is slightly more in BFS_{WSL} than in BFS_{WS}, the percentage of successful steal attempts is also higher in BFS_{WSL}. Similarly, the number of failed steal attempts as a result of a victim being idle is lower in BFS_{WSL}. Recall that a thread becomes idle when it runs out of work and gives up searching for work after a certain (say, MAX-STEAL-ATTEMPTS) number of failed steal attempts. Thus BFS_{WSL} achieves better load-balancing than BFS_{WS} which translates into better running time for BFS_{WSL}. Since BFS_{WSL} does not use locks, there is no failed steal attempt as a result of choosing a victim that is already locked. Instead, in BFS_{WSL} more steal attempts failed because the segment obtained by the thief is either too small (e.g., could happen if the victim does not have any work and so is also trying to steal), or stale (e.g., could happen if two thieves are stealing from the same victim), or invalid (e.g., could happen if more than one thief are trying to steal from the same victim and thus mess up the queue indices). In both implementations most of the steal attempts failed because of the large value used for MAX-STEAL-ATTEMPTS. A large MAX-STEAL-ATTEMPTS results in a large number of failed steal attempts at the end of each level which is reflected in the large number of steal attempts that failed because of idle victims.

Program	Time (sec)	Total Steal Attempts	Failed Steal Attempts Due To						Successful Steal Attempts
			Victim Locked	Victim Idle	Segment Too Small	Stale Segment	Invalid Segment	Total	
BFS_{WS}	7.72	732,535 (100.00%)	265,198 (36.20%)	271,731 (37.09%)	137,675 (18.79%)	49,387 (6.74%)	N/A	723,991 (98.83%)	8,544 (1.17%)
BFS_{WSL}	7.53	734,535 (100.00%)	N/A	268,710 (36.58%)	399,840 (54.43%)	56,849 (7.74%)	221 (0.03%)	725,620 (98.79%)	8,915 (1.21%)

TABLE 7.6: *Statistics of successful and failed steal attempts on the Wikipedia graph when run from 100 sources. For each program we report the average of 5 independent runs. Implementations are in* `Cilk++`.

7.10.4 Explicit vs. Implicit Work-stealing

Explicit Work-stealing: Performance difference of OpenMP and Cilk++. Theoret-

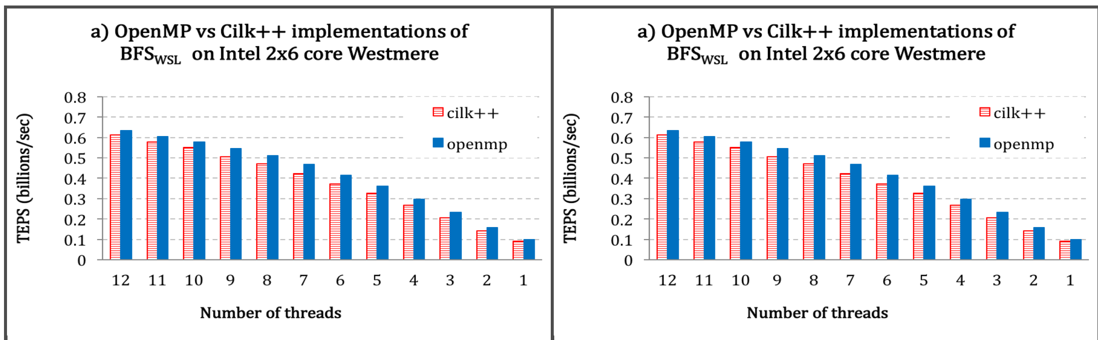


FIGURE 7.3: *Performance in terms of TEPS for Wikipedia Graph on (a) Lonestar (12 cores) and (b) Trestles (32 cores). All implementations are in* `Cilk++`.

ical and experimental results show that BFS_{WSL} is the most scalable version among all our proposed BFS algorithms. Hence, we've re-implemented it in `OpenMP` and found that the `OpenMP` implementation runs slightly faster than the original `Cilk` implementation. Figure 3 compares the performance of the two implementations on Lonestar and Trestles as the number of threads is varied. Conversion from `Cilk` to `OpenMP` is easy as our explicit work-stealing implementations are not tied to `Cilk's` features, e.g., `Cilk`'s work-stealing scheduler, nested parallelism, hyperobject library, etc.

Explicit vs. Implicit Work-stealing. To show the performance difference between explicit and implicit work-stealing for load-balancing, we've implemented BFS_{WSL} (uses explicit work-stealing) and BFS_{WSLDQ} (uses recursive divide and conquer and hence implicit work-stealing provided by `Cilk's` runtime scheduler) algorithms in `Cilk Plus` and run them on a single compute node (16-core Intel Sandy Bridge Xeon) of Stampede. Table 7.7 shows that the BFS_{WSLDQ} implementation performs better than both the `OpenMP` and `Cilk Plus` implementations of BFS_{WSL}. Since our work-stealing implementation is not as optimized as the Intel Cilk Plus's internal work-stealing, the result was not unexpected.

Machine	Lonestar			Stampede		
Cores	12	12	12	16	16	16
Parallel Platform	Clik++	OpenMP	Clik++	Cilk Plus	OpenMP	Clik Plus
Graph	Using Explicit Work-stealing	Using Explicit Work-stealing	Using Implicit cilk++ Work-stealing	Using Explicit Work-stealing	Using Explicit Work-stealing	Using Implicit Cilk Plus Work-stealing
Kkt-power	0.4	0.4	0.3	0.3	0.132	0.2
Cage14	42.6	32.2	32.9	24.6	22.381001	21.1
Freescale	37.0	40.0	35.7	28.8	23.599001	25.9
Wikipedia	73.4	69.4	72.1	58.6	58.196999	51.7
Cage15	141.0	120.0	125.7	90.6	85.067001	77.4
Graph-10M-100M	344.8	315.0	323.6	244.3	234.138	212.1

Table 7.7: *Running time (milliseconds) on Lonestar and Stampede for cilk++, Cilk Plus and OpenMP implementations (performance difference between explicit and implicit work-stealing (recursive divide and conquer)).*

7.10.5 Performance of Work-stealing on Intel Xeon Phi

Xeon Phi™(also called Many Integrated Cores or MIC) is a comparatively newer release of Intel® family featuring many smaller cores, many more hardware hyper threads, and wider vector units targeting highly parallel applications. Since BFS algorithm in general has very irregular accesses to memory, BFS can actually benefit from hyper-threading and better bandwidth that Xeon Phi offers. Hyper-threading can somewhat hide the delay of irregular memory accesses. That is why it is interesting to analyze performance of BFS algorithms on Xeon Phi architecture.

Each Stampede compute node is connected to a 61-core Intel Xeon Phi (Knights Corner) co-processor via a PCIe bus. The Xeon Phi has a DDR5 8GB memory, private L1 and L2 Caches. These 61 physical cores are connected by a bidirectional ring. Moreover, each core can run 4 hardware threads resulting in a total of 244 threads, and can reach a peak performance of 1TFLOPS. With hyper-threading, Xeon Phi schedules hardware threads running on the same core in a round-robin fashion, i.e., the same thread is not scheduled on the same core in back to back cycles. Therefore, to keep all cores busy at least 90% of the time, we should use at least 2 threads per core. Xeon Phi can run programs in two different modes: `Native` and `Offload`. In

`Native` mode Xeon Phi works as a general CPU, and in `Offload` mode it works as a coprocessor and the Xeon CPU offloads portions of its computations to the Xeon Phi. For all the experiments presented in this section, we used Xeon Phi in its `native` mode.

Scalability on Xeon Phi. In this section we show the scalability of BFS_{WSL} and BFS_{WSLDQ} on Xeon Phi.

For these scalability experiments, we've used the `Cilk Plus` versions of the programs only, as those were faster than their `cilk++` counterparts. We've also converted the code for Baseline1 to `Cilk Plus` for fair comparison. Figure 7.4 shows running times of BFS_{WSLDQ} for all graphs (expect the 1B graph which required more than 8GB of memory) on Xeon Phi. The Figure shows that running time decreases with number of threads for large graphs (especially more nicely for the scalefree Wikipedia and RMAT graphs). As expected, hyper-threading turned out to be favorable for memory-intensive BFS algorithm.

#Cores →	MIC/30	MIC/61	MIC/122	MIC/183	MIC/244
Graph	Cilk Plus BFS_{WSLDQ} Time in ms				
Kkt-power	1.4	1.7	2.4	2.9	4.2
Freescale	152.1	114.9	110.0	113.3	122.1
Cage14	125.9	79.3	61.5	56.5	57.4
Wikipedia	418.2	254.9	194.8	182.6	180.0
Cage15	424.5	238.8	163.6	56.5	130.5
Graph-10M-100M	1509.6	742.4	421.0	313.6	259.1

FIGURE 7.4: *Scalability of implicit and explicit work-stealing algorithms (Cilk Plus implementations) on Xeon Phi/MIC.*

Figure 7.5 shows speedup (T_1/T_p) obtained by BFS_{WSLDQ} and Baseline1 on `inline_1 stiffness matrix` graph. The diameter of the `inline_1` graph is quite large, and the number of vertices at each level is comparatively lower considering a large number of available threads. Clearly, for a graph with a large diameter and fewer vertices at each level, it is not beneficial to use many cores for level-synchronous BFS, because synchronization overhead increases with the diameter of the graph and the number of threads. In fact, the amount of work at each level becomes comparatively less than the overhead of scheduling and dynamic load-balancing for a large number of threads. For `inline_1`, the scalability curves of Baseline1, BFS_{WSLDQ} and $BFS_{WSL-OMP}$ followed trends similar to that reported by authors of [158] for their algorithm and baseline; however, our speedup decreased slowly compared to their implementations. To see the overhead of work-stealing we experimented with both `MAX-STEAL-ATTEMPTS = ` p and `MAX-STEAL = ` $p \log p$. We found that $BFS_{WSL-OMP}$ is faster (actually fastest) for `inline_1` if we use `MAX-STEAL-ATTEMPTS = ` p.

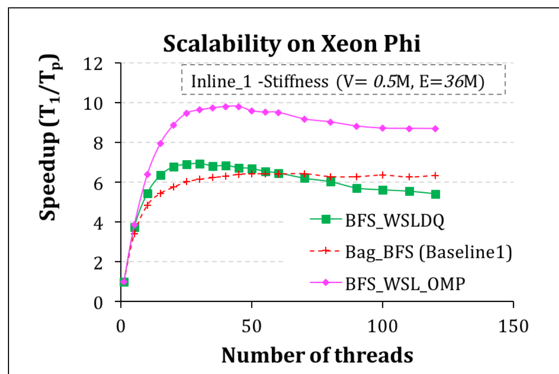


FIGURE 7.5: *Scalability of implicit and explicit work-stealing algorithms (Cilk Plus implementations) on Many Integrated Cores: on inline_1 Graph.*

Figure 7.6 shows speedup obtained by BFS_{WSLDQ} and Baseline1 on the scalefree Wikipedia and RMAT graphs. Note that although BFS_{WSLDQ}, $BFS_{WSL-OMP}$ and Baseline1 did not perform that well on `inline_1`, for both the Wikipedia and RMAT graphs BFS_{WSLDQ} and $BFS_{WSL-OMP}$ scaled till 243 threads. The reason behind this change in performance becomes clear when we analyze the nature of the graphs. Table 7.8 shows the number of vertices explored

Graph	Wikipedia	inline_1	RMAT
Depth from Source	n : 3566907, m : 45030389	n : 503712, m : 36312630	n : 10000000, m : 100000000
0	1	1	1
1	1	53	14
2	36	369	175
3	2582	729	3574
4	391296	1011	104083
5	1592339	1719	2302103
6	637166	2190	5359300
7	47556	2403	534230
8	4033	2076	9721
9	679	2271	136
10	201	2547	4
11	51	2829	-
12	16	3642	-
13	1	3690	-
14	3	3255	-
15-65	-	189273	-
66-115	-	178497	-
115-165	-	85203	-
>165	-	23268	-

TABLE 7.8: *Number of vertices explored at a given BFS level. Here n denotes the number of vertices and m denotes the number of edges.*

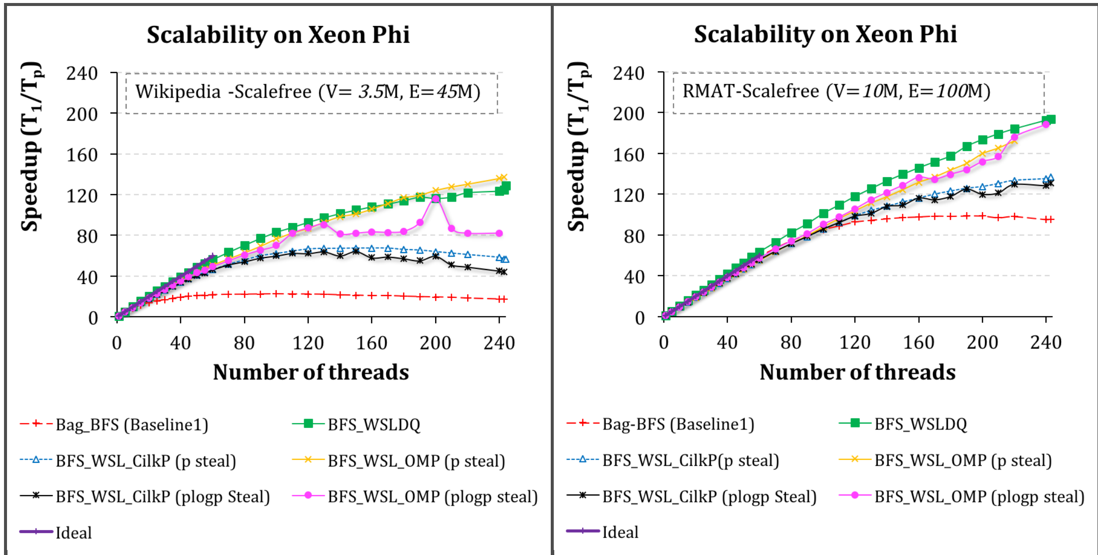


FIGURE 7.6: *Strong scalability on Intel® Xeon^{TM}Phi (a) Wikipedia, (b) RMAT-1M-100M.*

at each BFS level from a source for each of those graphs. Note that both Wikipedia and RMAT graphs are dense and scalefree, most of the work is done at middle levels where all threads have enough work to do. We see the opposite for the `inline_1` which explains why all algorithms did not scale well on inline_1. Also note that the RMAT graph is even denser than Wikipedia and has a smaller diameter which also tells why scalability is nicer for RMAT than Wikipedia. We observed that on the Wikipedia graph with `MAX-STEAL-ATTEMPTS = ` p, the OpenMP-based work-stealing implementation scaled still 243 threads on Xeon Phi which was slightly better than BFS_{WSLDQ}. However, for the RMAT graph, BFS_{WSLDQ} outperformed others. For RMAT graph (dense, low-diameter and scalefree), the difference between `MAX-STEAL-ATTEMPTS = ` p and `MAX-STEAL-ATTEMPTS = ` $p \log p$ is negligible. The Cilk Plus BFS_{WSLDQ} and BFS_{WSL} implementations scaled nicely till 130 and 243 threads for Wikipedia and RMAT, respectively.

7.11 Conclusion and Future Research

We have presented two types of lockfree parallel BFS algorithms along with their variants based on centralized job queues and distributed randomized work-stealing and analyzed their theoretical and experimental performance on different architectures (Intel's 12-core Westmere, 16-core Xeon, 61-core Xeon Phi, and 32-core AMD Magny Cours). We have used a novel optimistic parallelization technique to avoid the use of locks and atomic instructions. Although work-stealing is fairly popular technique for dynamic load-balancing, lockfree work-stealing is novel for BFS. Experimental results show that these algorithms are highly scalable and perform very well on massive, scalefree and sparse graphs, and achieve better performance compared to two other state-of-the-art algorithms on multi-cores and many-integrated-cores.

It would be interesting to see if optimistic parallelization technique can be used to improve the performance of other nontrivial parallel applications that require dynamic load-balancing. Lockfree optimistic parallelization for approximation algorithms where some error in the result is acceptable is also an interesting research direction to pursue.

Chapter 8

Theoretically Optimal Level-synchronous Parallel Work-aware BFS

8.1 Abstract

We present a work-aware work-efficient parallel level-synchronous Breadth-first Search (BFS) algorithm for shared-memory architectures which achieves the theoretical lower bound on parallel running time. The optimality holds regardless of the shape of the graph. We also demonstrate the implication of this optimality on the energy consumption of the program empirically. The key idea is to never use more processing cores than necessary to complete the work in any computation step efficiently. We keep rest of the cores idle to save energy and to reduce other resource contentions (e.g., bandwidth, shared caches, etc.). Our BFS algorithm does not use locks and atomic instructions and is easily extendible to shared-memory coprocessors.

8.2 Introduction

Given a graph $G = (V, E)$, with vertex set V ($|V| = n$), edge set E ($|E| = m$) and diameter D, a level-synchronous Breadth-first Search (BFS) traverses the graph from a given source vertex, s level by level, exploring the first level neighbors of s first, then the second level neighbors and so on until all vertices reachable from the s are explored. A straightforward way to implement a parallel level-synchronous BFS is to explore all vertices at a given level as well as all neighbors of a given vertex in parallel which takes $\mathcal{O}\left(D \log(m+n)\right)$ time using $\Theta(n+m)$ cores, since launching and synchronizing $m+n$ threads require $\Theta\left(\log(m+n)\right)$ time. In practice, the number of available processors $P \ll m + n$. The amount of parallelism achievable at each BFS level is constrained by the number of nodes and edges to be explored in that particular level, making performance of a BFS algorithm highly dependent on the structure of the graph itself.

In this chapter, we first analyze the theoretical lower bound for a level-synchronous BFS on shared-memory architectures assuming that k threads can be launched and synchronized in $\mathcal{O}(\log k)$ time, and then we present our work-aware BFS algorithm that is able to archive that optimal lower bound.

Let T_s be the running time of the most efficient serial level-synchronous BFS algorithm, and T_P be the running time of the most efficient parallel level-synchronous BFS algorithm on $P \geq 1$ processing cores.

Lower Bound for Parallel BFS. Since $T_s = \Theta(m+n)$, clearly, $T_P = \Omega\left(\frac{m+n}{P}\right)$. If W_l is the amount of work at level l of a level-synchronous BFS, an optimal algorithm will incur $\Theta(\log \min(P, W_l))$ synchronization overhead in that level.
Hence, $T_P = \Omega\left(\frac{m+n}{P} + \sum_{l=1}^{D} \log \min(P, W_l)\right)$. □

Apart from theoretical optimality, it has been observed that (see Table 7.8 in Chapter 7) for many practical (especially scalefree) graphs, the number of vertices explored at the beginning and at the end of a BFS exploration is significantly smaller than in the middle levels [105]. Therefore, using the same amount of computational resources (i.e., the number of cores/threads, even caches) at all levels of the BFS is neither economical nor efficient. Indeed, with the increase of the number of active threads synchronization overheads, false sharing, conflict misses and DRAM and CPU energy/power also increase, and without enough work, these overheads may dominate the running time. In our work-aware BFS algorithm, we fix the number of cores based on the amount of work at each computation step instead of using all cores across the entire BFS. We demonstrate the impact of this choice on theoretical optimality as well as on energy performance on modern multicores.

8.3 Algorithm

Ideally, to achieve optimal time complexity bound for BFS we should be able to explore all vertices and all edges in parallel without incurring any asymptotically dominating overhead for work distribution. With that in mind, we first designed a BFS algorithm that uses optimal prefix sum, splitting, binary search and scanning at each level for work-distribution and vertex exploration. Prefix sum is used to compute actual amount of work that needs to be done at any step which then equally gets split among the required number of workers. Each worker then independently searches for the work item (e.g., vertex and edge id) from which it needs to start working on. However, since each thread uses binary search to find the work item, this algorithm does not achieve the theoretical optimality due to the $O(Dp \max(\log n, \log m))$ cost for the overall binary search.

Therefore, to achieve optimal time-complexity, we need to replace binary search with another algorithm whose cost does not asymptotically increase the time-complexity of vertex exploration. In the following part, we describe our work-aware BFS algorithm that achieves that goal.

Theoretically optimal BFS Our work-aware BFS algorithm uses parallel prefix-**S**um, **S**plitting, and **S**canning to achieve work-optimality for arbitrarily shaped graphs. Figure 8.1 shows the pseudocode of the algorithm. We use the CurLevelVertices array to store vertices that are

going to be explored in a level. The **parallel for** loops are implemented using recursive divide and conquer where the recursion stops and switches to an iterative loop when a division size becomes $\leq$ GRAINSIZE and then each thread executes one of those divisions iteratively in parallel with other threads.

All major computations in this BFS algorithm are performed using ***parallel for***s. The main loop in lines $9-31$ executes until CURLEVELVERTICES becomes empty. The degrees of the vertices in CURLEVELVERTICES are collected in EDGESUM and then PARALLEL-PREFIX-SUM [24] is used to count the number of edges to be explored in the current level. Then each thread in parallel determines its start location (start vertex and start edge of that vertex) in CURLEVELVERTICES assuming an (almost) equal division of work. Each active thread explores its assigned segment of edges and stores the newly discovered vertices in a private queue. A thread also marks itself as an owner of a vertex that it stores in its queue for the next level. If multiple threads claim ownership of a vertex, only one of them becomes the owner (benign race). After the exploration, each thread deduplicates its own output queue by keeping only the vertices for which it is the final owner. Next, each thread copies the unique vertices left in its queue to the CURLEVELVERTICES, and the next level of BFS starts.

The FIND-STARTEXPPOINT[1] module demands separate explanation since it replaces the original binary search. In this function STARTEXPPOINT[t] stores an index from the CURLEVELVERTICES array denoting where thread t should start exploring, and STARTEXPPOINT[$t+1$] stores where thread t should stop and thread $t+1$ should start exploring. Lines $2-3$ are used to find an optimal number of threads necessary to execute this function. We divide the work of edge-exploration evenly among the threads. The STARTTHREADID variable is used to store a thread id that can potentially start at the ith edge, and ENDTHREADID is used to store a thread id that can potentially end at the ith edge in the EDGESUM array. In Lines $5-12$, for each entry in the EDGESUM array in parallel we determine the STARTTHREADID (i.e., a thread that should start exploring an edge connected to that node) and ENDTHREADID (i.e., a thread that could possibly end at that node). This range will be empty for all except $\leq P_{l_i}$ of them, where $P_{l_i} = min(P_l, q_l)$. Here, q_l denotes the number of vertices to be explored at BFS level l, and $P_l = P$ denotes the number of threads that is passed as a parameter to this function. For each nonempty range, we write this node's position as the corresponding thread's start location. In other words, if there is any such thread, we store this vertex position i at the corresponding thread's STARTEXPPOINT location. If there are $k \geq 1$ threads who should start from that vertex, we use exactly k threads to write the vertex position on those threads' STARTEXPPOINT locations. Observe that, the total work done by this function is only $O(q_l + P_{l_i})$.

For all other modules including prefix sum, we modified the algorithm to make sure that we never use more cores than needed.

8.4 Analysis

Our work-aware BFS algorithm achieves the lower bound proved before as follows. The entire BFS algorithm can be divided into D levels where each level l consists of a constant (say, c)

[1]This function was worked out by Yoni Fogel.

```
PARALLEL-BFS( s, P ) s is the source vertex from which distance is calculated. P is the maximum number of processors
to use. Returns D[0 : n − 1] which represents the distance from s to each vertex.
 1. P_l ← MIN( P, n )
 2. par for u ← 0 to n − 1 do //for grainsize = n/P_l.
 3.     D[u] ← ∞, OWNER[u] ← P //initialize distance & owner.
 4. D[s] ← 0, OWNER[s] ← 0 //initialize source values.
 5. CURLEVELVERTICES ← Arr[0 : n − 1] //current level nodes.
 6. EDGESUM ← Arr[0 : n − 1]//holds prefixsum of degrees.
 7. CURLEVELVERTICES[0] ← s //insert source vertex.
 8. L ← 0, P_l ← 1 //initial level and number of threads.
 9. while n_l ← |CURLEVELVERTICES| > 0 do //any vertex left.
10.     L ← L + 1 //n_l = #vertices at level l.
11.     P_l ← min(P, n_l)//P_l chosen based on max work.
12.     par for u ← 0 to n_l − 1 do //for grainsize = n_l/P_l.
13.         EDGESUM[u] ← Degree[CURLEVELVERTICES[u]]
14.     PARALLEL-PREFIX-SUM( EDGESUM, n_l, P_l ) //degree sum.
15.     e_l ← EDGESUM[n_l − 1] //total edges going to be explored.
16.     P_l ← MIN( P, e_l )//P_l chosen based on max work.
17.     STARTEXPPOINT ← FIND-STARTEXPPOINT( EDGESUM, e_l, P_l ) //Find starting exploration point for each
    thread.
    //main Exploration with optimal number of threads.
18.     Q ← EXPLORE-VERTEX (CURLEVELVERTICES, EDGESUM, STARTEXPPOINT, D, Neighbor, Degree, e_l, P_l, L)
    //deduplication (keep single copy of a vertex).
19.     SIZES ← Arr[0 : P_l − 1] //stores output Q sizes.
20.     par for i ← 0 to P_l − 1 do //for grainsize = 1.
21.         Q-NEW ← queue //temporary local queue.
22.             for v ∈ Q[i] do if OWNER[v] = i then Q-NEW.ENQUEUE( v ) //keeps only the owned vertices.
23.         Q[i] ← Q-NEW, SIZES[i] ← |Q[i]| //recollect the size of the deduplicated queues.
24.     PARALLEL-PREFIX-SUM( SIZES, P_l, P ) //compute the total size of the next level queue.
    //linearization (copy from distributed Q to a linear Q). By this time each thread exactly knows how much to
    copy and from where to where.
25.     CURLEVELVERTICES ← Arr[0 : SIZES[P_l − 1]]
26.     par for i ← 0 to P_l − 1 do
27.         if i = 0 then OFFSET ← 0 else OFFSET ← SIZES[i − 1]
28.         for j ← OFFSET to OFFSET + |Q[i]| do
29.             CURLEVELVERTICES[j] ← Q[i].DEQUEUE( )
```

FIGURE 8.1: *Theoretically optimal work-aware level-synchronous parallel breadth-first search.*

number of steps. All of these steps $\{i\}$ are implemented using **parallel for** loops. For each such step i with work $W_{l,i}$, we choose $P_{l,i} = \min(P, W_{l,i})$. We distribute the work among $P_{l,i}$ threads in $\Theta(\log P_{l,i})$ parallel time. After that each thread performs $\frac{W_{l,i}}{P_{l,i}}$ work serially. Hence, the parallel running time of that step is $\mathcal{O}\left(\frac{W_{l,i}}{P_{l,i}} + \log P_{l,i}\right)$ which reduces to $\mathcal{O}\left(\frac{W_{l,i}}{P} + \log P\right)$ when $P_{l,i} = P$, and to $\mathcal{O}(\log W_{l,i} + 1)$ when $P_{l,i} = W_{l,i}$. Hence, $\mathcal{O}\left(\frac{W_{l,i}}{P} + \log\min(P, W_{l,i}) + 1\right)$ is the parallel running time covering both cases.

Therefore, $T_P = \sum_{l=1}^{D}\sum_{i=1}^{c} \mathcal{O}\left(\frac{W_{l,i}}{P} + \log\min(P, W_{l,i}) + 1\right)$
$= \mathcal{O}\left(\frac{\sum_{l=1}^{D}\sum_{i=1}^{c} W_{l,i}}{P} + \sum_{l=1}^{D}\sum_{i+1}^{c}\log\min(P, W_{l,i}) + cD\right) = \mathcal{O}\left(\frac{m+n}{P} + \sum_{l=1}^{D}\log\min(P, W_l)\right)$,
where $W_l = \sum_{i=1}^{c}(W_{l,i})$.

FIND-STARTEXPPOINT(EDGESUM, W, P) *EdgeSum* is an array that stores prefix sum of number of edges of each vertex in the CURLEVELVERTICES array. W is the total number of edges thats going to be explored in this BFS level. This function returns StartExpPoint array of sixe P which stores from where in the CURLEVELVERTICES array each thread should explore to have a balanced work load. This function replaces binary search and helps us to achieve optimality.

1. STARTEXPPOINT $\leftarrow$ ***Arr***$[0 : P - 1]$
2. $N \leftarrow |$EDGESUM$|$
3. $P_n \leftarrow$ MIN(P, N)
4. #pragma grainsize N/P_n

   //Here for each edgesum from the EdgeSum Arr, in parallel we try to figure out what is the thread id that could possibly start/end from/to that location.
5. ***par for*** $i \leftarrow 0$ ***to*** $N - 1$ ***do***
6.   ***if*** $i = 0$ ***then***
7.   STARTTHREADID $\leftarrow 0$
8.   ***else***
9.   STARTTHREADID $\leftarrow \left\lceil \frac{P \cdot \text{EDGESUM}[i-1]}{W} \right\rceil$ //Possible thread id that could start here
10.   ENDTHREADID $\leftarrow \left\lceil \frac{P \cdot \text{EDGESUM}[i]}{W} \right\rceil - 1$ //Possible thread id that could end here
11.   ***par for*** $t \leftarrow$ STARTTHREADID ***to*** ENDTHREADID ***do*** //Use cores STARTTHREADID to ENDTHREADID
12.   STARTEXPPOINT$[t] \leftarrow i$
13. ***return*** STARTEXPPOINT

FIGURE 8.2: *An efficient way to find starting point in a list of vertices/edges from where a thread should start working on to get even partitioning of work.*

8.5 Experimental Results

We implemented our work-aware BFS algorithm using Intel® `Cilk Plus`. In this section, we show some experimental results to demonstrate the performance of our BFS algorithm on many real-world and synthetic (RMAT) graphs. We compiled our program using `‘‘-O3 -ansi-alias -opt-subscript-in-range’’` parameters and turned off the deduplication option for the experiments. Deduplication, indeed, improved performance for the dense synthetic RMAT graphs, whereas, for most of the others, performance degraded. A description of the input graphs used in these experiments can be found in Table 8.1. We need to keep in mind that, the practical benefit of this work-aware BFS should be more visible when the number of cores in the machine is more than the amount of work we have in each computation step. Therefore, our algorithm should run more efficiently on machines with a very large number of cores. When the number of cores is small, the overhead of extra work can degrade performance.

Graph	N	M	MaxD	Graph	N	M	MaxD
kkt-power	2.1E+06	8.1E+06	13	as-skitter	1.7E+06	1.1E+07	33
ca-AstroPh	1.9E+04	4.0E+05	12	freescale	3.4E+06	1.9E+07	148
com-amazon	3.3E+05	9.3E+05	13	cage14	1.5E+06	2.7E+07	38
com-dblp	3.2E+05	1.0E+06	25	com-lj	4.0E+06	3.5E+07	52
RoadNet-PA	1.1E+06	1.5E+06	146	wiki	3.6E+06	4.5E+07	16
RoadNet-TX	1.4E+06	1.9E+06	33	cage15	5.2E+06	9.9E+07	49
RoadNet-CA	2.0E+06	2.8E+06	37	RMAT100M	1.0E+07	1.0E+08	12
com-orkut	3.1E+06	3.1E+06	33	RMAT1B	1.0E+07	1.0E+09	6
-	-	-	-	com-friendster	6.6E+07	1.8E+09	47

TABLE 8.1: *Input graphs and their properties. Here N = #vertices, M = #edges, and MaxD = maximum diameter explored.*

Property	Intel32	Intel16	Xeon Phi	Intel16E
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 8.2: *System specifications. Intel16E is used for Power and Energy analyses.*

We used machines from the Stampede Supercomputing Cluster [6] to run the experiments and the system specifications can be found in Table 8.2. We used LIKWID[184] and MSR modules to measure the energy/power consumed by the program. We measure performance in terms of Traversed Edges Per Second (TEPS), Seconds, Joules and Watts. Only for the com-friendster graph, we used the Intel32 machine that had 1TB of memory. All other programs were run on the Intel16 machine. Energy and power statistics were collected using our inhouse Intel16E machine.

Performance on Xeon multicores. Figure 8.3 shows the runtime performance in terms of TEPS on the input graphs shown in Table 8.1. We used 100 - 1000 random sources and took the average during the TEPS computation. As the Figure shows, work-aware BFS achieves up to 3.5 Billion TEPS for large graphs and 50 Billion TEPS for small graphs.

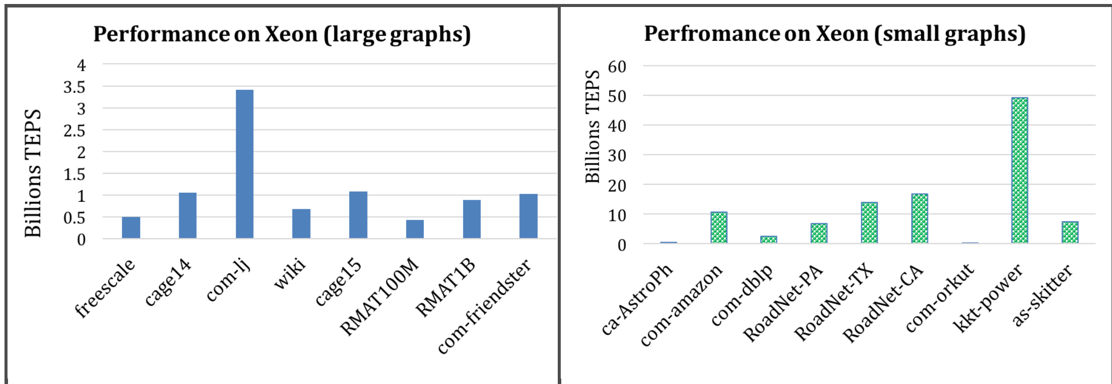


FIGURE 8.3: *Performance on Xeon (multicores).*

Performance on Xeon Phi manycores. Figure 8.4 shows the performance of our work-aware BFS algorithm on Intel®Xeon™Phi in its native mode (where Xeon Phi works as a general CPU instead of a coprocessor). It shows that work-aware BFS achieves around 0.8 billion TEPS on large graphs and 10 Billion TEPS on small graphs.

Scalability on Xeon and Xeon Phi. Figure 8.5 shows scalability of work-aware BFS on a 16-core Xeon and 61-core Intel Xeon Phi architecture in its native mode on different real-world and synthetic graphs. It shows that work-aware BFS scales almost linearly till 120 threads (61 physical cores) on Xeon Phi for large enough graphs and keeps scaling till 244 threads. Although work-aware BFS is quite scalable on Xeon, as explained earlier, for very small number of cores,

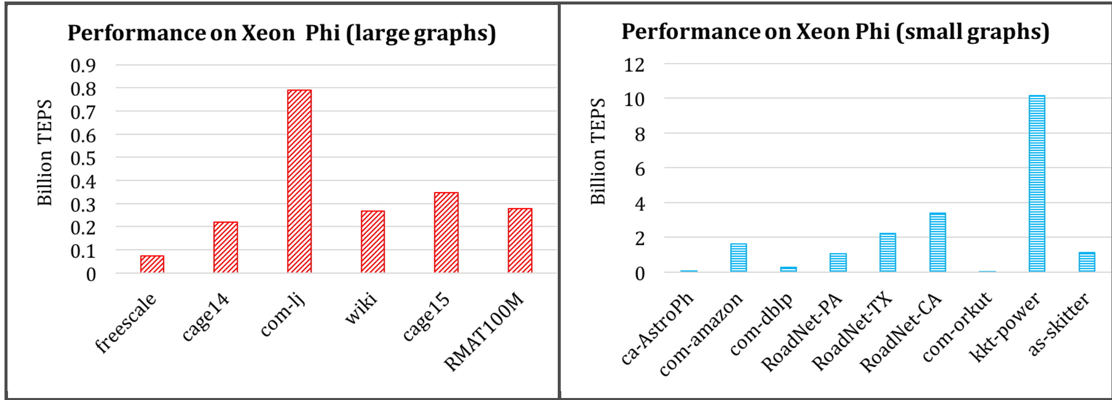


FIGURE 8.4: *Performance on Xeon Phi (manycores).*

the extra work done to achieve theoretical optimality really does not pay off and sometimes degrades performance.

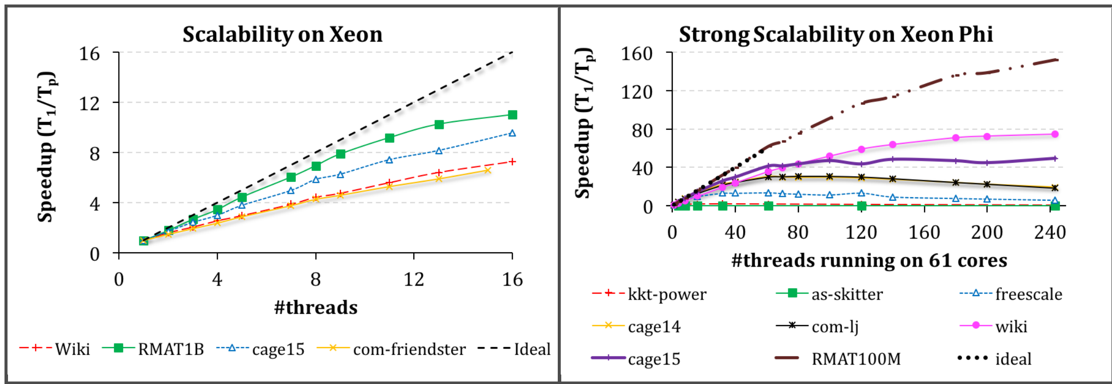


FIGURE 8.5: *Strong scalability on Xeon and Xeon Phi.*

Energy and Power Benefit. To show the energy benefits of choosing an optimal number of cores at each computation step, we profiled work-aware BFS while running on the Wikipedia graph with 1000 random sources. We repeated each run 30 times with each source, and the entire experiment 4 times, and took the average. We ran a work-unaware version of the work-aware BFS in which the maximum number of cores were used for the entire computation. We did not explicitly turn off the cores or caches, nor made the threads sleep. Neither we used DVFS (dynamic voltage frequency scaling) to reduce the frequency of the unused cores. Instead, we spawned the optimal number of threads before each operation. The expectation is that; the other threads will not use DRAM or other shared resources.

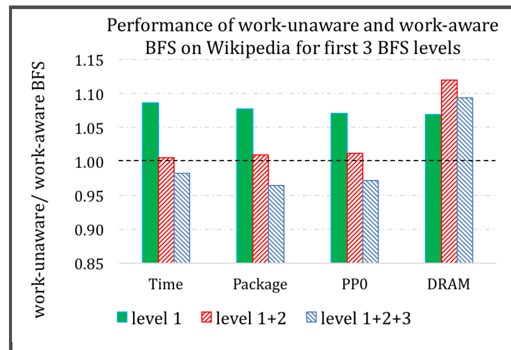


FIGURE 8.6: *This figure shows energy consumption by different components of the machine (Package = Socket (CPUs), PP0 = Power Plane0) for the first* 3 *BFS levels. Here 1 denotes energy consumed at BFS level 1, 1 + 2 denotes energy consumed at levels 1 and 2, and 1 + 2 + 3 denotes energy consumed by level 1, 2 and 3.*

Figure 8.6 shows a performance comparison of these two versions. In this figure a ratio > 1 means that the work-aware BFS is doing better, otherwise, the work-unaware version is doing better. Since the first two BFS levels of the Wikipedia graph typically have very few vertices and edges (see Table 7.8 in Chapter 7), the work-aware version should win, which is, indeed, the case in Figure 8.6. Since in level 3 the amount of work at each step is more than the maximum number of cores in our multicore machine (16-core Xeon Sandybridge), the ratios drop slightly bellow 1. Even then interestingly the ratios for DRAM energy and power consumption are still > 1. This shows potential energy benefits of work-aware BFS especially while running on thousands of cores.

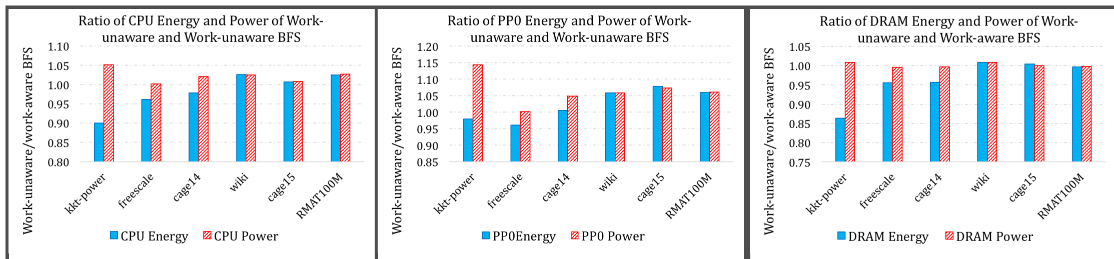


FIGURE 8.7: *Energy and power efficiency on many real-world graphs.*

Next we ran these work-aware and work-unaware BFS implementations on Intel16E for five graphs from our input set and collected the CPU, DRAM, and Power Plane 0 (uncore) energy and power. Figure 8.7 shows the ratio of CPU energy and power, ratio of Power Plane 0 (uncore including Last Level Cache) energy and power and ratio of DRAM energy and power consumption. In all these cases, a value > 1 is better. For almost all cases, the work-aware implementation consumes less energy as well as power and runs faster than the work-unaware version.

8.6 Conclusion and Future Research

We present a theoretically optimal level-synchronous work-aware parallel BFS algorithm which achieves optimality and reduces energy consumption by actively controlling the number of cores used in each computation step. We empirically show that for levels where the amount of work is significantly less than the available cores, this approach reduces energy and power considerably. This algorithm can bring more energy benefits while running BFS on thousands of processing cores and can be optimized with all known optimizations (e.g., direction optimizations, NUMA awareness). We believe similar techniques can be used for distributed and hybrid-coprocessor settings, which probably can bring even more energy savings.

Extending this idea to develop energy-efficient breadth-first search algorithm targeting distributed and distributed-shared memory platforms is interesting future research direction to pursue.

8.7 Acknowledgment

I would like to thank Yoni Fogel for working with me in this project and all his supports. We independently came up with similar algorithms for breadth-first search at the same time.

Part III

Algorithms using Spatial Trees

Chapter 9

Hybrid Algorithm using Octrees: Polarization Energy on Clusters of Multicores

9.1 Abstract

Computing polarization energy between a ligand (i.e., a small molecule such as a drug molecule) and a receptor (e.g., a virus molecule) is of utmost importance in drug design. We have designed and implemented *cache-oblivious recursive divide-and-conquer distributed-memory* and *distributed-shared-memory* parallel algorithms for approximating Generalized Born (GB) polarization energy (e.g., polar part of free energy of hydration) of protein molecules. This is an octree-based hierarchical algorithm, built on Greengard-Rokhlin type near-far decomposition of data points (i.e., atoms and points sampled from the molecular surface) for calculating the polarization energy of protein molecules using the surface based $\mathbf{r^6}$-approximation of Generalized Born radii of atoms. We have shown that our implementations outperform state-of-the-art GB-polarization energy implementations, such as *Amber* 12, $\mathbf{GBr^6}$, *Gromacs 4.5.3*, *NAMD 2.9* and *Tinker 6.0.* Using numerical and algorithmic approximations, cache-efficient data structures, efficient load-balancing and parallelization schemes, we achieve over 400× speedup w.r.t Amber with less than 1% error w.r.t. the naïve exact algorithm using as few as 144 cores (i.e., 12 compute nodes with 12 cores each) for molecules with as many as half a million of atoms.

9.2 Introduction

Whenever a molecule comes under the influence of an electric field, its charge distribution is relaxed in response to that field. The energy associated with this relaxation is known as the polarization energy (E_{pol}). It is typically negative in quantity, as a relaxation leads to decrease in energy [134]. Electronic polarization plays a crucial role in drug design, discovery and design of

new proteins, antivirus and antibiotics, protein-protein docking, molecular dynamics simulations for determining the molecular conformation with minimal total free energy, and so on.

The Poisson-Boltzmann [17, 88, 104, 124] model can be used to approximate E_{pol}. However, due to its high computational costs Poisson-Boltzmann method is rarely used for large molecules such as proteins. Instead E_{pol} is approximated using the Generalized Born (GB) model [100, 141, 166] – a popular approximation model which considers solvent as a statistical continuum. However, computing E_{pol} naïvely even based on the GB model takes time quadratic in the number of atoms in the molecule, and thus it remains computationally expensive for large molecules. Hence, *another level of approximation over the original GB-approximation is required* in order to reduce its complexity below quadratic, and preferably to linear.

An additional level of performance boost can be gained in GB-approximation by introducing parallelism in the computation [108]. Before multicores became widely available, distributed-memory parallel algorithms were typically used in high-performance parallel computing, and these algorithms were designed to use explicit distribution and communication of data among the compute nodes. Even though multicore computers allow implicit communication among the cores through the memory hierarchy and the shared memory space, when run on clusters of multicores, pure distributed-memory algorithms typically require separate memory space for each core of the same compute node, and explicit communication among the cores. One natural way of reducing excessive data replication and explicit communication among the cores of a compute node is to use hybrid algorithms – algorithms that use shared-memory parallelism inside each multicore node and distributed-memory parallelism across the nodes of the cluster. The goal is to reduce space usage (due to data replication) and communication time (due to explicit communication among threads) whenever possible.

The main contribution of this work is a hybrid distributed-shared-memory parallel algorithm for approximating GB polarization energy on a cluster of multicores. We use a fast approximation scheme based on a hierarchical spatial decomposition of the molecule[1] [40, 41], and apply a Greengard-Rokhlin type near-far approximation scheme [93] on the decomposition. We also present detailed performance results of our approach. We show that it runs faster than other state-of-the-art implementations of GB polarization energy namely, *Amber* 12 [57], *GBr*^{6} [183], *Gromacs 4.5.3* [102], *NAMD 2.9* [147, 174] and *Tinker 6.0* [72], and can handle molecules larger than most of them can process. We have also compared our hybrid algorithm with our own purely distributed-memory implementation of the same algorithm. We found that though for small molecules the hybrid algorithm runs slower, it outperforms the distributed-memory version as the size of the molecule increases.

The distributed-shared-memory and distributed-memory algorithms are based on the prior shared-memory cache-oblivious recursive divide-and-conquer algorithm [40, 41] which somewhat shows portability of a cache-oblivious recursive divide-and-conquer (CORDAC) approach to different platforms. Our distributed-shared-memory algorithm has the following properties:

- **Hybrid parallelism** ▷ We use shared-memory parallelism inside each compute node and distributed-memory parallelism across the compute nodes.

[1] consisting of atoms and points sampled from the surface of the molecule

- **Cache- and space-efficient data structure** ▷ We use *octrees* [107] for finding nonbonded atoms, which, unlike traditional *nonbonded lists* [146], always use space linear in the number of atoms in the molecule independent of any distance cutoff used. Octrees are recursive data structures and also known to be cache-friendly.

- **Space-independent speed-accuracy tradeoff** ▷ The algorithm uses user-defined approximation parameters, and by tuning these parameters one can get a more accurate approximation of E_{pol} at the cost of increasing the running time and vice versa. Unlike traditional distance cutoff based methods, the space usage is independent of the values of the approximation parameters.

- **Load-balancing** ▷ Inside each compute node, we use dynamic load-balancing based on efficient randomized work-stealing [27], and across nodes, we use static load-balancing in order to reduce the communication overhead.

The rest of this chapter is organized as follows. In Section 9.3 we provide necessary background on polarization energy as well as on the data structures and algorithms we use. In Section 9.4 we describe related work on the estimation of polarization energy. Section 9.5 presents our algorithms along with their theoretical complexity analysis. In Section 9.6 we present simulation results and a detailed comparison with other existing approaches namely, *Amber* 12, *GBr*^{6}, *Gromacs 4.5.3*, *NAMD 2.9* and *Tinker 6.0*. Finally, Section 9.7 concludes this chapter with some future research directions.

9.3 Background

In this section, we first explain the mathematical expressions for estimating E_{pol}. Then we provide some background on the cache-efficient octree data structure, and the near-far approximation scheme from [40, 41] which we extend to the distributed-shared-memory setting.

Polarization Energy: The polarization energy of a molecule depends on the difference of potential of that molecule in solvent and gas-phase, and its charge density:

$$E_{pol} = \frac{1}{2}\int \O_{reaction}(r).\rho(r), \tag{9.1}$$

where $\O_{reaction}$=$\O_{solvant} - \O_{gass-phase}$, and $\O(r)$ and $\rho(r)$ are the electrostatic potential and charge density of the molecule, respectively.

In the *GB*-model, the polarization energy of a molecule is given by the following equation:

$$E_{pol} = \frac{1}{2}\left(1 - \frac{1}{\epsilon_{solv}}\right)\sum_{i,j}\frac{q_i.q_j}{{f_{ij}}^{GB}}, \tag{9.2}$$

where ${f_{ij}}^{GB} = \left[{r_{ij}}^2 + R_iR_j \exp^{\frac{-{r_{ij}}^2}{4R_iR_j}}\right]^{\frac{1}{2}}$, and ϵ_{solv} = solvent di-electric, r_{ij} = distance between atoms i and j, R_k and q_k $(k\ \epsilon\{i,j\})$ denote the Born radius and charge value of atom k, respectively.

The effective Born radius reflects how deep a charge is buried inside the molecule. The Born radius of an atom i, R_i shows the extent of interaction of the atom with a solvent when it is dissolved in that solvent. If the atom is close to the molecular surface, R_i is small. An atom with large R_i has a weaker interaction with the solvent.

To approximate Born radii and polarization energy, we have used Gaussian quadrature points sampled from the molecular surface. Gaussian quadrature attempts to obtain the best numerical estimate of an integral (e.g., molecular surface function) by picking optimal abscissas x_i to evaluate the function. Gaussian quadrature is considered to be optimal as it fits all polynomials exactly up to a certain degree [190]. The triangulation of Gaussian quadrature function of the molecular surface yields an estimation of molecular surface normal at triangulation vertices, and at Gauss quadrature numerical integrations points in each triangle's interior. A constant number of quadrature points per triangle are needed for high accuracy of the Born radii calculation.

The evaluation of Born radii is essentially based on the Coulomb field approximation [96], which assumes that the electric displacement is in the Columbic form. Using this approximation, Born radii can be calculated as follows, where x_i represents the center of atom i.

$$\frac{1}{R_i} = \frac{1}{4\pi} \int \frac{1}{|r - x_i|^4} d^3r \tag{9.3}$$

We can obtain a discrete approximation of Born radii by applying Gaussian quadrature as shown in Equation 9.4 (known as r^4-approximation) [66]:

$$\frac{1}{R_i} \approx \frac{1}{4\pi} \sum_{k=1}^{N} w_k \frac{(r_k - x_i).\overrightarrow{n_k}}{|r_k - x_i|^4}, \tag{9.4}$$

where r_ks denote N Gaussian quadrature points on the molecular surface, $\overrightarrow{n_k}$ is the unit outward surface normal at r_k, and w_k is a weight assigned to the quadrature point in order to achieve higher order of accuracy for small N. However, the following approximation of Born radius (known as r^6-approximation) shows better accuracy for spherical solutes, e.g., proteins [96]:

$$\frac{1}{{R_i}^3} = \frac{3}{4\pi} \int_{ex} \frac{dr}{|r_k - x_i|^6} \approx \frac{1}{4\pi} \sum_{k=1}^{N} w_k \frac{(r_k - x_i).\overrightarrow{n_k}}{|r_k - x_i|^6}. \tag{9.5}$$

Octrees vs. Nblists: An octree is a tree data structure that recursively and adaptively subdivides a $3D$ space into 8 octants, and is often used as a container for rectilinear scalar field data. Octrees are very cache friendly because of their recursive nature. We use octrees to store the atoms in a molecule and the surface quadrature points. Once an octree is built, it can be used for any approximation parameter (approximation parameter is sort of similar to distance cutoff used in other molecular dynamics (MD) packages). Some existing MD packages, e.g., *Amber*, *NAMD* and *Gromacs* use `nblists` (nonbonded list) to represent interacting atom pairs. The size of the `nblist` of any given atom grows linearly with the number of atoms in the system, and cubically with the distance cutoff that truncates the non-bonded interactions. On the other hand, an octree uses space linear in the number of data points it holds, and its size does not change with the approximation parameter. Updating the `nblist` after the initial construction is costly and also not scalable with the distance cutoff. Often MD implementations that use

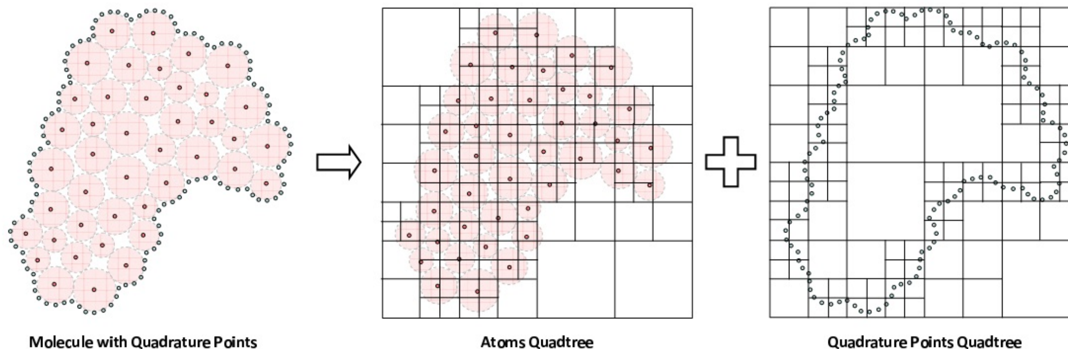


FIGURE 9.1: *In the Born radius approximation algorithm two octrees are constructed: one for the atoms in the molecule, and the other for the quadrature points. Born radii of all atoms are approximated by recursively traversing both octrees simultaneously. For simplicity, the octrees are drawn as quadtrees. This figure has been reused from [45] with permission.*

`nblists` run out of memory for molecules with millions of atoms. For large cutoffs, an octree is more space-efficient, update-efficient and cache-efficient compared to nblists [42].

Approximating Born Radii and GB Energy: This section gives a quick overview of the approximation algorithms for Born radii and polarization energy calculation described in [40]. We use the same basic ideas of near-far approximation in our distributed and distributed-shared-memory algorithms, although we change the algorithms as well as the approximation schemes for efficient work-division. Let A be the set of atoms in a molecule, and Q be the set of quadrature points (denoted *q-points*) sampled from the molecular surface. First, two octrees T_A and T_Q for A and Q, respectively, are built, and then Born radii are approximated by traversing them simultaneously starting at their root nodes.

Approximate integrals (using Equation 9.5) are collected at appropriate internal nodes of T_A and atoms of A. Suppose at some point during this traversal we are at node $A \in T_A$ and node $Q \in T_Q$. Let r_A (resp. r_Q) be the radius of A (resp. Q). If A and Q are far enough, i.e., the distance between their centers, r_{AQ} is larger than $(r_A + r_Q)\left(\frac{(1+\epsilon)^{1/6}+1}{(1+\epsilon)^{1/6}-1}\right)$ for some user-defined approximation parameter $\epsilon > 0$, then the contribution of all q-points in Q to the Born radius integral of each atom in A can be approximated by treating A (resp. Q) as a single pseudo-atom (resp. pseudo q-point) centered at the geometric center of the atoms (resp. q-points) under it. These approximated contributions are collected in A. If A and Q are not far enough but at least one of them is a non-leaf, we recurse using the children of the non-leaf/non-leaves. If both are leaves, then we compute the contributions exactly using the atoms under A and the q-points under Q, and collect them in the respective atoms. Next, we traverse T_A top-down, and collect and add partial integrals from all ancestors of an atom to it. Finally, we compute the Born radii values from these accumulated values [40]. E_{pol} is approximated using a similar technique. The pseudo-code for the Born radii and E_{pol} calculation can be found in [40]. Note that the accuracy and speedup of these algorithms can be tuned by changing the approximation parameters, ϵ; increasing ϵ gives better speedup while sacrificing accuracy in results more and vice-versa.

9.4 Related Work

Octree-based hierarchical treecode algorithms have already been used for energetics computations. These algorithms are typically based on Barnes-Hut clustering [109] or the Fast Multipole Method (FMM)[28], and have been implemented for both serial and distributed-memory parallel machines to compute Coulomb, London, Lennard-Jones, H-bonds potentials [65, 161], polarized Coulomb interactions [122], Yukawa potential [193], etc.

Popular Parallel E_{pol} Implementations The well-known *Amber* 12 [57] package has an MPI-based distributed-memory implementation for GB-energy calculation. Amber also has a shared-memory parallel implementation of GB-energy which uses vectorization [164]. *Gromacs* [102] has OpenMP based shared-memory and MPI based distributed-memory implementations of E_{pol}. On the other hand, *NAMD* [147, 174] uses Charm++ [110] and MPI for its shared and distributed-memory implementations, respectively. *Tinker-6.0* [72] is also a well-known MD package which supports OpenMP based shared-memory parallelism. On the other hand, GBr^6 has a serial approximation algorithm that uses volume-based r^6-approximation of Born radii as opposed to our surface-based r^6-approximation. We present the first efficient distributed-shared-memory implementation of GB energy computation. Although, most of these MD packages support multiple GB-models such as HCT [100], STILL [166], OBC [141] etc, we used STILL's [166] model of equations.

After we published this work, several improvements have been made in most of these MD packages. Amber, Gromacs, and NAMD have their GPU versions now. Gromacs GPU version still does not support implicit solvent GB energy computation. We show some comparison results with Amber-12 and 14 GPU versions at the end.

9.5 Our Contributions

Our main contributions in this work are as follows:

- We present efficient and scalable distributed-memory and hybrid distributed-shared memory parallel algorithms for approximating Born radii and polarization energy. A number of different load-balancing/work-distribution schemes have been explored.
- We show the theoretical time complexity analyses and detailed experimental performance analyses of the algorithms.
- We present performance comparison results of our distributed- and distributed-shared-memory parallel algorithms with five other state-of-the-art implementations of E_{pol}, namely, *Amber* 12, *Amber* 14 GBr^6, *Gromacs 4.5.3*, *NAMD 2.9* and *Tinker 6.0*, which show that our implementations outperform all of them.

The major difference of our approach from algorithms presented in [40] is that we only traverse one octree instead of two, and hence the approximation scheme is also different. Figures 9.2 and 9.3 show our modified algorithms.

APPROX-INTEGRALS(A, Q) (Here A denotes a node from atoms octree, and Q denotes a leaf node from quadrature points octree. For each atom a under the subtree rooted at the given node A in the atoms octree, this function approximates $\sum_{q \in Q} w_q \frac{(\mathbf{p}_q - \mathbf{p}_a) \cdot \mathbf{n}_q}{|\mathbf{p}_q - \mathbf{p}_a|^6}$. By $\mathbf{p}_a = \langle x_a, y_a, z_a \rangle$ we denote the center of an atom a, while by $\mathbf{p}_q = \langle x_q, y_q, z_q \rangle$, w_q and $\mathbf{n}_q = \langle nx_q, ny_q, nz_q \rangle$ we denote the location of a quadrature point q, weight assigned to q, and the unit outward normal on the molecular surface at q, respectively. By r_A (resp. r_Q) we denote the radius of the smallest ball that encloses all atom centers (resp. integration points) under A (resp. Q). The distance between the geometric centers of A and Q is given by $r_{A,Q}$. We assume $\widetilde{nx}_Q = \sum_{q \in Q} w_q nx_q$. Similarly, for $\widetilde{ny}_Q$ and $\widetilde{nz}_Q$. Each atom a has a field s_a, and each node A in the atoms octree has a field s_A, all of which are initialized to zero. The approximated sum is added to s_A provided A and Q are far enough in space so that the sum can be approximated reasonably well (controlled by an approximation parameter $\epsilon > 0$). Otherwise, the sums are computed recursively and added to the s field of appropriate descendants of A. By CHILD(A) we denote the set of non-empty octree nodes obtained by subdividing node A.)

1. ***if*** $r_{A,Q} - (r_A + r_Q) > 0 \;\wedge\; \frac{r_{A,Q} + (r_A + r_Q)}{r_{A,Q} - (r_A + r_Q)} > (1+\epsilon)^{\frac{1}{6}}$ ***then***
   $s_A = s_A + \frac{\widetilde{nx}_Q \cdot (x_Q - x_A) + \widetilde{ny}_Q \cdot (y_Q - y_A) + \widetilde{nz}_Q \cdot (z_Q - z_A)}{(r_{A,Q})^6}$ //far enough to approximate
2. ***elif*** LEAF(A) ***then*** //too close to approximate; compute exact value
   for each atom $a \in A$ ***do***
   for each quadrature point $q \in Q$ ***do***
   $s_a = s_a + \frac{w_q (nx_q \cdot (x_q - x_a) + ny_q \cdot (y_q - y_a) + nz_q \cdot (z_q - z_a))}{(r_{a,q})^6}$
3. ***else*** $\forall A' \in$ CHILD(A) : APPROX-INTEGRALS(A', Q)

PUSH-INTEGRALS-TO-ATOMS(A, s , s_{id}, e_{id}) (A is a node in the atoms octree, and $s = \sum_{A' \in \text{ANCESTORS(A)}} s_{A'}$. This function pushes $s + s_A$ to each descendant of A. If A is a leaf it computes the Born radius of each atom $a \in A$ using $s + s_A + s_a$. Here, s_{id} and e_{id} denote the start_id and end_id of the atoms assigned to a process.)

1. ***if*** LEAF(A) ***then*** $\forall a \in A$ that falls in $[s_{id}, e_{id}]$:
   $R_a = \max \left\{ r_a, \; \left(\frac{s_a + s + s_A}{4\pi} \right)^{-\frac{1}{3}} \right\}$ //compute Born radii of A's atoms
2. ***else*** $\forall A' \in$ CHILD(A) : **par** PUSH-INTEGRALS-TO-ATOMS($A', s + s_A, s_{id}, e_{id}$)//push integrals to A's descendants

FIGURE 9.2: *Octree-based algorithm for r^6-approximation of Born radii.*

APPROX-E_{pol}(U, V) (For two given nodes U and V in the atoms octree $\mathcal{T}_{\mathcal{A}}$ where, V is a leaf, approximate the part of E_{pol} resulting from the interaction between the set of atoms under U and V. By r_U we denote the radius of the smallest sphere that encloses all atom centers under U. For any atom $u \in U$, its center, radius, charge and Born radius are given by (x_u, y_u, z_u), r_u, q_u and R_u, respectively. For $0 \le k < M_\epsilon = \log_{1+\epsilon}(R_{max}/R_{min})$, $q_U[k] = \sum_{(u \in U) \,\wedge\, (R_u \in [R_{min}(1+\epsilon)^k, R_{min}(1+\epsilon)^{k+1}))} q_u$, where R_{min} and R_{max} are the minimum and the maximum Born radius among all atoms in $\mathcal{A}$. By CHILD(A) we denote the set of non-empty octree nodes obtained by subdividing node A.)

1. ***if*** LEAF(U) ***then return*** $-\frac{\tau}{2} \sum_{(u \in U) \,\wedge\, (v \in V)} q_u q_v \Big/ \sqrt{r_{uv}^2 + R_u R_v e^{-r_{uv}^2/4R_u R_v}}$ //exact value
2. ***elif*** $r_{U,V} > (r_U + r_V)\left(1 + \frac{2}{\epsilon}\right)$ ***then***
   return $-\frac{\tau}{2} \sum_{0 \le i,j < M_\epsilon} q_U[i] \cdot q_V[j] \Big/ \sqrt{r_{UV}^2 + R_{min}^2 (1+\epsilon)^{i+j} e^{-r_{UV}^2/4R_{min}^2(1+\epsilon)^{i+j}}}$ //approximate
3. ***else return*** $\sum_{U' \in \text{CHILD}(U)}$ APPROX-E_{pol}(U', V') //recurse on U **(parallel)**

FIGURE 9.3: *Octree-based algorithm for approximating E_{pol} from Born radii.*

9.5.1 Load Balancing

There are basically two possible ways of load-balancing in our distributed-shared- and distributed-memory algorithms:

- ▷ distribute only the work/computation (each process will have all the data),
- ▷ distribute both the data and work evenly among the processes (each process gets only a fraction of the data).

Here we only report the implementations in which we divide the work (each process has a complete set of data).

We use `MPI` [92] and `cilk++`[118] to implement our distributed and distributed-shared-memory algorithms. We choose `cilk++`because our algorithms are mainly based on nested fork-join parallelism, and such recursive parallel algorithms can be implemented very easily in `cilk++`. `Cilk++`'s randomized work-stealing scheduler allows efficient parallel execution of these recursive divide-and-conquer algorithms. In the rest of this chapter, we will refer to our hybrid distributed-shared implementation as $OCT_{MPI+CILK}$ and the pure distributed implementation as OCT_{MPI} .

Load-balancing on octree data structures has been discussed in [33]. We have used both static and dynamic load-balancing schemes in our algorithms. We use static load-balancing among the processes because static load-balancing is more efficient and less costly than dynamic load-balancing in this case. Our load-balancing scheme works in the following way:

- ▷ Explicit static load-balancing: Work is divided evenly among the processes. The ith process computes the Born radii and E_{pol} for the ith segment of atoms and leaf nodes, respectively, from the atoms octree.
- ▷ Implicit Dynamic load-balancing: Since the algorithm uses recursive divide-and-conquer technique and implemented using `cilk++`, the provably efficient work-stealing scheduler [27] of `cilk++` does dynamic load-balancing among the threads inside each process.

Different Work Distribution Approaches: In the distributed/distributed-shared-memory algorithms, one can distribute the work of calculating Born radii and polarization energy among the computing processes (a process consists of one or more cores), either by dividing the leaf nodes (Node-based-work-division[2]) or by dividing the atoms (Atom-based-work-division[3]).

Work Distribution for Born Radii calculation: For Born radii calculation work can be divided by dividing the atoms or nodes from any of the two octrees (atoms octree or quadrature points octree) evenly among the processes, assigning the job of computation on a particular segment of nodes or atoms to a particular process. To compute Born Radii, we distribute the work in two phases. Firstly, we evenly divide the leaf nodes from the quadrature points octree to the MPI processes. We assign the work of computing approximated integrals for the ith segment of leaf nodes to the ith MPI process. In the second phase (in Push-Integrals-to-Atoms), we divide the atoms evenly among the processes, and the ith MPI process computes the final Born Radii for the ith segment of the atoms. Note that each MPI process only traverses the atoms octree, and for each leaf node of the quadrature points octree that has been assigned to it, it computes the approximated integrals. In another implementation, we divide the atoms in both of these phases, and each process traverses both octrees (T_A and T_Q), but computes only for those nodes and atoms that fall within its range.

Work Distribution for E_{pol} calculation: For E_{pol} calculation, we first divide the leaf nodes of the atoms octree into P equal segments, where P is the number of MPI processes. Then we assign the work of computing the interaction of the ith segment of leaf nodes with the entire atoms octree to the ith MPI process. In this case, each process computes the interaction energy

[2] Each compute node computes only for the leaves assigned to it.
[3] Each compute node computes only for the atoms assigned to it.

due to all leaf nodes assigned to it, either by considering them in parallel (in $OCT_{MPI+CILK}$) or by taking them one at a time (in OCT_{MPI}) while it traverses the other atoms octree. We refer to the work division that divides leaf nodes for Born radii and energy computation as the *node–node work division*.

Other combinations of work divisions (e.g., *atom–node*, *atom–atom*, *qpoint–node*, *node–atom*, etc.) are also possible, but the *node–node* type work division scheme performed better than other alternatives in the experiments we conducted. We have observed that *atom–node* work division takes slightly more time than the purely node based (*node–node*) work division. Moreover, in *node–node* work division, only leaf nodes (of one octree) are considered during interaction computation (with another octree) which leads to less approximation compared to approximating at internal nodes. For this reason, the *node–node* work division performs better than others with respect to the percentage of error in the energy value. The error of atom based work division keeps changing with the number of processes even when the approximation parameters are kept fixed, because different division boundaries can split the same treenode differently in atom-based work division. On the contrary, for node-based work division, the error is constant for constant parameters, because each compute node always gets a full treenode, and hence the approximation does not change with the change of division boundaries. We have also observed the same trend of errors in Gromacs that also uses atom based work division techniques.

Dynamic load-balancing among threads: In our distributed-shared-memory algorithm, inside each compute node multiple threads (or cores) are used to accomplish the work assigned to a process. The `cilk++` runtime system provides dynamic load-balancing among threads using a randomized work-stealing scheduler [27]. In `cilk++` work-stealing scheduler, each thread maintains a double ended queue (deque) to store its outstanding work/tasks and adds the newly generated work to the bottom of the queue. On the other hand, when a thread runs out of work, it chooses a random victim thread and steals work from the top of the victim's queue (top task is ideally the biggest task) which helps to reduce inter-thread communication and guarantees progress [118].

9.5.2 Algorithm

Figure 1.4 shows a sketch of our hybrid distributed-shared-memory parallel octree based GB-radii and E_{pol} computation algorithms, where p denotes the number of threads running concurrently in shared-memory and is upper bounded by the number of cores in a single compute node. If the distributed-shared-memory algorithm runs with P processes, each running p threads internally, the corresponding distributed-memory algorithm should run $P \times p$ MPI processes to achieve the same level of parallelism (using the same number of cores).

It is important to design hybrid (distributed-shared) algorithms and explore their performance for the following reasons.

- ▷ Most modern supercomputers are networks of multicores, and hence, the future computation model is likely to be of a distributed-shared-memory type.

Distributed/Distributed-Shared-memory Octree Based GB-Polarization Energy Computation Algorithm (Suppose, we have P processes, each of which is running p threads internally. Therefore, if $p = 1$, it's a purely distributed approach and if $p > 1$, it's a distributed-shared approach. We first divide the work among the processes as evenly as possible. Inside each process (or node), work is further distributed among multiple threads dynamically by the `cilk++` framework.)

1. Each compute node builds atoms-octree, T_A and quadrature-points-octree, T_Q independently.
2. For $1 \leq i \leq P$ [*in parallel*], the ith process calculates the approximated integrals due to the ith segment of leaf nodes from the quadrature points octree by traversing T_A using the APPROX-INTEGRALS algorithm. //Node based work division.
3. Each process gathers the partial approximated integrals due to other segments of leaf nodes computed by other processes using MPI_Allreduce.
4. For $1 \leq i \leq P$ [*in parallel*], the ith process calls the PUSH-INTEGRALS-TO-ATOMS function and computes final Born radii for the ith segment of atoms. //Atom based work division.
5. Each process gathers the Born radii of other segments of atoms from other processes.
6. Each process traverses T_A, and for $1 \leq i \leq P$ [*in parallel*], the ith process calculates partial energy by computing the one-to-one interactions of the ith segment of leaf nodes from T_A on other nodes of T_A. //Node based work division.
7. The master process accumulates partial energy values from step 6 and generates the final E_{pol}.

FIGURE 9.4: *Octree-based distributed- and distributed-shared-memory algorithm.*

- ▷ A pure distributed-memory approach typically requires more memory than its distributed-shared-memory counterpart.[4]
- ▷ Running a single multi-threaded process with two threads on the same compute node (multicore machine) incurs less communication overhead than running two single-threaded processes on two different compute nodes.
- ▷ No distributed-shared-memory implementation of GB-energy is available yet (at the time of publication).

Suppose, in a shared-memory algorithm k threads share the same data of size s. Now if we launch these k threads as k different processes as in a distributed-memory setting, each process will require a separate copy of the same data occupying ks space in total. As long as this ks data fits in the shared-cache/main memory, the speedups from both distributed and distributed-shared memory approaches should be comparable. However, as k independent processes (distributed) use k times more memory than used by one process with k threads (shared), at some point, the distributed-shared-memory algorithm should outperform the distributed-memory algorithm. This happens when the input becomes so large that the ks data does not fit into the shared-cache/main memory and incurs severe memory overhead (page fault/cache misses and excessive pressure on bandwidth) causing a slowdown of the program. Moreover, *the typical cost of communication among k threads in shared-memory $<$ the cost of communication among k processes on a single compute node/socket $<$ the cost of communication among k processes on different sockets $<$ the cost of communication among computing nodes across the cluster.* This also implies that as we increase the number of processes, the overhead of purely distributed algorithm will be more than the distributed-shared-memory algorithm. We have also observed similar trends in our experiments.

[4] Distributed memory implementations are typically designed to replicate data instead of sharing.

9.5.3 Analysis of Time Complexity

In this section, we analyze the theoretical complexity of our distributed/distributed-shared-memory octree-based algorithms. We have used complexity results proved in [40] and [41] for this analysis. Let P be the number of MPI processes, and p be the number of threads running internally inside each process. Let, the molecule has M atoms in it.

Computational Cost, T_{comp}:

Step 1: Each process builds octrees from atoms and quadrature points, which takes $O(M \log M)$ time (assuming the number of Gaussian quadrature points, $m = O(M)$)[40]. Once the octrees have been built, we can approximate for any ϵ (recall that ϵ is an approximation parameter) without reconstructing them. Moreover, for drug-design and docking where we need to place the ligand at thousands of different positions w.r.t. the receptor, we can move the same octree to different positions or rotate it as needed by multiplying with proper transformation matrices, and then recompute the energy values. Therefore, we can consider the octree construction cost as a pre-processing cost and ignore it.

Step 2: Each process calculates the Born radii by traversing the atoms octree starting at the root node. The ith process computes only for the ith segment of leaf nodes from the quadrature points octree using the APPROX-INTEGRALS algorithm. Since each process gets approximately $\lceil M/P \rceil$ atoms, and inside each process each of the p cores/threads again does approximately $\frac{\lceil M/P \rceil}{p}$ part of the work, it costs $O\left(\left(\frac{1}{\epsilon^3}(\frac{M}{P}\frac{1}{p} + \log M\right)\right)$ time (using results from [41]).

Step 4: Each process calls PUSH-INTEGRALS-TO-ATOM, and the ith process calculates Born radii only for the ith segment of atoms. Traversing the entire tree takes $O(M \log M)$ time but each process traverses only that part of the tree that falls in its range. Eventually each thread traverses approximately $O\left(\frac{1}{P}(\frac{1}{p})\right)$ fraction of the tree. Therefore, this function will take $O\left(\frac{1}{P}(\frac{1}{p}(M \log M))\right)$ time.

Step 6: Each process traverses T_A, and the ith process calculates partial energy by computing the one-to-one interactions of the ith segment of leaf nodes from T_A with other nodes of T_A. Since each process gets $\lceil 1/P \rceil$ fraction of the total number of leaf nodes from the atoms-octree containing approximately $\lceil M/P \rceil$ atoms, each thread (or core) gets around $\frac{\lceil M/P \rceil}{p}$ of the atoms for computation. Hence, this step will take $O\left(\frac{1}{P}(\frac{1}{\epsilon^3}(\frac{M}{p} + 1) \log M)\right)$ time (using results from [41]).

Therefore, the total parallel computation time is, $T_{comp} = O\left(\frac{1}{P}\frac{1}{\epsilon^3}(\frac{M}{p} + 1) \log M\right)$.

Communication cost T_{comm}:

Step 3 & 5: Each process gathers the approximated integrals and Born radii of other segments from other processes. It takes $O(t_s \log P + t_w \frac{M}{P}(P-1))$ time, where t_s is the startup time and t_w is the message passing time per word (costs for MPI primitives can be found in Table 4.1 of [92]).

Step 7: The master process accumulates partial energy values from Step 6 using *MPI_Allreduce* and generates the final E_{pol} which takes $O(t_s \log P + t_w(P-1))$ time.

Therefore, the total parallel time, $T_p = T_{comp} + T_{comm} = O\left(\frac{1}{P}\frac{1}{\epsilon^3}(\frac{M}{p}+1)\log M + t_s \log P + t_w \frac{M}{P}(P-1)\right)$ $= O\left(\frac{1}{Pp}\frac{1}{\epsilon^3} M \log M + t_w M\right)$.

Attribute Name	Property
Processors	3.33 GHz-Hexa-Core 64-bit Intel-Westmere
Cores/node	12
RAM size and speed	24 GB, 1333 MHz
Cluster Interaction Type	InfiniBand, fat-tree topology, 40Gb/s p2p bandwidth
Cache	12 MB $L3$, 64 KB private $L1$, 256 KB private $L2$
Operating System	Linux CentOS 5.5.
Parallelism Platform	Intel Cilk-4.5.4, MPI (MVAPICH2/1.6)
Optimization parameter	-O3

TABLE 9.1: *Simulation environment.*

9.6 Simulation Results

All experiments included in this section were performed on the Lonestar4 computing cluster located at the Texas Advanced Computing Center [6]. All algorithms were tested on ZDock Benchmark Suite-2.0 containing 84 complexes (168 proteins) both in bound and unbound states. We used proteins from the bound dataset only. The number of atoms per protein varied from around 400 to $16,000$. Important properties of the simulation environment are summarized in Table 9.1.

We have compared three different octree based implementations, namely, the shared-memory, distributed-memory, and distributed-shared-memory implementations with GBr^6 [183], and the GB-polarization energy implementations from four existing well-known molecular dynamics packages, namely, *Gromacs* 4.5.3 [102], *NAMD* 2.9 [147, 174], *Amber* 12 [57] and *Tinker* 6.0 [72]. Table 9.2 summarizes some important properties of these programs. We have also reported the running times and energy values computed by the naïve serial implementations of Equations 9.2 and 9.5.

Package	GB-Model	Parallelism
Gromacs 4.5.3[102]	HCT[100]	Distributed (MPI)
NAMD 2.9[174]	OBC[141]	Distributed (MPI)
Amber 12[57]	HCT	Distributed (MPI)
Tinker 6.0[72]	STILL[166]	Shared (OpenMP)
GBr^6[183]	STILL	Serial

Name	GB-Model	Parallelism
OCT_{CILK}	STILL	Shared (`cilk++`)
OCT_{MPI}	STILL	Distributed (MPI)
$OCT_{MPI+CILK}$	STILL	Distributed (MPI+`cilk++`)
Naïve	STILL	Serial

TABLE 9.2: *Packages with GB models and types of parallelism used.*

9.6.1 Dealing with NUMA Effect

To reduce the impact of *NUMA* (Non-uniform memory architecture) on Intel machines, we ran all the MPI programs with *ibrun tacc_affinity*, which is basically a wrapper around the *mpirun* or *mpiexec*, and it fixes the affinity of the processes to the cores, sockets, and caches to reduce overall cache misses. On the other hand, `cilk++`does not provide any thread affinity manager. The `cilk++`work-stealing scheduler allows a thread to steal from any other thread. However, by stealing the oldest entry from a deque (least recently used data), it tries to reduce the number of cache misses. On Lonestar4, each machine was dual socket, and we launched one process with 6 threads on each socket for the $OCT_{MPI+CILK}$ program, which bounded those 6 threads only to one socket and alleviated the NUMA effect.

9.6.2 Scalability

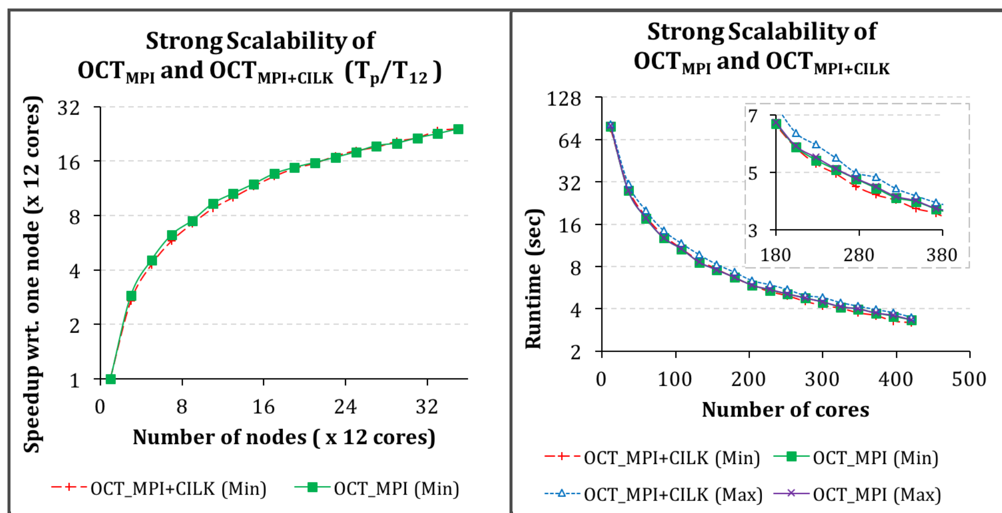


FIGURE 9.5: *Strong scalability with increasing number of cores.*

Figures 9.5 shows how the running time decreases and speedup increases, i.e., the scalability of our OCT_{MPI} and $OCT_{MPI+CILK}$ implementations with the number of cores. We ran this experiment on the Blue Tongue Virus (BTV) that has 6-millions of atoms and over 3-millions quadrature points. Since for a small number of cores (or processes), each core needs to handle a comparatively larger data segment, that segment may not fit in the cache fully at the same time leading to more cache misses. However, as the number of cores or processes increases, because of the balanced work division, each core will work only on a smaller portion of data which can easily fit into the cache.

For OCT_{MPI} program, we ran 12 processes in each compute node, and for $OCT_{MPI+CILK}$ program, we ran 2 processes each running 6 threads in each compute node. For each configuration, we ran all programs 20 times and plotted the minimum and maximum running times in the Figure 9.5. We observe that the minimum running time of $OCT_{MPI+CILK}$ is always smaller than the minimum running time of OCT_{MPI} after the core count reaches 180, whereas we always (independent of core count) see the opposite for the maximum running times. As the OCT_{MPI} program has 6 times more processes than $OCT_{MPI+CILK}$, the communication overhead of OCT_{MPI} was more than $OCT_{MPI+CILK}$. Similarly, the memory overhead was also more in OCT_{MPI}. For these reasons $OCT_{MPI+CILK}$ eventually ran faster than OCT_{MPI}. For BTV, when run on a single node with 12 cores, $OCT_{MPI+CILK}$ (2 processes, each with 6 threads) took approximately 1.4GB of memory, whereas $OCT_{MPI+CILK}$ (12 processes, each with 1 thread) occupied 8.2GB, which is 5.86 times more than that of $OCT_{MPI+CILK}$ (as expected). This ratio continues to hold as we increase the number of compute nodes.

9.6.3 Running Time and Speedup

Next we ran OCT_{MPI} and $OCT_{MPI+CILK}$ on a 12-core machine for the ZDock benchmark molecules, and compared their performance with that of OCT_{CILK}. Note that the algorithms underlying OCT_{MPI} and $OCT_{MPI+CILK}$ were different from the one used by OCT_{CILK}. All these algorithms were run with approximation parameters set to 0.9 (Born Radii) and 0.9 (E_{pol}),

respectively. We used approximate math for computing square root and power functions. No vectorization was used.

We observed that OCT_{CILK} showed better performance than both OCT_{MPI} and $OCT_{MPI+CILK}$ for molecules with less than 2500 atoms, since for small molecules the communication cost dominated the computation cost. The OCT_{MPI} implementation was significantly faster than OCT_{CILK} for molecules with greater than 2500 atoms, because for larger molecules computation costs beaten communication cost, and the differences in running times increased with the size of the molecules. The OCT_{MPI} implementation was also slightly faster than $OCT_{MPI+CILK}$ for molecules with less than 7500 atoms. After molecule size 7500, both OCT_{MPI} and $OCT_{MPI+CILK}$ showed similar performance. As OCT_{MPI} was using almost 6 times more memory than $OCT_{MPI+CILK}$, the difference in performance diminishes with the size of the molecule. MPI turns out to be more optimized compared to the `cilk++`implementation[5] and `cilk++`does not maintain thread affinity. There is an additional overhead of interfacing cilk++ and MPI. These overheads of $OCT_{MPI+CILK}$ were prominent for smaller molecules and became less dominant as the size of the molecule increased.

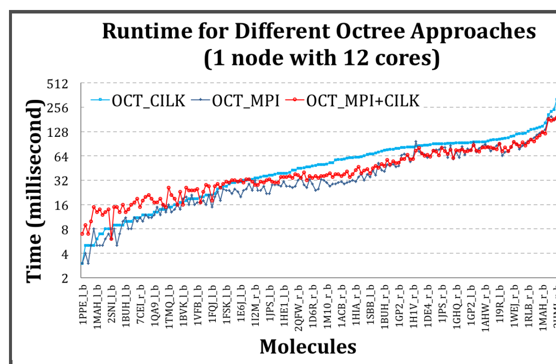
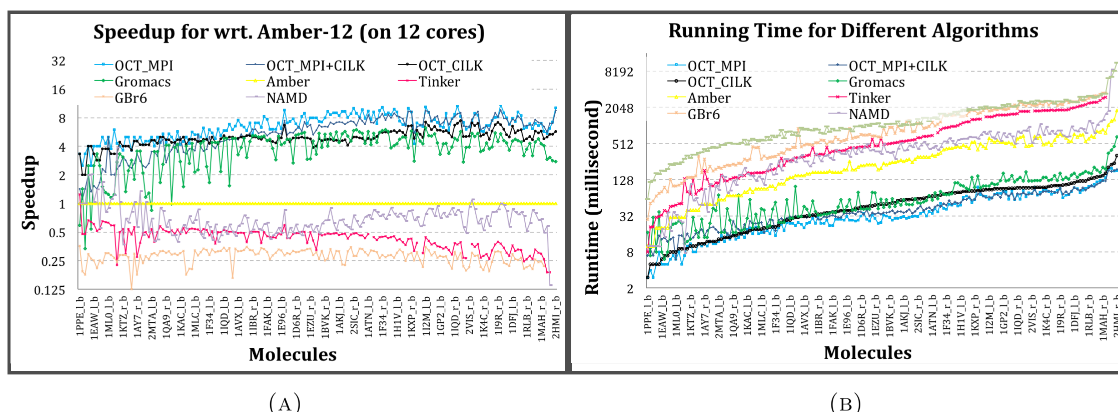


FIGURE 9.6: *Performance comparison of different octree based algorithms (results are sorted by the OCT_{CILK} time).*

Gromacs also has a shared-memory implementation of GB-energy, and we observed that for Gromacs, too, the distributed-memory implementation was slightly faster than the shared-memory implementation. Hence, in the rest of this section, we only compare with the MPI based distributed-memory implementation of Gromacs.



(A) (B)

FIGURE 9.7: *Performance comparison of different algorithms. Results are sorted by molecule size.*

For comparison purposes, we ran all programs mentioned in Table 9.2 on a 12-core machine (single compute node). For the distributed implementations (NAMD, Gromacs, Amber, OCT_{MPI}), we ran 12 different MPI processes on these 12 cores. For NAMD, we were not able to find any way

[5] We have used cilk-4.5.4, which is a predecessor of Intel cilk plus, and Intel cilk plus is likely to be much better optimized than cilk-4.5.4.

to compute only the GB-energy. So, we first computed the total electrostatic potential with GB energy turned on and then computed the electrostatic energy with GB energy turned off, and took the difference to retrieve actual GB energy. We also took the difference of running times of these two runs to get the time of GB energy computation. We took the average of 10 runs to reduce noise. Figure 9.7 shows the performance of different algorithms. From the plot of running times for GB-energy (including Born radii), we observe that overall OCT_{MPI} and $OCT_{MPI+CILK}$ perform the best among all algorithms. The differences in performance among Gromacs, OCT_{MPI} and $OCT_{MPI+CILK}$ become prominent as the size of the molecule increases. On the other hand, *Amber* was much slower than both OCT_{MPI} and Gromacs but faster than NAMD, Tinker, and GBr^6. Our results show that Tinker is slightly faster than GBr^6.

We can get a glimpse of the speedup achieved by these programs on 12 cores of one compute node (1 core for GBr^6) compared to *Amber* in Figure 9.7 (b) which shows that OCT_{MPI} achieves a speedup of approximately 11 w.r.t. *Amber* for a molecule of size 16,301 using only 12 cores, whereas Gromacs achieves a speedup of ~ 2.7 for the same molecule (although the maximum speedup achieved by Gromacs is 6.2 for a molecule with 2260 atoms). The maximum speedup achieved by *NAMD*, Tinker and GBr^6 for the ZDock benchmark molecules are 1.1, 2.1 and 1.14, respectively.

9.6.4 Energy Value

Figure 9.8 plots the GB-energy values for the ZDock benchmark molecules calculated by different algorithms mentioned in Table 9.2. The energy values computed by Amber, GBr^6, Gromacs, NAMD and OCT_{MPI} match closely with GB-energy computed by the naïve approach. Energy values reported by Tinker were around 70% of the naïve energy. All octree based algorithms reported approximately the same energy value. We have observed that Tinker and GBr^6 do not work for larger molecules ($> 12k$ and $> 13k$ respectively) as they run out of memory.

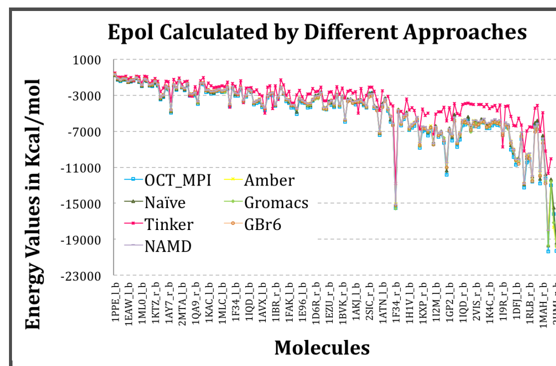


FIGURE 9.8: *Energy value computed by different algorithms.*

9.6.5 Change in Error and Runtime with Approximation Parameter

Recall that the octree-based algorithms are tunable because we can change the running time (as well as the error in result) by changing the approximation parameters. An increase in approximation parameter ϵ increases error in energy value and decreases running time. However, for small molecules, running times do not depend on ϵ at all. Figure 9.9 shows the impact of approximation parameter on our distributed-shared-memory algorithm's percentage of error in energy value and running time. The distributed-memory algorithm also follows the same trend. For this experiment, we kept the approximation parameter of Born Radii calculation fixed at 0.9 and varied the approximation parameter of E_{pol} from 0.1 to 0.9. We ran the $OCT_{MPI+CILK}$ implementation

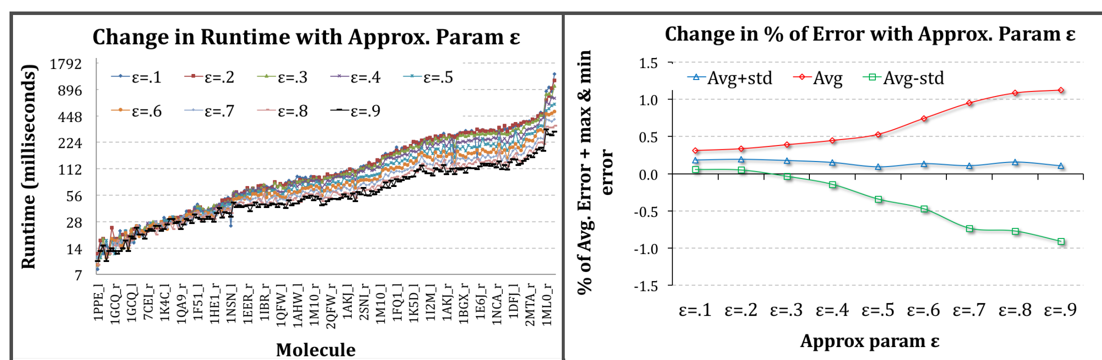
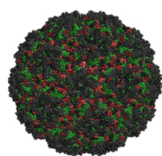


FIGURE 9.9: *Change in performance of the $OCT_{MPI+CILK}$ algorithm with approximation parameter, ϵ; Born Radius ϵ is fixed at 0.9 and E_{pol} ϵ varies.*

on all protein molecules of the ZDock benchmark suite. Approximate math was turned "off". Turning approximate math "on" shifted the error by $4-5\%$ and decreased the running times by a factor of 1.42 on average (Figure 9.6 vs. Figure 9.9). We collected the average and standard deviation of percentage of error for E_{pol}, and plotted the avg. ± std. for all molecules.

9.6.6 Scalability with Larger Molecule

We also ran all octree-based implementations and *Amber* on the Cucumber Mosaic Virus (CMV) shell consisting of 509,640 atoms and 1,929,128 quadrature points. GBr^6 and Tinker ran out of memory for CMV. We were able to run Gromacs and NAMD on CMV only for cutoff values up to 2 and 60, respectively, which are not reasonable cutoff values for such a large molecule. For CMV, OCT_{MPI} and $OCT_{MPI+CILK}$ achieved a speedup of more than $400-500$ using only 12 cores of a single compute node and $300-400$ times speedup using 144 cores (12 compute nodes each running 12-threads internally) w.r.t. Amber, while the errors w.r.t. the naïve energy were still less than 1% [6]. Note that we get such a high speedup because of three levels of acceleration: (a) from parallelism, (b) from two levels of approximations (numerical and algorithmic) in calculations (in Born Radii and E_{pol}), (c) from using the cache-friendly octree data structure and (d) using a recursive divide-and-conquer algorithm.



(A) *CMV Virus Shell.*

Program	12 Cores (Time)	144 Cores (Time)	Speedup wrt Amber using 12 Cores	Speedup wrt Amber using 144 Cores	Energy Value Kcal/Mol (10^6)	% of Difference with Naïve
OCT_{CILK}	12.5s	X	187	X	-1.48	-0.95
Amber	39min	3.3min	1	1	-1.44	2.2
$OCT_{MPI+CILK}$	4.8s	0.61s	488	325	-1.47	-0.07
OCT_{MPI}	4.5s	0.46s	520	430	-1.47	-0.07

(B) *Scalability on a large molecule (Cucumber Mosaic Virus shell).*

9.6.7 Comparison with Amber GPU Implementations

After we finished this work, a lot of improvements have been made in Amber, Gromacs, and NAMD packages. For example, all of these MD packages now have GPU based implementations which in general perform much better than their CPU-based implementations that we used for

[6] At present, Amber does not support concurrent execution of more than 256 cores.

comparison in this work. So we have redone some of the experiments to compare the performance of our algorithms with Amber GPU versions [57, 58, 89] and the result has been shown in Table 9.3. For these experiments, we use CMV as an input.

Amber 12 on Tesla M2090		Amber 12 on Kepler 20		Amber 14 on GTX780		Amber 14 on Kepler 80		OCT_MPI on 12-core Intel Westmere		OCT_MPI+CILK on 12-core Intel Westmere	
Energy	-1410965	Energy	-1410965	Energy	-1410965	Energy	-1410965	Energy	-1470540	Energy	-1470530
Time	91.79	Time	61	Time	46	Time	49	Time	5	Time	4.8
98% of Total Time	90	98% of Total Time	60	98% of Total Time	45	98% of Total Time	48	Speedup w.r.t fastest Amber	10	Speedup w.r.t fastest Amber	9.4

TABLE 9.3: *Comparison with Amber GPU implementations on CMV.*

We ran Amber 12 and Amber 14 on four different GPUs (Tesla, GTX and Kepler) with different compute capabilities. Since Amber GPU versions do not directly report the GB energy time, we consider 98% of the non-bonded energy time reported by Amber as the GB time as suggested by an Amber GPU developer [155]. Table 9.3 shows that our original OCT_{MPI} and $OCT_{MPI+CILK}$ implementations are still 10× faster than Amber GPU implementations. Amber's GPU implementations compute the forces, that we do not compute. This might be one reason of Amber's slow down.

9.6.8 Full Vs. Half Energy

In all our implementations we target to compute all $O(M^2)$ pairwise interactions (of course some of them were approximated for being far). However, since pairwise interactions for polarization energy is symmetric (i.e., the energy between x and y is the same as the energy between y and x), it is possible to compute only $\mathcal{O}\left(\frac{M^2}{2}\right)$ energy terms. Multiplying this half energy by 2 will give the desired full energy value. Computing full energy from the half energy directly would reduce the running time by half of what we have presented here so far. We conducted one experiment where we computed half of the energy terms, and Figure 9.11 shows the result. In this case, the OCT_{CILK} algorithm runs 16× faster than Amber, whereas in Figure 9.7 it ran around 6.3× faster for full energy computation. Note that computing half energy not only reduces the amount of computation but also the amount of data that need to be loaded from memory/cache. That might explain why the speedup is > 2×. This result shows that, it is possible to further improve all running times presented earlier in this chapter.

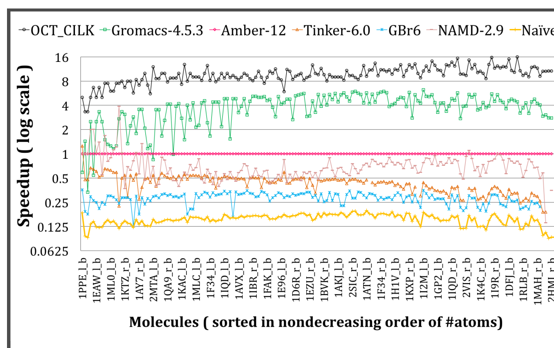


FIGURE 9.11: *Speedup w.r.t. Amber when only half of the energy terms are computed.*

9.7 Conclusion

In this chapter, we have presented a hybrid distributed-shared-memory parallel octree based algorithm for approximating molecular polarization energy, and provided detailed performance comparison with popular MD packages: Gromacs, NAMD, Amber, Tinker and GBr^6. We have shown that our octree based polarization energy approximation algorithms run significantly faster than *Amber*, Gromacs, NAMD, Tinker, and GBr^6, and can handle molecules with millions of atoms which cannot be handled by most of the other implementations. The presented octree-based algorithms also have very good scalability with the number of cores and molecule size. We believe that octree is the right data-structure to use in MD packages instead of nonbonded lists that cause most MD packages to run out of memory for very large molecules. A complete MD package based on octrees and the ideas presented in this chapter will accelerate MD simulation process undoubtedly. However, for that, we need to compute forces using octrees, which is an interesting research that needs to be done.

Our experience says that if a program runs significantly faster than another program while doing the same computation, it also consumes less energy than the later. Since our octree-based algorithms are significantly faster than other available MD packages, they should also consume less energy than those packages. Analyzing the energy and bandwidth profiles, cache-adaptivity in a multi-programming environment of these algorithms is interesting. The current distributed and distributed-shared memory algorithms are processor aware, making them processor oblivious while maintaining the performance is also interesting.

9.8 Acknowledgment

We thank Qin Zhang for his help in preparing the input files. We also thank Prof. Chandrajit L. Bajaj for his support and advice as always. We are thankful to Mark Abraham, Justin Lemkul and Szilárd Páll for their help with Gromacs, and to Michael Schnieders for helping with Tinker.

Chapter 10

Future Research

In this chapter, we discuss implications of our work to future parallel algorithm design, and ways to extend our research on other domains. We first discuss the scope of improvements and open problems related to each research work presented in the prior chapters. We end by discussing some open research problems.

Chapter 2: Dynamic programming on spatial architectures. Exploring the possibility of mapping other dynamic programming problems with non-local dependencies on the Triggered Instruction Spatial Architectures (TIA) is the next step in this research. Solving cache-oblivious wavefront (COW) algorithms [173] on TIA is an interesting research direction to pursue since both use the concept of a trigger: COW algorithms use the triggers in the scheduler, where triggered instruction spatial architecture implements them in hardware scheduler.

Chapter 3: CORDAC for solving dynamic programming problems. Apart from the research presented in this dissertation, we have also been working on automating the process of generating CORDAC algorithms from their corresponding serial iterative DP implementations by analyzing the data access patterns [14]. Having a full-fledged system that can take a serial iterative implementation of any DP problem and generate an efficient CORDAC algorithm while predicting the theoretical parallelism and cache-complexity, and then can generate an efficient implementation of the generated algorithm will immensely benefit the computational scientists (e.g., biologists, chemists and others) who occasionally deal with big scale dynamic programming problems and also need high-performance. Our work presented in Chapter 3 shows that optimization of the CORDAC algorithms in a systematic process. Building a specialized CORDAC compiler that can automatically optimize CORDAC algorithms is also an interesting research direction to pursue.

The shared-distributed-shared-memory algorithmic framework presented for CORDAC algorithms is processor-aware. Designing a processor-oblivious shared-distributed-shared-memory algorithmic framework for dynamic programming problems is challenging and interesting. Since in the near future, we are going to have machines with many more diverse architectural properties, the question that we ask is, “is it possible to design a framework that can generate and implement algorithms for cross-platform operativeness - e.g., algorithms that can work efficiently

on shared-memory, distributed-memory, distributed-shared memory and co-processor settings?". If the answer is "yes", building such a system will be a challenging research project.

Chapter 4: Robustness and adaptivity of CORDAC algorithms. There are many open questions regarding adaptivity and robustness that we still do not have answers to. We do not know how the cache-adaptivity changes based on overall parallelism, space usage and cache-complexity of a program. Other questions that we would like to answer in the future are as follows. Is there any predictable relationship among adaptivity, parallelism, space-usage, cache-efficiency and obliviousness (in both single and multiprogramming environments)? Are the cache-oblivious wavefront algorithms more/less adaptive than CORDAC algorithms? What are the relationships among energy consumption, cache-miss, bandwidth and running time? Can they be represented as simple equations that can portrait the asymptotic relations correctly? What about other parallel programming platforms, such as host + coprocessor setting?

Chapter 5: Provably efficient scheduling of cache-oblivious wavefront algorithms. There are a lot of opportunities to extend this work. It is highly likely that the complete-time, start and end timestamps for each cell and each recursive function call in a recursive wavefront algorithm can be automatically generated. If that is done, the next step would be to develop an *autowave* system that can automatically convert a standard 2-way CORDAC algorithm to a cache-oblivious wavefront algorithm while guaranteeing cache-optimality with improved parallelism.

Chapter 6: Cache-efficient Viterbi algorithm. The open problem here is to understand and find out whether is it possible to extend the ideas of rank convergence to solve other irregular DP problems such as the knapsack problem. Is it possible to improve the performance of the known recursive algorithm for LCS by using rank-convergence? It will be interesting to assess cache-adaptivity and bandwidth-performance of Viterbi algorithm. Extending Viterbi algorithm to run on manycores and distributed settings is also interesting.

Chapter 7 and 8: Optimistic parallelization and graph algorithms. It would be interesting to see if optimistic parallelization technique can be used to improve the performance of other nontrivial parallel applications that use dynamic load-balancing. Lockfree optimistic parallelization for approximation algorithms where some error in the result is acceptable is also an interesting research direction to pursue. Furthermore, even for the dynamic programming problems presented in this dissertation, where we need to take minimum or maximum from a range of values, use of optimistic parallelization may improve parallelism even further.

Chapter 9: Molecular Energetics. Since our octree based near-far approximation algorithms to compute polarization energy run orders of magnitude faster than other Molecular Dynamic (MD) packages, it is likely that other molecular energetics terms can also be computed faster by following similar octree-based approach. For other energetic terms, although shared-memory implementations are available, octree-based distributed and distributed-shared-memory algorithms still need to be implemented. A complete MD package based on octrees using similar ideas may accelerate MD simulation process significantly. However, for that to happen, we need to compute forces using octree-like data structures, which is another way to extend this work.

Open problem: Completing the automation pipeline. One of the major contribution of this dissertation is to show that recursive divide-and-conquer algorithms for solving dynamic programming problems are high-performing, and their performance is more robust than the other two popular options to solve them, namely the iterative and tiled-loop techniques. However, to motivate scientists to use these algorithms for their own unique dynamic programming problems, it is very important to complete the following pipeline.

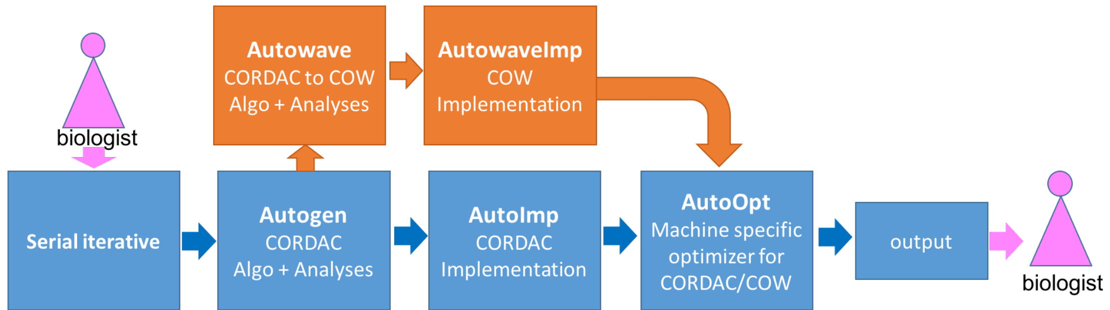


FIGURE 10.1: *Ideal pipeline for the automatic generation of efficient recursive algorithms and their implementations.*

At the beginning of the pipeline a biologist or a scientist who needs to solve a dynamic programming problem would write a simple iterative solution to her problem and feed that to *Autogen* [14], which would generate a recursive divide-and-conquer (CORDAC) algorithm along with the theoretical time and cache complexity from the given serial iterative implementation. Then the output of Autogen will be fed to *AutoImp*: an automatic implementer which can read the generated pseudocode and generate an actual executable implementation of the algorithm. The generated code will then be fed to *AutoOpt*: a machine specific optimizer which will automatically optimize the code so that it leads to better practical performance on a particular machine, with a given compiler and input size. At this point, if the biologist wants, she can take the optimized implementation and run her application. Otherwise, she can use the *Autowave* to generate a cache-oblivious wavefront algorithm to get even better parallelism. The generated recursive wavefront algorithm will be fed to an *AutoWaveImp* which will generate an implementation for a recursive wavefront algorithm of the original DP problem. Finally, the *AutoOpt* will be used to optimize the generated code for a given machine and input range which can then be used by the scientist for her computation.

To summarize, the aim of this dissertation was to present algorithms/algorithmic frameworks that solve many problems in bioinformatics more efficiently than their existing solutions on modern heterogeneous parallel architectures. However, our algorithms/algorithmic techniques are generic. Some of the algorithmic frameworks are also suitable for automation on modern multicore and manycore machines. The results are very promising and should encourage the rest of the research community to use our approaches, and extend them as needed.

Bibliography

[1] Cache Pirate. http://www.it.uu.se/research/group/uart/measurement/shared_resource_sensitivity/cache_pirate.

[2] Comet supercomputing cluster. http://www.sdsc.edu/support/user_guides/comet.html.

[3] Intel® Cilk™ Plus. http://www.cilkplus.org/.

[4] Papi-5.3. http://icl.cs.utk.edu/papi/index.html.

[5] Stampede supercomputing cluster. https://www.tacc.utexas.edu/stampede/.

[6] Texas Advanced Computing Center (TACC). https://www.tacc.utexas.edu/.

[7] Bioinformatics market by sector (molecular medicine, agriculture, forensic, animal, research and gene therapy), segment (sequencing platforms, knowledge management and data analysis) and application (genomics, proteomics and metabolomics) - global forecast to 2020. marketsandmarkets.com, report id: BT 3321, April, 2015.

[8] S. Aga, S. Krishnamoorthy, and S. Narayanasamy. Cilkspec: optimistic concurrency for cilk. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 83. ACM, 2015.

[9] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable graph exploration on multicore processors. *SC 10*, 2010.

[10] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, 2010.

[11] L. Akeila, O. Sinnen, and W. Humadi. Object oriented parallelization of graph algorithms using parallel iterator. *8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010), Brisbane, Australia,Conferences in Research and Practice in Information Technology (CRPIT).*, 107, 2010.

[12] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010. ISSN 0001-0782. doi: 10.1145/1735223.1735245. URL http://doi.acm.org/10.1145/1735223.1735245.

[13] M. Anderson. Better benchmarking for supercomputers. *IEEE communications of The ACM Spectrum*, 48:12–14, Jan 2011.

[14] C. Bachmeier, R. A. Chowdhury, P. Ganapathi, B. Kuszmaul, C. E. Leiserson, A. Solar-Lezama, Y. Tang, and J.J. Tithi. Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *PPoPP*, 2016.

[15] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. *ICPP*, pages 523–530, 2006.

[16] V. Bafna and N. Edwards. On de novo interpretation of tandem mass spectra for peptide identification. In *International conference on Research in computational molecular biology*, pages 9–18, 2003.

[17] N. Baker, M. Holst, and F. Wang. Adaptive multilevel finite element solution of the poisson-boltzmann equation ii. refinement at solvent-accessible surfaces in biomolecular systems. *J. Comput. Chem.*, pages 1343–1352, 2000.

[18] S. Beamer, K. Asanovic, and D.Patterson. Direction-optimizing breadth-first search. *SC 12*, November 2012.

[19] A. Beckmann and U. Meyer. Deterministic graph-clustering in external-memory with applications to breadth-first search. 2009.

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

[21] M. Bender, R. Ebrahimi, J. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Portland, OR, USA, January 2014.

[22] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[23] G. Blelloch and B. Maggs. A brief overview of parallel algorithms. *Carnegie Mellon University*.

[24] G. E. Blelloch. Prefix sums and their applications. 1990.

[25] G. E. Blelloch and M Reid-Miller. Pipelining with futures. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 249–259, 1997. ISBN 0-89791-890-8.

[26] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, pages 355–366, 2011.

[27] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, pages 46(5):720–748, 1999.

[28] J. Board, J. Causey, J. Leathrum, A. Windemuth, and K. Schulten. Accelerated molecular dynamics simulation with the parallel fast multipole algorithm. *Chemical Physics Letters.*, 198:89–94, 1992.

[29] 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.

[30] R. Brent. The parallel evaluation of general arithmetic expressions. *JACM*, pages 21:201–206, 1974.

[31] A. Buluc and K. Madduri. Parallel breadth-first search on distributed memory systems. *SC 11*, November 2011.

[32] C. Burge and S. Karlin. Prediction of complete gene structures in human genomic dna. *Journal of molecular biology*, 268(1):78–94, 1997.

[33] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and . J. D. T eresco. Dynamic Octree load balancing using space-filling curves. *Williams College Department of Computer Science Technical Report.*, 2003.

[34] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Bozeman, MT, USA, 1969.

[35] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, et al. Runnemede: An architecture for ubiquitous high-performance computing. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 198–209. IEEE, 2013.

[36] D. Cha, Q. Zhang, J.J. Tithi, A. Rand, R. A. Chowdhury, and C. Bajaj. Accelerated molecular mechanical and solvation energetics on multicore cpus and manycore gpus. In *Proceedings of the 6th ACM Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 222–231. ACM, 2015.

[37] C. Cherng and R. E. Ladner. Cache efficient simple dynamic programming. In *International Conference on Analysis of Algorithms DMTCS*, page 49:58, 2005.

[38] J. Chhugani, N. Satish, J. Sewall C. Kim, and P. Dubey. Fast and efficient graph traversal algorithm for cpus : Maximizing single-node efficiency. *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[39] R. 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.

[40] R. Chowdhury and C. Bajaj. Multi-level grid algorithms for faster molecular energetics. *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling.*, pages 147–152, 2010.

[41] R. Chowdhury and C. Bajaj. Multi-level grid algorithms for faster molecular energetics. *Journal Version under review.*, pages 147–152, 2010.

[42] R. Chowdhury, C. Bajaj D. Beglov, Y. Paschalidis, S. Vajda, P. Vakili, and D. Kozakov. Space-efficient maintenance of nonbonded lists for flexible molecules using dynamic octrees. *Journal Version under review.*, pages 147–152, 2011.

[43] R. Chowdhury, D. Beglov, M. Moghadasi, I. C. Paschalidis, P. Vakili, S. Vajda, C. Bajaj, and D. Kozakov. Efficient maintenance and update of nonbonded lists in macromolecular simulations. *Journal of chemical theory and computation*, 10(10):4449–4454, 2014.

[44] R. Chowdhury, P. Ganapathi, Y. Tang, and J.J. Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2016.

[45] R. A. Chowdhury and C. Bajaj. Algorithms for faster molecular energetics, forces, and interfaces. *ICES Report. The Insititute for Computational Engineering and Sciences, The University of Texas at Austin*, pages 10–32, 2010.

[46] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *SODA*, pages 591–600, 2006.

[47] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *In Proc. ACM SPAA*, pages 207–216. ACM, 2008.

[48] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, pages 47(4):878–919, 2010.

[49] R. A. Chowdhury, P. Ganapathi, V. Pradhan, J.J. Tithi, and Y. Xiao. A cache-efficient viterbi algorithm. In *under review*.

[50] R. A. Chowdhury, H-S. Le, and I. V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *Transactions on Computational Biology and Bioinformatics*, 7(3): 495–510, 2010.

[51] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, pages 73(7):911–925, 2013.

[52] R. Cledat and S. Pande. Discovering optimistic data-structure oriented parallelism. *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar'12)*, 2012.

[53] K. Compton and S. Hauck. Reconfigurable Computing: A Survey Of Systems and Software. *ACM Computer Survey*, 34(2):171–210, 2002. ISSN 0360-0300. doi: 10.1145/508352.508353.

[54] G. Cong and D. Bader. Lock-free parallel algorithms: An experimental study. In *In Proceedings of the 11th International Conference High Performance Computing*, pages 516–528. Springer, 2004.

[55] G. Cong and D. A. Bader. Designing irregular parallel algorithms with mutual exclusion and lock-free protocols. *J. Parallel Distrib. Comput.*, 66(6):854–866, 2006.

[56] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2009.

[57] D. A. Case et al. AMBER 12, 2012. University of California, San Francisco.

[58] D. A. Case et al. AMBER 14, 2014. University of California, San Francisco.

[59] L. Dagum and M. Ramesh. OpenMP: an industry standard api for shared-memory programming. *Computational Science & Engineering (CSE'98)*, pages 46–55, 1998.

[60] A. Darte, G. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, pages 7(4):379–392, 1997.

[61] T. A. Davis. University of florida sparse matrix collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[62] S. de O, E. Flavius, and A. C. de Melo. Retrieving smith-waterman alignments with optimizations for megabase biological sequences using gpu. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):1009–1021, 2013. ISSN 1045-9219. doi: 10.1109/TPDS.2012.194.

[63] J. Du, C. Yu, J. Sun, C. Sun, S. Tang, and Y. Yin. Easyhps: A multilevel hybrid parallel system for dynamic programming. In *IPDPSW*, pages 630–639, 2013.

[64] Z. Du, Z. Yin, and D. Bader. A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[65] Z. Duan and R. Krasny. An adaptive treecode for computing nonbonded potential energy in classical molecular systems. *Journal of Computational Chemistry.*, pages 184–195, 2001.

[66] D. Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *Int. J. Numer. Meth. Engng.*, pages 1129–1148, 1985.

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

[68] S. Dydel and P. Bała. Large scale protein sequence alignment using fpga reprogrammable logic devices. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 23–32. Springer, 2004.

[69] R. C. Edgar and K. Sjölander. Satchmo: sequence alignment and tree construction using hidden markov models. *Bioinformatics*, 19(11):1404–1411, 2003.

[70] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 165–175, Sept 2011.

[71] J. S. Emer, P. S. Ahuja, E. Borch, A. Klauser, C.K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. L. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, 2002.

[72] W. J. Ponder et al. Tinker molecular dynamics package. *"http://dasher.wustl.edu/ffe/"*, 2012.

[73] R. Farivar, H. Kharbanda, S. Venkataraman, and R. H. Campbell. An algorithm for fast edit distance computation on gpus. In *Proceeding of Innovative Parallel Computing*, pages 1–9, 2012.

[74] J. Feldman, I. Abou-Faycal, and M. Frigo. A fast maximum-likelihood decoder for convolutional codes. *Vehicular Technology, IEEE Transactions on*, pages 371–375, 2002.

[75] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[76] T. Flouri, C. Iliopoulos, M. S. Rahman, L Vagner, and M. Voráček. Indexing factors in dna/rna sequences. *Czech Science Foundation as project No. 201/06/1039.*, 99:13—15, June 2010.

[77] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[78] G. D. Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[79] G. C. Fox, S. W. Otto, and A. J.G. Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4(1):17–31, 1987.

[80] M. Frasca, K. Madduri, and P. Raghavan. Numa-aware graph mining techniques performance and energy efficiency. *SC 12*, November 2012.

[81] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[82] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, pages 79–90, Calgary, Canada, 2009. ACM.

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

[84] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than o(1) dependency. *J. of Parallel and Distributed Computing*, 21(2):213–222, 1994.

[85] R. Giegerich. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, 16(8):665–677, 2000.

[86] R. Giegerich and G. Sauthoff. Yield grammar analysis in the bellman's gap compiler. In *Workshop on language descriptions, tools and applications*, page 7, 2011.

[87] K. S. Gilhousen, R. Padovani, A. J. Viterbi, L. A. Weaver Jr., and C. E. Wheatley III. On the capacity of a cellular cdma system. *Vehicular Technology, IEEE Transactions on*, pages 303–312, 1991.

[88] M. Gilson, M. Davis, B. Luty, and J. A. McCammon. Computation of electrostatic forces on solvated molecules using the poisson-boltzmann equation. *J. Phys. Chem.*, pages 3591–3600, 1993.

[89] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. generalized Born. 8(5):1542–1555, 2012. doi: 10.1021/ct200909j. URL http://pubs.acs.org/doi/abs/10.1021/ct200909j.

[90] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing completion time for loop tiling with computation and communication overlapping. In *IPDPS*, pages 10–pp, 2001.

[91] V. Govindaraju, C. Ho, and K. Sankaralingam. Dynamically Specialized Datapaths for Energy Efficient Computing. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011.

[92] A. Grama, A. Gupta, G. Karypis, and V. kumar. *Introduction to Parallel Computing (2nd Edition).* 2008.

[93] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal Of Computational Physics.*, page 280–292, 1997.

[94] J. A. Grice, R. Hughey, and D. Speck. Parallel sequence alignment in limited space. In *Proceedings of Intelligent Systems for Molecular Biology*, pages 145–153, 1995. ISBN 0-929280-83-0.

[95] T. Grosser, A. Groesslinger, and C. Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[96] T. Grycuk. Deficiency of the coulomb-field approximation in the generalized born model: An improved formula for born radii evaluation. *J. Chem. Phys.*, pages 4817–4826, 2003.

[97] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[98] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[99] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, 1997.

[100] G. D. Hawkins, C. J. Cramer, and D. G. Truhlar. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges froma dielectric medium. *J. Phys. Chem.*, pages 19824—-19839, 1996.

[101] A. Heilper and D. Markman. Vectorization of sequence alignment computation using distance matrix reshaping. (US Patent 7343249), 2008.

[102] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.*, pages 435–447, 2008.

[103] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. ISSN 0001-0782. doi: 10.1145/360825.360861.

[104] M. Holst, N. Baker, and F. Wang. Adaptive multilevel finite element solution of the poisson-boltzmann equation i. algorithms and examples. *J. Comput. Chem.*, pages 1319–1342, 2000.

[105] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multicore cpu and gpu. *Parallel Architectures and Compilation Techniques(PACT).*, 99:100–113, 2011.

[106] X. Huang, W. Miller, S. Schwartz, and R. C. Hardison. Parallelization of a local similarity algorithm. *Computer applications in the biosciences: CABIOS*, 8(2):155–165, 1992.

[107] C. L. Jackins and S. L. Tanimoto. Octrees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, 1980.

[108] C. L. Janssen and I. M. Nielsen. *Parallel computing in quantum chemistry*. CRC Press, 2008.

[109] I. Jeffrey and V. Okhmatovskil. Effect of multilayered substrate on the barnes-hut center-of-charge clustering approximation: Half-space case study. *SPI IEEE*, 2008.

[110] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[111] I. Kand and A. N. Willson. Low-power viterbi decoder for cdma mobile terminals. *Solid-State Circuits, IEEE Journal of*, pages 473–482, 1998.

[112] J. Karlander. Algorithms and complexity (exercise 3+4), 2013. URL http://www.csc.kth.se/utbildning/kth/kurser/DD2352/algokomp13/.

[113] K. B. Kent, R. B. Proudfoot, and Y. Zhao. Parameter-specific fpga implementation of edit-distance calculation. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pages 209–215, 2006.

[114] M. Korpar and M. Sikic. Sw# - gpu enabled exact alignments on genome scale. *Bioinformatics*, 29(19):2494–2495, 2013.

[115] A. Krogh, B. Larsson, G. Von Heijne, and E. L.L. Sonnhammer. Predicting transmembrane protein topology with a hidden markov model: application to complete genomes. *Journal of molecular biology*, 305(3):567–580, 2001.

[116] M. Kulkarni, B. Walter K. Pingali, G. Ramanarayanan, K. Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. *Communications of The acm.*, 25(9), 2009.

[117] I.-T A Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 140–151, 2013.

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

[119] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). *Parallel Architectures and Compilation Techniques(PACT).*, 99:100–113, 2011.

[120] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to algorithms.* The MIT press, 2001.

[121] A. Lew and H. Mauch. *Dynamic programming: A computational tool*, volume 38. Springer, 2006.

[122] P. Li, H. Johnston, and R. Krasny. A cartesian treecode for screened coulomb interactions. *Journal of Computational Physics.*, pages 3858–3868, 2009.

[123] W. Liu and B. Schmidt. A generic parallel pattern-based system for bioinformatics. In *Euro-Par*, pages 989–996, 2004.

[124] B. Lu, D. Zhang, and J. A. McCammon. Computation of electrostatic forces between solvated molecules determined by the poisson-boltzmann equation using a boundary element method. *J. Comput. Chem.*, pages 214102–214109, 2005.

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

[126] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 219–232. ACM, 2014.

[127] R. Mao. *Distance-based Indexing and Its Applications in Bioinformatics.* 2007.

[128] A. Marquardt, V. Betz, and J. Rose. Speed and Area Tradeoffs in Cluster-Based FPGA Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1): 84 –93, Feb. 2000. ISSN 1063-8210. doi: 10.1109/92.820764.

[129] D. C. McShan, S. Rao, and I. Shah. Pathminer: predicting metabolic pathways by heuristic search. *Bioinformatics.*, pages 1692–8, 2003.

[130] B. Mei, S. Vernalde, D. Verkest, H. Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 61–70, 2003.

[131] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. *PPoPP 12*, February 2012.

[132] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 157–166, 1996.

[133] A. Mislove, M. Marcon, P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. *Internet Measurement Conf.*, 48:29–42, Jan 2007.

[134] J. B. O. Mitchell. Multipole-based calculation of the polarization energy. *Theor Chim Acta.*, 1996.

[135] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge Press, 2005.

[136] G. Morrisett and M. Herlihy. Optimistic parallelization. Technical report, 1993.

[137] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[138] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746.

[139] P. Norvig. Teach yourself programming in ten years.

[140] U. Ohler, H. Niemann, G. Liao, and G. M. Rubin. Joint modeling of dna sequence and physical properties to improve eukaryotic promoter recognition. *Bioinformatics*, 17(suppl 1):S199–S206, 2001.

[141] A. Onufriev, D. Bashford, and D. A. Case. Exploring protein native states and large-scale conformational changes with a modified generalized born model. *PROTEINS: Structure, Function, and Bioinformatics.*, page 383–394, 2004.

[142] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *Computers, IEEE Transactions on*, pages 48(2):142–149, 1999.

[143] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions: a control paradigm for spatially-programmed architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 142–153, 2013. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485935.

[144] Y. Park, S. Shackney, and R. Schwartz. Network-based inference of cancer progression from microarray data. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 6(2):200–212, 2009.

[145] J. S. Pedersen and J. Hein. Gene finding with a hidden markov model of genome structure and evolution. *Bioinformatics*, 19(2):219–227, 2003.

[146] R. J. Petrella, I. Andricioaei, B. R. Brooks, and M. Karplus. An improved method for nonbonded list generation: Rapid determination of near-neighbor pairs. *Journal of Computational Chemistry.*, pages 222—231, January 2003.

[147] J. C. Phillips, R. Braun, J. Gumbart W. Wang, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with namd. *Journal of Computational Chemistry.*, pages 1781–1802, 2005.

[148] 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 *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, New Orleans, LA, 2010. IEEE.

[149] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. In *ACM SIGPLAN Notices*, pages 46(10):83–98, 2011.

[150] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman. A run-time reconfigurable system for gene-sequence searching. In *Proceedings of the 16th Annual VLSI*, pages 561–566, 2003. doi: 10.1109/ICVD.2003.1183193.

[151] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[152] A. Radenski. Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps, 2011.

[153] R. Reitzig. Automated parallelisation of dynamic programming recursions. 2012.

[154] L. Renganarayanan, D. Kim, M. M. Strout, and S. Rajopadhye. Parameterized loop tiling. *TOPLAS*, page 34(1):3, 2012.

[155] W. Ross, 2014. Personal communication.

[156] B. Sahoo, T. Swarnkar, and S. Padhy. Implementation of parallel edit distance algorithm for protein sequences using reconfigurable accelerator. In *Proceedings of the International Conference on Advances in Computing, Communication and Control*, pages 26–29, 2009. ISBN 978-1-60558-351-8. doi: 10.1145/1523103.1523109.

[157] V. Sarkar and N. Megiddo. An analytical model for loop tiling and its solution. In *ISPASS*, pages 146–153, 2000.

[158] E. Saule and U. V. Catalyurek. An early evaluation of the scalability of graph algorithms on the intel mic architecture. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.

[159] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 63–66, May 2002.

[160] A. Siepel, G. Bejerano, J. S. Pedersen, A.S. Hinrichs, M. Hou, K. Rosenbloom, H. Clawson, J. Spieth, L. W. Hillier, S. Richards, et al. Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes. *Genome research*, 15(8):1034–1050, 2005.

[161] T. Simonson and A. Bruenger. Solvation free energies estimated from macroscopic continuum theory: An accuracy assessment. *J. Phys. Chem.*, pages 4683–4694, 1994.

[162] M. Sniedovich. *Dynamic programming: Foundations and principles.* CRC press, 2010.

[163] E-F. Soong, F. K .and Huang. A tree-trellis based fast search for finding the n-best sentence hypotheses in continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 705–708. IEEE, 1991.

[164] C. P. Sosa, T. Hewitt, M. R. Lee, and D. A. Case. Vectorization of the generalized born model for molecular dynamics on shared-memory computers. *Journal of Molecular Structure: THEOCHEM*, 549(1):193–201, 2001.

[165] M. Spann and R. Wilson. A quad-tree approach to image segmentation which combines statistical and spatial information. *Pattern Recognition*, 18(3):257–269, 1985.

[166] W. C Still, A. Tempczyk, R. C. Hawley, and T. Hendrickson. Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am.Chem.Soc.*, pages 6127–6129, 1990.

[167] B. Su, K. Keutzer, and Tasneem. G. Brutch. Parallel bfs graph traversal on images using structured grid. *IEEE 17th International Conference on Image Processing.*, 2010.

[168] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, K. Putnam, A.and Michelson, M. Oskin, and S. J. Eggers. The WaveScalar Architecture. *ACM Transactions on Computer Systems*, 25(2):4:1–4:54, May 2007. ISSN 0734-2071. doi: 10.1145/1233307.1233308.

[169] G. Tan, S. Feng, and N. Sun. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In *SC*, page 78. ACM, 2006.

[170] S. Tang, C. Yu, J. Sun, B. Lee, T. Zhang, Z. Xu, and H. Wu. Easypdp: An efficient parallel dynamic programming runtime system for computational biology. *TPDS*, pages 23(5):862–872, 2012.

[171] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *SPAA*, pages 117–128, 2011.

[172] Y. Tang, R. You, H. Kan, J.J. Tithi, P. Ganapathi, and R. A. Chowdhury. Improving parallelism of recursive stencil computations without sacrificing cache performance. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, pages 1–7. ACM, 2014.

[173] Y. Tang, R. You, H. Kan, J.J. Tithi, P. Ganapathi, and R. A. Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *PPoPP*, pages 205–214, 2015.

[174] D. E. Tanner, K. Chan, J. C. Phillips, and K. Schulten. Parallel generalized Born implicit solvent calculations with namd. *J. Chem. Theory Comput.*, pages 3635—-3642, 2011.

[175] J. Thiyagalingam, O. Beckmann, and P. H. Kelly. Minimizing associativity conflicts in morton layout. In *Parallel Processing and Applied Mathematics*, pages 1082–1088. Springer, 2006.

[176] J.J. Tithi and R. A. Chowdhury. Poster: Polarization energy on a cluster of multicores. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1379–1379. IEEE, 2012.

[177] J.J. Tithi and R. A. Chowdhury. Polarization energy on a cluster of multicores. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 569–578. IEEE, 2013.

[178] J.J. Tithi, D. Matani, G. Menghani, and R. A. Chowdhury. Avoiding locks and atomic instructions in shared-memory parallel bfs using optimistic parallelization. *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1628–1637, 2013.

[179] J.J. Tithi, N. C. Crago, and J. S. Emer. Exploiting spatial architectures for edit distance algorithms. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 23–34. IEEE, 2014.

[180] J.J. Tithi, P. Ganapathi, A. Talati, and R. A. Chowdhury. High-performance recursive dynamic programming for bioinformatics using mm-like flexible kernels. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 600–601. ACM, 2014.

[181] J.J. Tithi, Y. Fogel, and R. A. Chowdhury. Two novel shared-memory parallel bfs algorithms and their performance on multicores and manycores. 2015.

[182] J.J. Tithi, P. Ganapathi, A. Talati, S. Aagarwal, and R. Chowdhury. High-performance energy-efficient recursive dynamic programming using matrix-multiplication-like flexible kernels. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2015.

[183] H. Tjong and H. X. Zhou. GBr6: A parameterization-free, accurate, analytical generalized Born method. *J. Phys. Chem. B.*, pages 3055–3061, 2007.

[184] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 207–216, 2010. doi: 10.1109/ICPPW.2010.38.

[185] G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Bddt: Block-level dynamic dependence analysis for task-based parallelism. In *Advanced Parallel Processing Technologies*, pages 17–31. 2013.

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

[187] A. J. Viterbi. Convolutional codes and their performance in communication systems. *Communication Technology, IEEE Transactions on*, 19(5):751–772, 1971.

[188] S. Warshall. A theorem on boolean matrices. *J. of the ACM (JACM)*, 9(1):11–12, 1962.

[189] M. Waterman. *Introduction to computational biology: maps, sequences and genomes.* Chapman & Hall Ltd, 1995.

[190] Weisstein and Eric W. Gaussian quadrature, from mathworld–a wolfram web resource. {http://mathworld.wolfram.com/GaussianQuadrature.html}"., pages 454–468, 1991.

[191] M. E. Wolf. *Improving locality and parallelism in nested loops.* PhD thesis, Citeseer, 1992.

[192] Y. Xu, R. J. Mural, and E. C. Uberbacher. Constructing gene models from accurately predicted exons: an application of dynamic programming. *Computer applications in the biosciences: CABIOS*, 10(6):613–623, 1994.

[193] Z. Xu. Treecode algorithm for pairwise electrostatic interactions with solvent-solute polarization. *Physical Review E.*, 2010.

[194] Y. Yand, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. CatalyÄurek. A scalable distributed parallel breadth-first search algorithm onbluegene/l. *SC.*, 2005.

[195] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA*, pages 93–104. ACM, 2007.

[196] K. Zhang, Z. Qin, T. Chen, J. S. Liu, M. S. Waterman, and F. Sun. Hapblock: haplotype block partitioning and tag snp selection software using a set of dynamic programming algorithms. *Bioinformatics*, 21(1):131–134, 2005.