Improving Parallelism of Recursive Stencil Computations without Sacrificing Cache Performance [*]

Yuan Tang Ronghui You
Haibin Kan
Software School, School of Computer Science
Fudan University
Shanghai, P. R. China
[yuantang, 11300720164, hbkan]@fudan.edu.cn

Jesmin Jahan Tithi Pramod Ganapathi
Rezaul A. Chowdhury
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790, USA
[jtithi, pganapathi, rezaul]@cs.stonybrook.edu

Abstract

The state-of-the-art "trapezoidal decomposition algorithm" for stencil computations on modern multicore machines use recursive divide-and-conquer (DAC) to achieve asymptotically optimal cache complexity cache-obliviously. But the same DAC approach restricts parallelism by introducing artificial dependencies among subtasks in addition to those arising from the defining stencil equations. As a result, the trapezoidal decomposition algorithm has suboptimal parallelism.

In this paper we present a variant of the parallel trapezoidal decomposition algorithm called "cache-oblivious wavefront" (COW) that starts execution of recursive subtasks earlier than the start time prescribed by the original algorithm without violating any real dependencies implied by the underlying recurrences, and thus reducing serialization due to artificial dependencies. The reduction in serialization leads to an improvement in parallelism. Moreover, since we do not change the DAC-based decomposition of tasks used in the original algorithm, cache performance does not suffer.

We provide experimental measurements of absolute running times, burdened span by Cilkview, and L1/L2 cache misses by PAPI to validate our claims.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; G.4 [*Mathematical Software*]: Algorithm design and analysis.; D.2.8 [*Software Engineering*]: Metrics—complexity measures, performance measures

General Terms Algorithms, Performance

Keywords parallel cache-oblivious algorithm; stencil; cache-oblivious wavefront; atomic operation; multi-core; nested parallel computation; Cilk

[*] Rezaul A. Chowdhury and Pramod Ganapathi were supported in part by NSF grant CCF-1162196.

WOSC '14, October 20, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2308-6/14/10…$15.00.
http://dx.doi.org/10.1145/2686745.2686752

1. Introduction

A stencil defines how to compute the value of a grid point in a spatial grid at any given time step t as a function of values at neighboring grid points at recent time steps before t. A stencil computation [3, 13–16, 18, 23–29, 33] performs such computations for every grid point over many time steps. Frigo and Strumpen [16] presented a serial cache-oblivious algorithm[1] for stencil computation based on a recursive divide-and-conquer approach called "trapezoidal decomposition", and then extended it to a multithreaded algorithm with optimal serial cache performance [17, 18]. The parallelism of the multithreaded trapezoidal decomposition algorithm was further improved in [31] without losing cache efficiency by performing a hyper-space cut.

The recursive divide-and-conquer (DAC) approach used in the trapezoidal decomposition algorithm allows it to achieve asymptotically optimal serial cache performance through increased "temporal locality"[2] without using any knowledge of the parameters of the cache hierarchy. However, for such DAC algorithms scheduling decisions are made at the level of subtasks that are only a constant factor smaller than the parent task which introduces artificial dependencies among recursive subtasks in addition to those arising from the defining stencil equations. As a result, though the trapezoidal decomposition algorithm has optimal serial cache complexity, the parallelism is suboptimal.

[1] A cache-oblivious algorithm is designed without mention of any cache parameters (e.g., cache size and cache line size) in the code [20].

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

For good parallel performance under state-of-the-art schedulers on modern multicore machines, an algorithm must show good cache performance on a serial machine and also have high parallelism. For example, the "randomized work-stealing scheduler" [1, 5, 19] for distributed caches guarantees the following bounds w.h.p.[3]: $T_p = O(T_1/p + T_\infty)$ and $Q_p = Q_1 + O(p(M/B)T_\infty)$, where, p is the number of processing cores, T_p and Q_p are the running time and the cache complexity of the algorithm, respectively, on p processing cores, M is the cache size, and B is the cache line size. The "parallel depth-first scheduler" [4] for shared caches, on the other hand, guarantees that $Q_p \leq Q_1$ provided $M_p \geq M_1 + \Theta(pT_\infty)$, where M_p is the size of the cache shared by p cores, and M_1 is the size of the cache on a serial machine. In both cases, the lower the value of Q_1 (i.e., serial cache complexity) and T_∞ (which is called the "span"), the better the parallel performance. Since "parallelism" is defined as $\frac{T_1}{T_\infty}$, a lower span implies a higher parallelism provided T_1 remains fixed.

With the increase of core count on multicores, and the emergence of manycore processors with cache hierarchies (e.g., "Intel MIC" [22]), the need for algorithms with good cache performance and high parallelism continues to grow.

Recursive parallel DAC-based algorithms typically divide the input task into a small number (upper bounded by a constant) of smaller subtasks at each level of recursion, and solve them recursively. The order of execution of these subtasks are decided based on the dependency relationships at the granularity of the subtasks themselves. If a subtask u depends only on a very small portion of another subtask v, this approach fails to start the execution of u as soon as the dependency relations are resolved even if there are idle processing cores. Thus artificial dependency relations imposed by the structure of the decomposition prevents many completely independent subtasks from executing in parallel which leads to lower parallelism. However, one can reduce the number of artificial dependencies and thus improve parallelism by dividing the input task into more subtasks of even smaller sizes at each level of recursion. Unfortunately, the faster the rate at which task sizes decrease during recursive calls, the worse it is for Q_1 because if task sizes decrease fast the largest subproblems that fit into the cache is often much smaller than the size of the cache leading to cache underutilization and loss of temporal locality. In the light of the discussion above, traditional wisdom may suggest that one should find a balance point between parallelism and cache complexity in order to get good performance in practice.

Contrary to conventional wisdom, in this paper we show that one can achieve optimal Q_1 without trading off parallelism in recursive DAC-based algorithms. Indeed, we present a mechanism to execute the recursive subtasks generated by trapezoidal decomposition algorithm in such a way that the execution order does not violate any real dependencies, but reduces stallings due to artificial dependencies. Since we do not change the recursive divide-and-conquer structure of the original algorithm, the algorithm remains cache-oblivious and Q_1 remains optimal. Our technique works by propagating a wavefront of executing and ready-to-be-executed subtasks (satisfying all real dependency constraints) through a dynamically unfolding recursive divide-and-conquer tree which is essentially different from state-of-the-art parallel task graph execution systems [2, 32] that usually unroll the entire execution beforehand and execute all subtasks in a parallel looping fashion.

Our Contributions. Our major contributions are as follows.

- **[Algorithmic Contribution]** We present a variant of the parallel trapezoidal decomposition algorithm which performs divide-and-conquer of the input task in exactly the same way as the original cache-optimal recursive DAC-based algorithm, but considers a recursive subtask ready for execution as soon as all its real dependency constraints are satisfied. Artificial dependency constraints introduced by the DAC structure no longer prevent independent subtasks from executing in parallel, and thus leading to potentially improved parallelism. The new algorithm works by propagating a wavefront of executing and ready-to-be-executed subtasks through a dynamically unfolding recursive DAC tree. We call our new algorithm the "Cache-Oblivious Wavefront" (COW) based trapezoidal decomposition algorithm.
- **[Experimental Results]** We have implemented the new trapezoidal decomposition variant and compared it with the standard parallel trapezoidal decomposition algorithm based on recursive DAC as well as parallel looping implementations with and without tiling on 16-core and 32-core machines. Experimental measurements of absolute running times, relative speedups w.r.t. direct parallel loops, burdened span by Cilkview [21], and L1/L2 cache misses by PAPI [6] validate our claims.

Organization of the Paper. The rest of the paper is organized as follows. We explain our new algorithm in Section 2. Experimental results are presented in Section 3, and a survey of related work in Section 4. Finally, we include some concluding remarks in Section 5.

2. Improving Parallelism of Cache-optimal Stencil Computations

This section shows how to apply the cache-oblivious wavefront technique on the cache-oblivious trapezoidal decomposition algorithm [16–18, 31] to improve its parallelism without any asymptotic loss in cache performance.

[3] For an input of size n, an event E occurs w.h.p. (with high probability) if, for any $\alpha \geq 1$ and c independent of n, $Pr(E) \geq 1 - \frac{c}{n^\alpha}$. The larger the value of n, the closer $Pr(E)$ is to 1, and $\lim_{n\to\infty} Pr(E) = 1$.

The COW algorithm tries to execute a subtask as soon as all its real dependency constraints are satisfied while still following the same recursive divide-and-conquer scheme of a cache-optimal DAC algorithm. For example, suppose X and Y are two tasks in such an algorithm and subtask X_i of X has a real dependency on subtask Y_j of Y, and there are no other real dependencies between X and Y. Let's assume for simplicity that each subtask of X and Y takes only $O(1)$ time to execute. The standard DAC algorithm will not let X execute until all subtasks of Y completes execution though Y_j has perhaps completed execution much earlier and there was no need for X to delay execution. In our approach, task X is spawned at the same time as Y. Instead of the task X waiting for the completion of the entire task Y, subtasks of X are spawned in parallel with those of Y. Subtask X_i will be busy waiting for the completion of Y_j by continuously checking the wavefront data structure. When Y_j completes execution it updates the wavefront and within constant time X_i (and thus X) starts its execution. This approach works efficiently provided the structure of computation guarantees that real dependency constraints of X are satisfied within $O(1)$ time of the start of execution of Y.

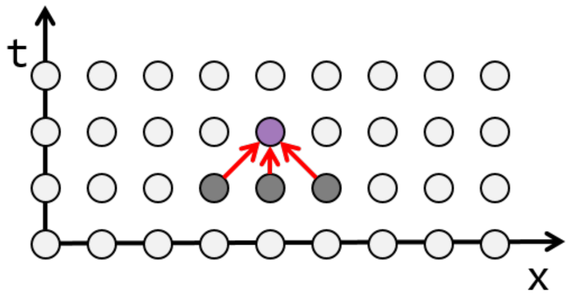


Figure 1: A 1D 3-point stencil.

A simple description of the trapezoidal decomposition is given below. We assume for simplicity that we are given a 1D 3-point stencil (see Figure 1) to be computed on an array of length $n = 2h$ for h time steps, where h is a power of 2. Hence, the space-time grid X for stencil updates will be an isosceles triangle X of base $2h$ and height h. The algorithm then works as follows (as shown in Figure 2). It first performs a "time cut" which splits X into the top isosceles triangle X_T (of height $h/2$ and base h) and the bottom isosceles trapezoid X_{LBR} (of height $h/2$ and bases h and $2h$). It solves X_{LBR} recursively first which involves performing a "space cut" of the trapezoid. It takes the midpoint of the larger of the two parallel sides and splits the trapezoid into three isosceles triangles (two upright triangles X_L and X_R with one inverted triangle X_B between them) by drawing lines parallel to the two lateral sides (legs). The algorithm then first recursively solves X_L and X_R in parallel, and then solves X_B. The algorithm recursively solves X_T after it is done solving X_{LBR}.

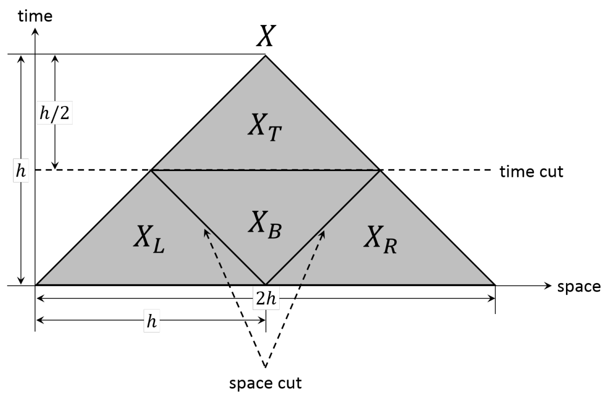
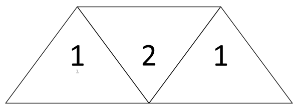


Figure 2: Illustration of how the standard trapezoidal decomposition algorithm [17] performs a stencil computation using recursive divide-and-conquer. We assume for simplicity that we are applying a 1D 3-point stencil and updating a stencil array of length $2h$ for h time steps, where h is a power of 2. Hence, the space-time grid X for stencil updates will be an isosceles triangle X of base $2h$ and height h. The trapezoidal decomposition algorithm first performs a time cut of X to split it into an upper triangle X_T and a bottom trapezoid X_{LBR} $(= X_L + X_B + X_L)$. The bottom trapezoid is solved recursively first. But a space cut is used to split the bottom trapezoid into two upright triangles X_L and X_R with one inverted triangle X_B between them. The algorithm then first recursively solves X_L and X_R in parallel, and then solves X_B recursively. After it is done solving the bottom trapezoid it recursively solves X_T.

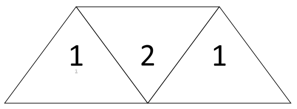
Frigo and Stumpen prove that this algorithm has optimal serial cache complexity [16–18].

The span (and thus parallelism) of the simple version of the trapezoidal algorithm described above can be improved by increasing the number of sub-trapezoids created during each space cut from 3 to at least $2r-1$ for some $r > 2$. As shown in [17] the resulting algorithm has a span of $O\left(rhn^{\frac{1}{\lg 2(r-1)}}\right)$, where h is the number of time steps to execute. The span can be improved further by using "compound space-time cuts" as explained in Figure 3. A compound space-time cut performs multiple time cuts and then space cuts inside each time slice. Such an approach can improve the span significantly. However, the resulting sub-trapezoids can become too small for the cache to fully exploit the temporal locality. As a result, the cache performance will drop.

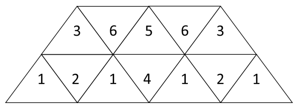
We apply the COW technique on the version of the trapezoidal algorithm with $r = 2$. That means each task still generates at most 3 recursive subtasks. There is no compound space-time cut. However, during a space cut all sibling subtasks (i.e., X_L, X_B and X_R in Figure 2) are spawned at the same recursion level simultaneously, and atomic operations are used to guard the data dependency on neighboring sub-



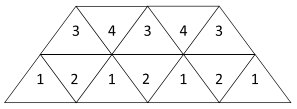
(a) Standard algorithm: timecuts = 0, parallel steps = 2.



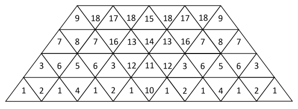
(d) Compound space-time cut: timecuts = 0, parallel steps = 2.



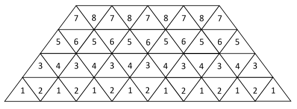
(b) Standard algorithm: timecuts = 1, parallel steps = 6.



(e) Compound space-time cut: timecuts = 1, parallel steps = 4.



(c) Standard algorithm: timecuts = 3, parallel steps = 18.



(f) Compound space-time cut: timecuts = 3, parallel steps = 8.

Figure 3: Comparison of the number of parallel steps executed by standard trapezoidal decomposition algorithm [17] and trapezoidal algorithm with compound space-time cuts. All blocks with the same number are executed together with blocks numbered i are executed before blocks numbered $i+1$. Vertical axis is time dimension and horizontal axis is space dimension.

tasks. A single array of length n suffices to implement all guards. Figure 4b shows the resulting COW algorithm while Figure 4a shows the original cache-oblivious trapezoidal decomposition algorithm with $r = 2$ (named 2-way CO).

Experimental results in Section 3 validate our claim that the COW version of the trapezoidal decomposition algorithm, indeed, improves span of the standard algorithm without losing cache-obliviousness and cache-efficiency.

3. Experimental Results

In this section we report empirical results on 1D 3-point stencils comparing our COW algorithm with parallel loops (without tiling), blocked parallel loops (with tiling) and the original trapezoidal decomposition algorithm.

For a fair comparison, we have used the same base case function and base case size for blocked loops, original trapezoidal decomposition and COW implementations of the same benchmark problem. As a result, the main difference between the original trapezoidal decomposition algorithm and the corresponding COW algorithm is in the approach used to choose the execution order of the recursive subtasks. In order to validate our claim that COW algorithms improve parallelism without sacrificing cache efficiency, we measure the burdened span of all algorithms using Cilkview [21] and L1/L2 cache misses using PAPI [6]. All measurement results reported in this section are "min" of at least three independent runs. The hardware platforms on which the experiments

Name	*Intel32*	*Intel16*
System	Intel Xeon E5-4650	Intel Xeon E5-2680
Clock	2.70 GHz	2.70 GHz
#Cores	4x8	2x8
L1 data cache	32 KB	32 KB
Last-level cache	20 MB	20 MB
Memory	1 TB	32 GB
OS	CentOS 6.3	CentOS 6.3
Compiler	icc v14.0	icc v14.0

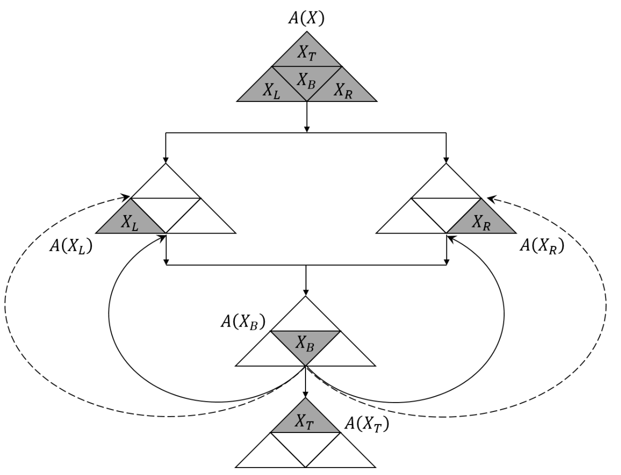
Figure 5: Machine specifications.

were performed are listed in Figure 5.

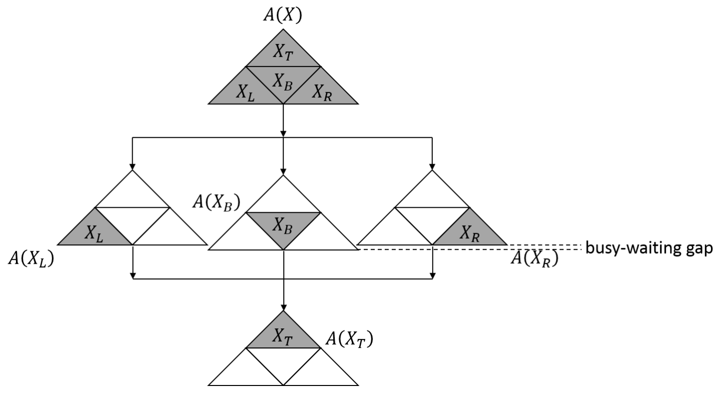
In Figure 6a we show the burdened span measured by Cilkview [21] for 1D stencil. The horizontal axis is the "side length n" which corresponds to the spatial dimension for 1D stencil (assuming that the length of 1D stencil array = $2\times$ the number of time steps). The vertical axis corresponds to the "burdened span" in log scale. Span is the length of the critical path in the directed acyclic graph (DAG) representation of a parallel program. Since parallelism is the average work performed by the program per unit length of the span, the smaller the span, the higher the parallelism assuming that the total work performed remains fixed. Burdened span adds the contributions of scheduling overhead to theoretical span. Cilkview [21] measures this span by counting the number of binary instructions on the critical path. Figure 6a shows that our COW algorithm (solid line with empty square) always has the lowest span among all algorithms compared. Parallel loops (without tiling) implementations (solid line with solid diamond), on the other hand, have the worst burdened span among the four algorithms because the loops are parallelized with granularity 1, which is too fine-grained to amortize the scheduling cost. Blocked loops and 2-way CO are in between although 2-way CO (solid line with solid square) generally has higher burdened span than blocked loops (solid line with empty diamond) because of the overhead of recursion in 2-way CO.

In Figure 6b we plot the L1 cache misses incurred by the algorithms on *Intel32*. The misses were measured by PAPI [6] by tracking hardware counters. The vertical axis corresponds to "L1 cache misses" and the horizontal axis to "base case size". Both axes are in log scale. Observe that our COW algorithm always has the best cache behavior. When the problem size is fixed, the gap between COW and blocked loops reduces with the increase of base case size as both implementations approach theoretical optimum. Cache misses incurred by blocked loop implementations approach that of COW algorithm as the base case size approaches cache size. L2 cache misses show similar patterns, and so are omitted.

Figure 7a plots absolute performance (updated points / second) of all algorithms on *Intel32*. Suffix 32 indicates results on *Intel32* and suffix 16 is used for *Intel16*. We mea-

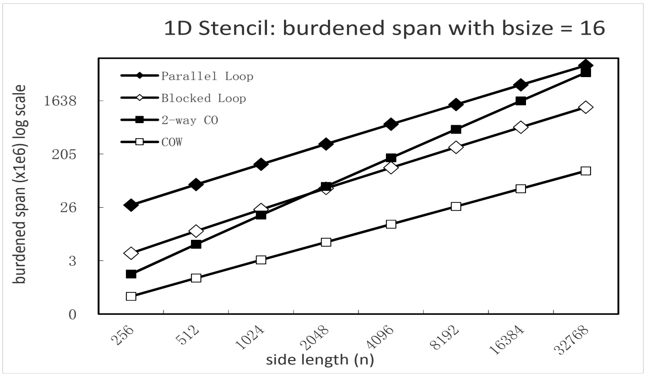


(a) Procedure A of trapezoidal decomposition in 2-way CO algorithm ($r = 2$).

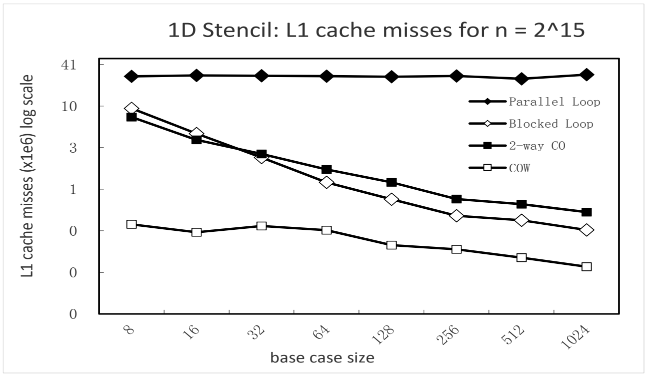


(b) Procedure A of trapezoidal decomposition using the COW technique.

Figure 4: Comparison between standard parallel cache-oblivious and COW algorithms for trapezoidal decomposition. The solid arrows indicate the dependencies from defining recurrence. The dashed arrows indicate the dependencies introduced by the divide-and-conquer structure of the algorithm. We assume that stencil computations on a space-time grid X is performed by calling $\mathrm{A}(X)$.



(a) Burdended span of 1D stencil

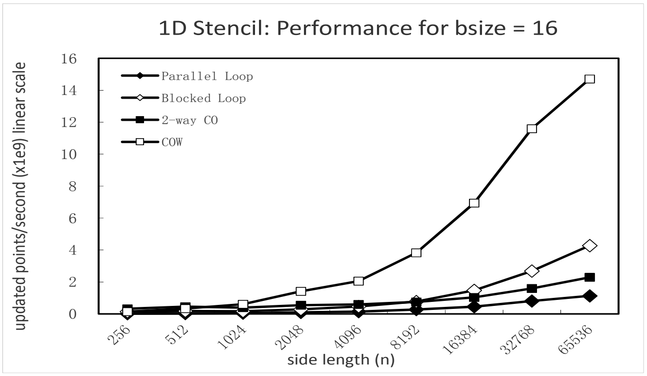


(b) L1 misses incurred by 1D stencil

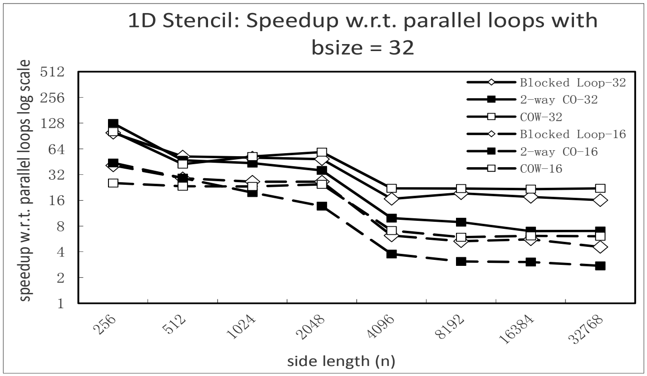
Figure 6: Comparison of burdened span (by Cilkview) and cache misses (by PAPI) among parallel loops, blocked loops, 2-way CO, and COW algorithms. *bsize* is the base case size. Both vertical and horizontal axes are log scale in both subfigures.

sured absolute performance using the `clock_gettime` function in Linux with monotonic clock (i.e., `CLOCK_MONOTONIC`). The vertical axis is “points updated per second” in linear scale and horizontal axis is “side length (n)” (please see the discussion on burdened span for the definition of “side length”) in log scale. Blocked loops always have better performance than 2-way CO when using exactly the same base case function and base case size. Higher parallelism of blocked loops may have contributed to its better performance. However, our COW algorithm beats blocked loops in almost all cases because it catches up with blocked loops in parallelism.

Figure 7b plots relative performance (speedup w.r.t. parallel loops) of various algorithms on *Intel32* and *Intel16*. The vertical axis represents “speedup w.r.t. parallel loops” in linear scale and horizontal axis is “side length (n)” in log scale. Observe that the COW algorithm generally benefits when the number of cores in the machine increases, especially for problem sizes between small and medium. Since the COW algorithm has been designed with the goal to improve parallelism without increasing cache complexity beyond that of standard 2-way CO algorithms, the behavior shown in the plots matches our expectations. When the problem size becomes very large compared to the base case size, all algorithms will have enough parallelism, and hence



(a) Absolute performance of 1D stencil



(b) Relative performance of 1D stencil

Figure 7: Comparison of absolute performance (updated points / second) on *Intel32* and relative performance (speedup with respect to parallel loops) on *Intel16* and *Intel32*. In both subfigures, the vertical axis is linear scale and horizontal axis is log scale. In charts of relative performance, the suffix 32 is the speedup on *Intel32* and the suffix 16 is the speedup on *Intel16*.

the running time will be bounded by work law (reduced gap in Figure 7b), i.e., the running time on p processing cores won't be smaller than the running time on 1 core divided by p, i.e., $T_p(n) \geq T_1(n)/p$.

4. Related Work

Recursive DAC algorithms, both serial and parallel, most of them having optimal serial cache complexity have been developed, implemented, and evaluated for LCS [7, 10], pairwise sequence alignment [11], parenthesis problem [8], gap problem [7], Floyd-Warshall's APSP [9], stencil computation [16–18, 30] etc. Some of these algorithms, such as algorithms for parenthesis and gap problems also achieved better parallelism than their parallel looping counterparts.

Hybrid r-way DAC algorithms with different values of r at different levels of recursion have been considered in [8]. These algorithms can reach parallel cache complexity matching the best sequential cache complexity, but the algorithms then become complicated to program, processor-aware, and often cache-aware.

Current implementation of our cache-oblivious wavefront technique relies on atomic operations on the data dependency path to guard the correctness of the algorithm in addition to the fork-join primitives. Atomic operations are also commonly used to implement parallel task graph execution systems such as Nabbit [2], BDDT [32], etc. The key difference between our technique and these task parallel graph execution systems is that these systems usually unroll the entire execution beforehand and execute all subtasks in a parallel looping fashion and as a result may lose cache efficiency. In case of our algorithm, recursion unfolds dynamically on-the-fly, and it inherits the cache-obliviousness and cache-optimality properties of the underlying 2-way recursive DAC algorithm.

5. Conclusion

We have shown how to achieve locality-preserving improvements in parallelism of the trapezoidal decomposition algorithm – a standard recursive DAC-based cache-oblivious stencil algorithm. Our approach works by propagating a wavefront of executing and ready-to-execute tasks through the dynamically unfolding recursive DAC tree. We have provided experimental results to validate our claims.

In stencil computations each cell in the space-time grid depends only on local/neighboring cells in the previous few time steps. We plan to extend our technique to recursive dynamic programming algorithms with more general dependency patterns, e.g., with dependencies on variable number of nonlocal cells (parenthesis problem) and horizontal dependencies (e.g., edit distance) [12].

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of the 12th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA 2000)*, pages 1–12, 2000.

[2] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work stealing. In *IPDPS*, pages 1–12. IEEE, April 2010.

[3] R. Bleck, C. Rooth, D. Hu, and L. T. Smith. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the North Atlantic. *Journal of Physical Oceanography*, 22(12):1486–1505, 1992. ISSN 0022-3670.

[4] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244. ACM, 2004.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded

runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[6] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. *SC Conference*, 0: 42, 2000.

[7] 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.

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

[9] R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4):878–919, 2010.

[10] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *In Proc. of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 06*, pages 591–600, 2006.

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

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[13] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 4:1–4:12, Austin, TX, Nov. 15–18 2008.

[14] H. Dursun, K.-i. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par*, pages 642–653, Delft, The Netherlands, Aug. 25–28 2009.

[15] H. Dursun, K.-i. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, Las Vegas, NV, July13–16 2009.

[16] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, pages 361–366, Cambridge, MA, June 20–22 2005.

[17] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA*, pages 271–280, 2006.

[18] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, New York, NY, Oct. 17–19 1999.

[21] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, Santorini, Greece, June 13–15 2010.

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

[23] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP*, pages 36–43, Chicago, IL, June 12 2005.

[24] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC*, pages 51–60, San Jose, CA, 2006. ISBN 1-59593-578-9.

[25] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, San Diego, CA, June 10–13 2007.

[26] A. Nakano, R. Kalia, and P. Vashishta. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Computer Physics Communications*, 83 (2-3):197–214, 1994. ISSN 0010-4655.

[27] A. Nitsure. Implementation and optimization of a cache oblivious lattice Boltzmann algorithm. Master's thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2006.

[28] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11, Rome, Italy, May 23–29 2009.

[29] A. Taflove and S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Norwood, MA, 2000. ISBN 1580530761.

[30] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *SPAA*, San Jose, CA, USA, 2011.

[31] Y. Tang, R. A. Chowdhury, C.-K. Luk, and C. E. Leiserson. Coding stencil computation using the Pochoir stencil-specification language. In *HotPar'11*, Berkeley, CA, USA, May 2011.

[32] 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 - 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers*, pages 17–31, 2013.

[33] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *IPDPS*, pages 1–14, Miami, FL, Apr. 2008.