The Cache Complexity of Multithreaded Cache Oblivious Algorithms

Matteo Frigo and Volker Strumpen*
IBM Austin Research Laboratory
11501 Burnet Road, Austin, TX 78758

December 7, 2007

Abstract

We present a technique for analyzing the number of cache misses incurred by multithreaded cache oblivious algorithms on an idealized parallel machine in which each processor has a private cache. We specialize this technique to computations executed by the Cilk work-stealing scheduler on a machine with dag-consistent shared memory. We show that a multithreaded cache oblivious matrix multiplication incurs $O(n^3/\sqrt{Z} + (Pn)^{1/3}n^2)$ cache misses when executed by the Cilk scheduler on a machine with P processors, each with a cache of size Z, with high probability. This bound is tighter than previously published bounds. We also present a new multithreaded cache oblivious algorithm for 1D stencil computations incurring $O(n^2/Z + n + \sqrt{Pn^{3+\epsilon}})$ cache misses with high probability, one for Gaussian elimination and back substitution, and one for the length computation part of the longest common subsequence problem incurring $O\big(n^2/Z + \sqrt{Pn^{3.58}}\big)$ cache misses with high probability.

1 Introduction

In this paper we derive bounds to the number of cache misses (the ***cache complexity***) incurred by a computation when executed by an idealized parallel machine with multiple processors. We assume that the computation is ***multithreaded***: The computation expresses a partial order on its instructions, and a scheduler external to the computation maps pieces of the

*This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

computation onto processors. The computation itself has no control over the schedule. Our main focus is on analyzing the cache complexity of parallel multithreaded cache oblivious algorithms [16], although, as a special case, our bounds also apply to a sequential process migrated from one processor to another by an operating system.

Past studies of the cache complexity have considered two complementary settings, each modeling different aspects of real machines. In the ***distributed-cache*** model, each processor is connected to a private cache that interacts somehow with the other caches to maintain a desired memory model. In the ***shared-cache*** model, a single cache is common to all processors, which are also referred to as (hardware) threads. This paper focuses on the distributed-cache model.

A multithreaded computation defines a partial execution order on its instructions, which we view as a directed acyclic graph (***dag***) [1, 6, 9, 26]. The ***work*** T_1 is the total number of nodes in the dag, and the ***critical path*** T_∞ is the length of a longest path in the dag. It is well-known that these two parameters characterize the dag for scheduling purposes: the execution time T_P of the dag on P processors satisfies $T_P \geq \max(T_1/P, T_\infty)$, and a greedy scheduler [10, 21] matches this lower bound within a factor of 2. A greedy scheduler is no longer asymptotically optimal when taking cache effects into account, however, and the best choice of a scheduler depends upon whether caches are distributed or shared. Roughly speaking, on a shared cache, threads that use the same data should be scheduled concurrently so as to maximize data reuse. On distributed caches, threads that do not share data should be scheduled concurrently so as to minimize inter-cache communication.

Multithreaded computations in the shared-cache model have been investigated by Blelloch and Gibbons [5] who proved a strong result: If the cache complexity of a computation is Q_1 on one processor with cache size Z_1, then a parallel schedule of the computation exists such that the cache complexity Q_P on P processors satisfies $Q_P \leq Q_1$, assuming that the P processors share a cache of slightly larger size $Z_P \geq Z_1 + PT_\infty$. Blelloch and Gibbons explicitly show a scheduler that achieves this bound.

The analysis of distributed caches is more involved. Acar et al. [1] construct a family of dags with work $\Theta(n)$ and critical path $\Theta(\lg n)$ whose cache complexity is bounded by $O(Z)$ on one processor with a cache of size Z, but it explodes to $\Omega(n)$ when the dag is executed in parallel on distributed caches. This result shows that a low cache complexity on one processor does not imply a low cache complexity on multiple processors for general dags. For series-parallel dags, however, more encouraging results are known. Blumofe

et al. [7] prove that the Cilk randomized work-stealing scheduler [9] executes a series-parallel computation on P processors incurring

$$Q_P(Z) \le Q_1(Z) + O(ZPT_\infty) \tag{1}$$

cache misses with high probability, where Q_1 is the number of cache misses in a sequential execution and Z is the size of one processor's cache. This bound holds for a "dag-consistent" shared memory with LRU caches. Acar et al. [1] prove a similar upper bound for race-free series-parallel computations under more general memory models and cache replacement strategies, taking into account the time of a cache miss and the time to steal a thread. The bound in Eq. (1) diverges to infinity as the cache size increases, and is actually tight for certain pathological series-parallel computations [1]. On the other hand, as we show in this paper, Eq. (1) is not tight for those "well-designed" programs whose sequential cache complexity decreases as the cache size increases, including cache oblivious algorithms.

In this paper, we introduce the ***ideal distributed cache model*** for parallel machines as an extension of the (sequential) ideal cache model [16], and we give a technique for proving bounds stronger than Eq. (1) for cache oblivious algorithms [16]. Our most general result (Theorem 1) has the following form. Consider the sequence of instructions of the computation in program order (the ***trace***). Assume that a parallel scheduler can be modeled as partitioning the trace into S "segments" of consecutive instructions, and that the scheduler assigns each segment to some processor. Cilk's work-stealing scheduler, for example, can be modeled in this way. Assume that the cache complexity of any segment of the trace is bounded by a nondecreasing concave function f of the work of the segment. Then the cache complexity of the parallel execution is at most $Sf(T_1/S)$. For the majority of existing cache oblivious algorithms, the cache complexity is indeed bounded by a concave function of the work, and therefore this analysis is applicable.[1] Furthermore, for the Cilk scheduler, the number of segments is $O(PT_\infty)$ with high probability, and thus we derive bounds to the cache complexity in terms of the work T_1, the critical path T_∞, and the sequential cache complexity.

[1] A trivial concave bound to the cache complexity always exists because the cache complexity is always at most as large as the work, and this linear bound is by definition concave. Our theory yields nontrivial results when the function is strictly concave, as is the case for the algorithms presented in this paper, for the FFT and sorting problems [16], for dynamic programming [11], etc., but not for problems such as matrix transposition where each cached datum is reused $\Theta(1)$ times.

For example, a multithreaded program for multiplying two $n \times n$ matrices without using additional storage has work $T_1 = O(n^3)$, critical path $T_\infty = O(n)$, and sequential cache complexity $Q_1 = O(n^3/\sqrt{Z} + n^2)$ [7]. When the program is executed on P processors by the Cilk scheduler, we prove that its cache complexity is $Q_P = O\big(n^3/\sqrt{Z} + (T_\infty P)^{1/3} n^2\big)$ with high probability. As another application, we present a new multithreaded cache oblivious algorithm for stencil computations, derived from our sequential algorithm [18]. Our one-dimensional stencil algorithm for a square space-time region has $T_1 = O(n^2)$, $T_\infty = O(n)$, and sequential cache complexity $Q_1 = O(n^2/Z + n)$. When executed on P processors by the Cilk scheduler, the cache complexity is $Q_P = O\big(n^2/Z + n + \sqrt{P n^{3+\epsilon}}\big)$ with high probability.

These bounds on the cache complexity allow a programmer to determine whether a program has sufficient temporal locality. If this is the case, the programmer can ignore the issues of data distribution and communication schedules without suffering a performance penalty. For example, matrix multiplication requires $\Omega\big(n^3/\sqrt{Z} + n^2\big)$ cache misses [24] irrespective of the number of processors, and we prove that a simple recursive matrix multiplication algorithm incurs $O\big(n^3/\sqrt{Z} + (nP)^{1/3} n^2\big)$ cache misses on P processors under the randomized work-stealing scheduler. If n is so large that the first term dominates the cache complexity, then any attempt to carefully orchestrate the communication schedule—which typically leads to complicated message-passing programs—would only yield incremental gains in the negligible second term. Thus, for large n or, equivalently, small P, one can achieve near-optimal performance with a simple program.

The remainder of this article is structured as follows. In Section 2 we present the ideal distributed cache model. In Section 3, we analyze the cache complexity of multithreaded computations on a machine with an ideal distributed cache. Then, in Section 4, we apply our cache complexity bounds to the analysis of multithreaded, cache oblivious programs for matrix multiplication, stencil computations, Gaussian elimination and back substitution, and for the length computation part of the longest common subsequence problem.

2 The Ideal Distributed Cache Model

In this section, we introduce the ***ideal distributed cache model*** for parallel machines as an extension of the ideal (sequential) cache model [16].

An ideal distributed-cache machine has a two-level memory hierarchy. The machine consists of P processors, each equipped with a private ideal

cache connected to an arbitrarily large shared main memory. An ***ideal cache*** is fully associative and it implements the optimal off-line strategy of replacing the cache line whose next access is farthest in the future [2]; see [16, 27] for a justification of this assumption.

Each private cache contains Z words (the ***cache size***), and it is partitioned into ***cache lines*** consisting of L consecutive words (the ***line size***) that are treated as atomic units of transfers between cache and main memory.

A processor can only access data in its private cache. If an accessed datum is not available in the cache, the processor incurs a ***cache miss*** to bring the data from main memory into its cache.

The number of cache misses incurred by a computation running on a processor depends on the initial state of the cache. The ***cache complexity*** Q of a computation is defined as the number of cache misses incurred by the computation on an ideal cache starting and ending with an empty cache.

The ideal distributed cache model assumes that the private caches are ***noninterfering***: the number of cache misses incurred by one processor can be analyzed independently of the actions of other processors in the system. Whether this assumption is true in practice depends on the consistency model maintained by the caches. For example, caches are noninterfering in the dag-consistent memory model maintained by the Backer protocol [7]. Alternatively, caches are noninterfering in the HSMS model [1] if the computation is race-free.

Our ideal distributed cache model is almost the same as the dag-consistent model analyzed by Blumofe et al. [7], except that we assume ideal caches instead of caches with LRU replacement. Bender et al. [3] consider a distributed-cache model, but with cache coherence and atomic operations. This model is harder to analyze than ours but it supports lock-free algorithms that are not possible with noninterfering caches. The shared ideal cache model of Blelloch and Gibbons [5] features an ideal cache which, unlike in our model, is shared by all processors. Like the PRAM [15] and its variants, the ideal distributed cache model aims at supporting a shared-memory programming model. Unlike the lock-step synchronous PRAM, and unlike bulk-synchronous models such as BSP [30] and LogP [14], our model is asynchronous, and processors operate independently most of the time. Like in the QSM model [19], each processor in our model features a private memory, but the QSM manages this private memory explicitly in software as part of each application, whereas we envision an automatically managed cache with hardware support.

3 The Cache Complexity of Multithreaded Computations

In this section, we prove bounds on the cache complexity of a multithreaded computation in terms of its sequential cache complexity, assuming an ideal distributed-cache machine. Specifically, Theorem 1 bounds the cache complexity of a multithreaded computation assuming a "generic" scheduler. Theorem 2 refines the analysis in the case of the Cilk work-stealing scheduler. Finally, Theorem 5 gives a technical result that simplifies the analysis of the cache complexity of divide-and-conquer computations.

Let the ***trace*** of a multithreaded computation be the sequence of the computation's instructions in some order consistent with the partial order defined by the multithreaded computation. Let a ***segment*** be a subsequence of consecutive instructions of the trace. We denote with $|\mathcal{A}|$ the length of segment $\mathcal{A}$, and with $Q(\mathcal{A})$ the number of cache misses incurred by the execution of segment $\mathcal{A}$ on an ideal cache that is empty at the beginning and at the end of the segment.

We assume that the computation is executed in parallel by a scheduler whose operation can be modeled as partitioning the trace into segments and assigning segments to processors. For each segment assigned to it, a processor executes the segment fully, and then proceeds to the execution of the next segment. When completing a segment, we assume that a processor completely invalidates and flushes its own cache (but not other caches), and we count the cache misses incurred by these actions as part of the parallel cache complexity. This technical assumption makes our upper-bound proofs easier; a real scheduler may apply optimizations to avoid redundant flushes. For correctness of the parallel execution, the scheduler must ensure that the assignment of segments to processors respects the data dependencies of the multithreaded computation, but our analysis holds for all partitions, including incorrect ones.

Recall that a function $f(x)$ is ***concave*** if $\alpha f(x_0)+(1-\alpha)f(x_1) \leq f(\alpha x_0 + (1-\alpha)x_1)$ holds for $0 \leq \alpha \leq 1$, for all x_0 and x_1 in the domain of f. For a concave function f and integer $S \geq 1$, ***Jensen's inequality*** holds:

$$\sum_{0 \leq i < S} f(x_i)/S \leq f\left(\sum_{0 \leq i < S} x_i/S\right).$$

Our first result relates the parallel cache complexity to the sequential cache complexity and the number of segments.

Theorem 1 *Let $\mathcal{M}$ be a trace of a multithreaded computation. Assume that a scheduler partitions $\mathcal{M}$ into S segments and executes the segments on an ideal distributed-cache machine. Let f be a concave function such that $Q(\mathcal{A}) \le f(|\mathcal{A}|)$ holds for all segments $\mathcal{A}$ of $\mathcal{M}$.*

Then, the total number $Q_P(\mathcal{M})$ of cache misses incurred by the parallel execution of the trace is bounded by

$$Q_P(\mathcal{M}) \le S \cdot f(|\mathcal{M}|/S) .$$

Proof: Let $\mathcal{A}_i$, $0 \le i < S$ be the segments generated by the scheduler. Because we assume that the scheduler executes a segment starting and ending with an empty cache, and because caches do not interfere with each other in the ideal-cache model, the parallel execution incurs exactly $Q_P(\mathcal{M}) = \sum_{0 \le i < S} Q(\mathcal{A}_i)$ cache misses. By assumption, we have $\sum_{0 \le i < S} Q(\mathcal{A}_i) \le \sum_{0 \le i < S} f(|\mathcal{A}_i|)$. By Jensen's inequality we have $\sum_{0 \le i < S} f(|\mathcal{A}_i|) \le Sf\left(\sum_{0 \le i < S} |\mathcal{A}_i|/S\right) = Sf(|\mathcal{M}|/S)$, and the theorem follows. Q.E.D.

Because a segment incurs at most as many cache misses as its number of memory accesses, Theorem 1 can always be applied trivially with $f(x) = x$. Theorem 1 becomes useful when we can find nontrivial concave functions, as in the examples in Section 4.

We now analyze the cache complexity of multithreaded Cilk [8, 17] programs assuming a dag-consistent shared memory [7]. Cilk extends the C language with fork/join parallelism managed automatically by a provably good work-stealing scheduler [9]. In general, a Cilk procedure is allowed to execute one of three actions: (1) execute sequential C code; (2) spawn a new procedure; or (3) wait until all procedures previously spawned by the same procedure have terminated. The latter operation is called a "sync". When a parent procedure spawns a child procedure, Cilk suspends the parent, makes it available to be "stolen" by another processor, and begins work on the child. When a processor returns from a child procedure, it resumes work on the parent if possible, or otherwise the processor becomes idle. An idle processor attempts to steal work from another, randomly selected processor. A procedure executing a sync may block, in which case the processor executing the procedure suspends it and starts stealing work.

The execution of a Cilk program can be viewed as a dag of dependencies among instructions. Whenever the dag contains an edge from node u to a node v executing on a different processor, the Backer protocol [7] inserts the following actions to enforce dag-consistent memory. The processor executing u, after executing it, writes all dirty locations in its cache back

to main memory. The processor executing v, before executing it but after the write back succeeding u, flushes and invalidates its cache and resumes with an empty cache. In terms of the Cilk source program, Backer can be viewed as inserting memory consistency actions in two places: (1) after a spawn at which a procedure is stolen, and (2) before the sync that waits for such a spawn to complete (the sync associated with the steal). If we view each processor as working on a segment, then a steal breaks the segment into four parts such that the noninterference assumption holds within each part: (1) the portion of the segment executed by the victim before the steal; (2) the portion of the segment executed by the victim after the steal; (3) the portion of the segment executed by the thief before the sync associated with the steal; and (4) the portion of the segment executed by the thief after the sync associated with the steal. Thus, each steal operation increases the number of segments by three. Combining this insight with Theorem 1 and the upper bounds of Acar et al. [1], we obtain the following theorem.

Theorem 2 (Cilk cache complexity) *Consider a Cilk computation with work T_1 and critical path T_∞, executed on an ideal distributed-cache machine with memory consistency maintained by the Backer protocol. Assume that a cache miss and a steal operation take constant time. Let f be a concave function such that $Q(\mathcal{A}) \leq f(|\mathcal{A}|)$ holds for all segments $\mathcal{A}$ of the trace of the computation.*

Then, the parallel execution on P processors incurs

$$Q_P = O(S \cdot f(T_1/S)) \tag{2a}$$

cache misses, where with high probability

$$S = O(PT_\infty) \ . \tag{2b}$$

Proof: Acar et al. [1, Lemma 16] prove that the Cilk scheduler executes $O(\lceil m/s \rceil PT_\infty)$ steals with high probability, where m is the time for a cache miss and s is the time for a steal. Their proof depends neither on the cache replacement policy nor on the memory model. By assumption, $\lceil m/s \rceil = \Theta(1)$ and we hide these parameters in the O-notation from now on. Each steal creates three segments, and therefore the number of segments is $S = O(PT_\infty)$ with high probability, proving Eq. (2b).

The length of the trace is the same as the work T_1. Caches maintained by Backer are noninterfering within each segment and therefore Theorem 1 applies, proving Eq. (2a). Q.E.D.

While we derived Theorem 2 for Cilk with dag-consistent shared memory, we could have applied the same analysis to race-free computations in the HSMS model of Acar et al. [1], obtaining the same bound.

In order to apply Theorems 1 and 2, one must prove a bound on the number of cache misses incurred by each of the unique segments of trace $\mathcal{M}$, which is hard to do in general. For example, consider a divide-and-conquer computation that recursively solves a problem of size n by reducing it to problems of size n/r. One can prove bounds on the cache complexity by induction on complete subtrees of the recursion tree, but this proof technique does not work for segments that do not correspond to complete subtrees.

To aid these proofs, we now prove Theorem 5 below, which bounds the cache complexity of an arbitrary segment in terms of the cache complexity of a subset of recursively nested segments. We call such a subset a *recursive decomposition* of the trace. In a divide-and-conquer computation, segments in the recursive decomposition would correspond to complete subtrees of the recursion tree. Then, Theorem 5 extends a bound on complete subtrees to a bound valid for all segments.

Definition 3 (Recursive segment decomposition) *Let $\mathcal{A}$ be a segment and $r \geq 2$ be an integer. A r-***recursive decomposition** *of $\mathcal{A}$ is any set $\mathcal{R}$ of subsegments of $\mathcal{A}$ produced by the following nondeterministic algorithm:*

If $|\mathcal{A}| = 1$, *then $\mathcal{R} = \{\mathcal{A}\}$.*

If $|\mathcal{A}| > 1$, *choose integer $q \geq 2$ and segments $\mathcal{A}_1$, $\mathcal{A}_2$, ..., $\mathcal{A}_q$ whose concatenation yields $\mathcal{A}$, such that $|\mathcal{A}_i| \geq |\mathcal{A}|/r$. Let $\mathcal{R}_i$ be a r-recursive decomposition of $\mathcal{A}_i$. Then $\mathcal{R} = \{\mathcal{A}\} \cup \mathcal{R}_1 \cup \mathcal{R}_2 \ldots \cup \mathcal{R}_r$. We say that segment $\mathcal{A}$ is the* **parent** *of the segments $\mathcal{A}_i$.*

Before proving Theorem 5, we state a rather obvious property of ideal caches.

Lemma 4 (Monotonicity of an ideal cache) *Let $\mathcal{A}$ and $\mathcal{B}$ be segments of a trace with $\mathcal{B} \subset \mathcal{A}$. Then $Q(\mathcal{B}) \leq Q(\mathcal{A})$.*

Proof: Execute $\mathcal{B}$ on a cache that incurs exactly the same cache misses as an optimal execution of $\mathcal{A}$, in the same order. In this case, execution of $\mathcal{B}$ incurs exactly $Q(\mathcal{A})$ cache misses. An optimal replacement policy for $\mathcal{B}$ incurs at most as many cache misses as the policy that we have just discussed. Q.E.D.

Theorem 5 *Let $\mathcal{R}$ be a r-recursive decomposition of trace $\mathcal{M}$. Let f be a nondecreasing function such that $Q(\mathcal{A}) \leq f(|\mathcal{A}|)$ for all $\mathcal{A} \in \mathcal{R}$. Then, for all segments $\mathcal{A}$ of $\mathcal{M}$ (not only those in $\mathcal{R}$) we have $Q(\mathcal{A}) \leq 2f(r\,|\mathcal{A}|)$.*

Proof: We prove that $\mathcal{A}$ is included in the concatenation of at most two segments in $\mathcal{R}$ of length at most $r|\mathcal{A}|$. The theorem then follows from Lemma 4.

Let $\mathcal{B}$ be the smallest segment in $\mathcal{R}$ that includes $\mathcal{A}$. Such a segment exists because the entire trace $\mathcal{M} \in \mathcal{R}$.

If a child $\mathcal{B}' \in \mathcal{R}$ of $\mathcal{B}$ exists that is included in $\mathcal{A}$, then $|\mathcal{B}|/r \leq |\mathcal{B}'| \leq |\mathcal{A}|$. By Lemma 4 we have $Q(\mathcal{A}) \leq Q(\mathcal{B}) \leq f(|\mathcal{B}|) \leq f(r\,|\mathcal{A}|) \leq 2f(r\,|\mathcal{A}|)$ and we are done.

Otherwise, two consecutive children $\mathcal{B}'$ and $\mathcal{B}''$ of $\mathcal{B}$ exist in $\mathcal{R}$ such that $\mathcal{A}$ is included in the concatenation of $\mathcal{B}'$ and $\mathcal{B}''$. Let $\mathcal{A}' = \mathcal{A} \cap \mathcal{B}'$ and $\mathcal{A}'' = \mathcal{A} \cap \mathcal{B}''$. Then, by construction, $\mathcal{A}'$ is a suffix of $\mathcal{B}'$ and $\mathcal{A}''$ is a prefix of $\mathcal{B}''$.

We now prove that $Q(\mathcal{A}') \leq f(r\,|\mathcal{A}'|)$. Let $\mathcal{C}'$ be the smallest segment in $\mathcal{R}$ that includes $\mathcal{A}'$. If $\mathcal{A}'$ is empty or $\mathcal{A}' = \mathcal{C}'$, then $Q(\mathcal{A}') \leq f(|\mathcal{A}'|) \leq f(r\,|\mathcal{A}'|)$. Otherwise, a child of $\mathcal{C}'$ exists in $\mathcal{R}$. By construction, $\mathcal{A}'$ is a suffix of $\mathcal{C}'$, and therefore the rightmost child $\mathcal{D}'$ of $\mathcal{C}'$ is included in $\mathcal{A}'$. Therefore, we have $|\mathcal{C}'|/r \leq |\mathcal{D}'| \leq |\mathcal{A}'|$. By Lemma 4, we have $Q(\mathcal{A}') \leq Q(\mathcal{C}') \leq f(|\mathcal{C}'|) \leq f(r\,|\mathcal{A}'|)$, as claimed.

A symmetric argument, substituting "prefix" for "suffix," proves that $Q(\mathcal{A}'') \leq f(r\,|\mathcal{A}''|)$.

By Lemma 4, we have $Q(\mathcal{A}) \leq Q(\mathcal{A}') + Q(\mathcal{A}'') \leq f(r\,|\mathcal{A}'|) + f(r\,|\mathcal{A}''|)$. By monotonicity of f, we conclude that $Q(\mathcal{A}) \leq 2f(r\,|\mathcal{A}|)$ and the theorem is proven. Q.E.D.

By combining Theorems 2 and 5, we obtain the following bound on the parallel cache complexity in terms of the number of segments and of the sequential cache complexity of a recursive decomposition.

Corollary 6 *Let $\mathcal{R}$ be a r-recursive decomposition of trace $\mathcal{M}$. Let f be a nondecreasing concave function such that $Q(\mathcal{A}) \leq f(|\mathcal{A}|)$ for all $\mathcal{A} \in \mathcal{R}$. Assume a Cilk scheduler with Backer as in Theorem 2.*

Then, the total number $Q_P(\mathcal{M})$ of cache misses incurred by the parallel execution of the trace on P processors is

$$Q_P = O(S \cdot f(rT_1/S))$$

cache misses, where, with high probability,

$$S = O(PT_\infty)\ .$$

Proof: Let $g(x) = 2f(rx)$. Then, g is concave. By Theorem 5, we have $Q(\mathcal{A}) \le g(|\mathcal{A}|)$ for all segments $\mathcal{A}$ of $\mathcal{M}$. The corollary then follows from Theorem 2. Q.E.D.

Remark: If $Sf(rT_1/S)$ happens to be a concave function of S, then the bounds hold in expectation as well, because then we have $E[Sf(rT_1/S)] \le E[S]f(rT_1/E[S])]$, and $E[S] = O(PT_\infty)$ holds [1].

4 Applications

In this section, we apply Corollary 6 to the analysis of parallel cache oblivious algorithms for matrix multiplication, stencil computations, a linear system solver, and longest common subsequence computations. All applications are programmed in Cilk [17]. A similar analysis could be applied to suitable parallelizations of other cache oblivious algorithms.

4.1 Matrix Multiplication

Fig. 1 shows the pseudo code of multithreaded procedure `matmul` for multiplying two $n \times n$ matrices for $n = 2^k$ [7]. This procedure executes $T_1 = O(n^3)$ work and its critical path is $T_\infty = O(n)$.[2]

If we ignore the `spawn` annotations and the `sync` statements, we obtain a special case of the sequential cache oblivious matrix multiplication algorithm, which incurs $Q(n, Z, L) = O\big(n^3/(L\sqrt{Z}) + n^2/L + 1\big)$ cache misses [16] when executing on one processor with an ideal cache of size Z and cache line size L, assuming a "tall" cache with $Z = \Omega(L^2)$. This cache complexity is asymptotically optimal [24].

The trace of procedure `matmul` admits a simple 8-recursive decomposition comprising all segments that compute a complete subtree of the call tree. Moreover, the analysis of the sequential case applies to each complete subtree. Hence, on our ideal distributed-cache machine, each subtree that multiplies $n \times n$ matrices incurs $O\big(n^3/(L\sqrt{Z}) + n^2/L + 1\big)$ cache misses.

To apply Corollary 6, we must find a concave function f that bounds the number of cache misses Q as a function of a segment's length, for all segments in the recursive decomposition. Consider a segment in the recursive decomposition that multiplies $n \times n$ matrices. Ignoring constant factors, let

[2] A shorter critical path is possible at the expense of additional storage if addition is associative; see [7].

```
cilk void matmul(n, A, B, C)
{
    if (n == 1) {
        C += A * B;
    } else {
        spawn matmul(n/2, A11, B11, C11);
        spawn matmul(n/2, A11, B12, C12);
        spawn matmul(n/2, A21, B11, C21);
        spawn matmul(n/2, A21, B12, C22);

        sync;

        spawn matmul(n/2, A12, B21, C11);
        spawn matmul(n/2, A12, B22, C12);
        spawn matmul(n/2, A22, B21, C21);
        spawn matmul(n/2, A22, B22, C22);
    }
}
```

Figure 1: Cilk pseudo code for computing $C = C + AB$, where A, B, and C are $n \times n$ matrices. The code for partitioning each matrix into four quadrants is not shown. The `spawn` keyword declares that the spawned procedure may be executed in parallel with the procedure that executes the spawn. A `sync` statement waits until all procedures spawned by the current procedure have terminated. Cilk implicitly `sync`'s before returning from a procedure.

$w = n^3$ be the length of the segment. Then $Q \leq f(w)$ for some concave function $f(w) \in O\big(w/(L\sqrt{Z}) + w^{2/3}/L + 1\big)$.

Since f is concave, we obtain the cache complexity of a parallel execution of `matmul` by Corollary 6 as

$$Q_P(n, Z, L) = O\big(n^3/(L\sqrt{Z}) + S^{1/3}n^2/L + S\big) , \tag{4}$$

where $S = O(Pn)$ with high probability.

Comparison With Previous Bounds. Assume now for simplicity that $L = \Theta(1)$. The sequential cache complexity is $Q(n, Z, L) = O(n^3/\sqrt{Z} + n^2)$ and the Cilk cache complexity is

$$Q_P(n, Z) = O\big(n^3/\sqrt{Z} + (Pn)^{1/3}n^2\big) . \tag{5}$$

How does the "new" bound Eq. (5) compare to the "old" bound

$$Q_P(n, Z) = O\big(n^3/\sqrt{Z} + ZPn\big) \tag{6}$$

that was derived by Blumofe et al. [7]? As $Z \to \infty$, Eq. (5) remains bounded, whereas Eq. (6) diverges, and thus the new bound is asymptotically tighter than the old bound for some values of the parameters. If $n^3/\sqrt{Z} \geq (Pn)^{1/3}n^2$, then the new bound is $O(n^3/\sqrt{Z})$ and the old bound is $\Omega(n^3/\sqrt{Z})$, and therefore the old bound is not tighter than the new one. Otherwise, we have $n^3/\sqrt{Z} \leq (Pn)^{1/3}n^2$, and thus $\sqrt{Z} \geq (Pn)^{-1/3}n$, from which $ZPn \geq (Pn)^{1/3}n^2$ follows. Consequently, the new bound is $O((Pn)^{1/3}n^2)$, whereas the old bound is $\Omega(ZPn) = \Omega((Pn)^{1/3}n^2)$, and therefore the old bound is not tighter than the new one in this case either. Thus, we conclude that the new bound strictly subsumes the old bound.

4.2 1D Parallel Stencil Algorithm

In this section, we present a parallel cache oblivious algorithm for stencil computations, derived from our sequential cache oblivious algorithm [18], and we analyze its cache complexity in the ideal cache model. Bilardi and Preparata [4] analyze a more complicated parallel cache oblivious stencil algorithm in a limiting technology where signal propagation at the speed of light is the primary performance bottleneck.

A ***stencil*** defines the computation of an element in an n-dimensional spatial grid at time t as a function of neighboring grid elements at time $t - 1, \ldots, t - k$. The n-dimensional grid plus the time dimension span an $(n + 1)$-dimensional ***spacetime***. For brevity we restrict our discussion to one-dimensional stencils. The algorithm can be extended to stencils with arbitrary dimensions as discussed in [18].

Our parallel, cache oblivious stencil algorithm applies to ***in place*** computations that store only a bounded number of spacetime points for each space position, as opposed to storing the entire spacetime. For example, the sequential, non cache oblivious program in Fig. 2 computes $(t_1 - t_0)(x_1 - x_0)$ spacetime points using only $2(x_1 - x_0)$ memory locations. We call a multidimensional array, such as u`[2][`N`]` in Fig. 2, with two places to store two versions of each value a ***toggle array***.[3] Most stencil computations used in practice are in place, predominantly employ variants of toggle arrays, and therefore this restriction is not serious. Reusing storage is necessary to benefit from a cache: A stencil computation that did not reuse storage would incur a number of cache misses proportional to the size of the spacetime irrespective of the cache size.

[3] Other in-place storage schemes are possible [28].

```
double u[2][N];

void kernel(int t, int x)
{
     u[(t+1)%2][x] = f(u[t%2][x-1], u[t%2][x], u[t%2][x+1]);
}

void iter(int t0, int t1, int x0, int x1)
{
     int t, x;
     for (t = t0; t < t1; t++)
          for (x = x0; x < x1; x++)
               kernel(t, x);
}
```

Figure 2: Sequential C program that implements a 3-point stencil computation. The program computes all spacetime points (t, x) in the rectangle $t_0 \leq t < t_1$, $x_0 \leq x < x_1$, where point $(t+1, x)$ depends upon points $(t, x-1)$, (t, x), and $(t, x+1)$. The exact dependency is determined by a function `f`, not shown. The program is in place: It does not store all spacetime points in separate memory locations, but it reuses the same location for multiple spacetime points. Specifically, the program stores point (t, x) in array position u[t mod 2][x]. Reusing storage is necessary to benefit from a cache. (The C syntax `a%b` denotes the remainder of `a` divided by `b`.)

4.2.1 Description of the 1D Stencil Algorithm

Procedure `walk1` in Fig. 6 visits all points (t, x) in a rectangular spacetime region, where $0 \leq t < T$, $0 \leq x < N$, and t and x are integers. The procedure visits point $(t+1, x)$ after visiting points $(t, x+k)$ for $|k| \leq \sigma$, where $\sigma \geq 1$ (the ***slope***) is an integer sufficiently large to ensure that the procedure respects the dependencies imposed by the stencil.[4] For example, a suitable slope for for a 3-point stencil is $\sigma = 1$, because spacetime point $(t+1, x)$ depends upon points $(t, x-1)$, (t, x), and $(t, x+1)$.

Although we are ultimately interested in traversing rectangular spacetime regions, the procedure operates on more general trapezoidal regions such as the one shown in Fig. 3. For integers t_0, t_1, x_0, $\dot{x}_0$, x_1, and $\dot{x}_1$, we define the ***trapezoid*** $\mathcal{T}(t_0, t_1, x_0, \dot{x}_0, x_1, \dot{x}_1)$ to be the set of integer pairs (t, x) such that $t_0 \leq t < t_1$ and $x_0 + \dot{x}_0(t - t_0) \leq x < x_1 + \dot{x}_1(t - t_0)$. (We use the Newtonian notation $\dot{x} = dx/dt$.) The ***height*** of the trapezoid is

[4]It is possible to modify our algorithm to work for $\sigma = 0$, although it is always safe to choose a value of σ larger than strictly necessary.

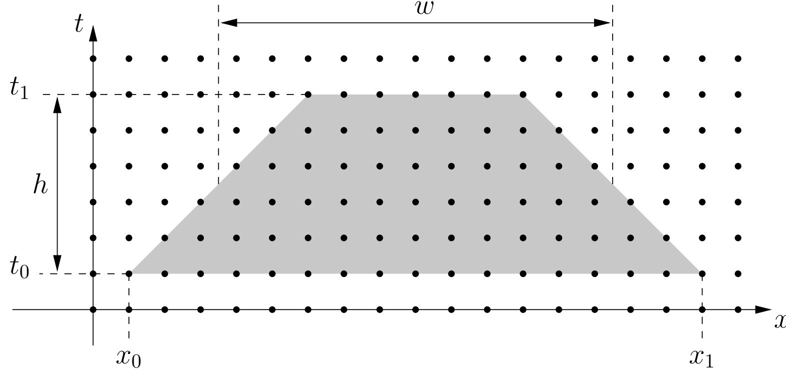

Figure 3: Illustration of the trapezoid $\mathcal{T}(t_0, t_1, x_0, \dot{x}_0, x_1, \dot{x}_1)$ for $\dot{x}_0 = 1$ and $\dot{x}_1 = -1$. The trapezoid includes all points in the shaded region, except for those on the top and right edges.

$h = t_1 - t_0$, and we define the ***width*** to be the average of the lengths of the two parallel sides, i.e. $w = (x_1 - x_0) + (\dot{x}_1 - \dot{x}_0)h/2$. The ***center*** of the trapezoid is point (t, x), where $t = (t_0 + t_1)/2$ and $x = (x_0 + x_1)/2 + (\dot{x}_0 + \dot{x}_1)h/4$ (i.e., the average of the four corners). The ***area*** of the trapezoid is the number of points in the trapezoid. We only consider ***well-defined trapezoids***, for which these three conditions hold: $t_1 \geq t_0$, $x_1 \geq x_0$, and $x_1 + h \cdot \dot{x}_1 \geq x_0 + h \cdot \dot{x}_0$.

Procedure `walk1` decomposes $\mathcal{T}$ recursively into smaller trapezoids, according to the following rules.

Parallel space cut: Whenever possible, the procedure executes a parallel space cut, decomposing $\mathcal{T}$ into into r "black" trapezoids and some number of "gray" trapezoids, as illustrated in Fig. 4. The procedure spawns the black trapezoids in parallel, waits for all of them to complete, and then spawns the gray trapezoids in parallel. Such an execution order is correct because the procedure operates the cut so that (1) points in different black trapezoids are independent of each other, (2) points in different gray trapezoids are independent of each other, and (3) points in a black trapezoid do not depend on points in a gray trapezoid.

The base of each black trapezoid has length $l = \lfloor (x_1 - x_0)/r \rfloor$, except for the rightmost one, which may be larger because of rounding. A black trapezoid has the form $\mathcal{T}(t_0, t_1, x, \sigma, x + l, -\sigma)$. Slope σ of the edges is necessary to guarantee that a point in a black trapezoid does

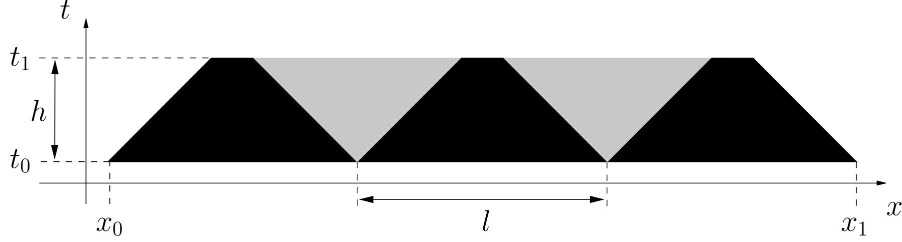

Figure 4: Illustration of a *parallel space cut*. The black trapezoids are independent of each other and can be visited in parallel. Once these trapezoids have been visited, the gray trapezoids can in turn be visited in parallel.

not depend on points in a gray trapezoid. A black trapezoid is well-defined only if the condition $l \geq 2\sigma h$ holds, or else the trapezoid would be self-intersecting. Therefore, r black trapezoids fit into $\mathcal{T}$ only if $x_1 - x_0 \geq 2r\sigma h$, which is the condition for the applicability of the parallel space cut.

The procedure always generates $r+1$ gray trapezoids, of which $r-1$ are located between black trapezoids, as in Fig. 4, and two are located at the left and right edges of $\mathcal{T}$. In Fig. 4, the trapezoids at the edges happen to be have zero area. The gray trapezoids in the middle are in fact triangles of the form $\mathcal{T}(t_0, t_1, x, -\sigma, x, \sigma)$.

We leave the constant r unspecified for now. The choice of r involves a tradeoff between the critical path and the cache complexity, which we analyze in Section 4.2.2.

Time cut: If $h > 1$ and the parallel space cut is not applicable, procedure `walk1` cuts the trapezoid along the horizontal line through the center, as illustrated in Fig. 5. The recursion first traverses trapezoid $\mathcal{T}_1 = \mathcal{T}(t_0, t_0 + s, x_0, \dot{x}_0, x_1, \dot{x}_1)$, and then trapezoid $\mathcal{T}_2 = \mathcal{T}(t_0 + s, t_1, x_0 + \dot{x}_0 s, \dot{x}_0, x_1 + \dot{x}_1 s, \dot{x}_1)$, where $s = \lfloor h/2 \rfloor$.

Base case: If $h = 1$, then $\mathcal{T}$ consists of the line of spacetime points (t_0, x) with $x_0 \leq x < x_1$. The base case visits these points, calling the application-specific procedure `kernel` for each of them. The traversal order is immaterial because these points are independent of each other.

The work (sequential execution time) of procedure `walk1`, when traversing a trapezoid, is proportional to the trapezoid's area, i.e., $T_1 = \Theta(wh)$ where w is the width of the trapezoid and h is its height. This fact is

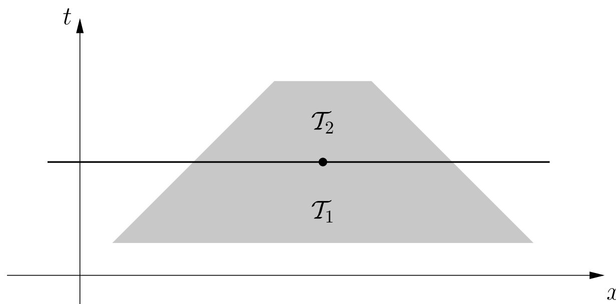

Figure 5: Illustration of a *time cut*. The algorithm cuts the trapezoid along the horizontal line through its center, it recursively visits $\mathcal{T}_1$, and then it visits $\mathcal{T}_2$.

not completely obvious because the procedure may spawn up to two empty gray trapezoids in case of a space cut, and the procedure needs nonconstant $\Theta(h)$ time to execute an empty trapezoid of height h. This additional work is asymptotically negligible, however. Procedure `walk1` obeys the bound $T_1(w, h) \leq 2rT_1(w/(2r), h) + O(h)$ in case of a space cut, and bound $T_1(w, h) \leq 2T_1(w, h/2) + O(1)$ in case of a time cut. One can verify by induction that $T_1(w, h) \leq c(wh - w - h)$ holds for some constant c and sufficiently large w and h. Alternatively, one can modify the procedure to test for empty trapezoids at the beginning, thus avoiding this problem altogether, but we prefer to keep the code simple even if the analysis becomes slightly harder.

We conclude this section with the analysis of the critical path length of `walk1`.

Theorem 7 *The critical path of `walk1` when visiting trapezoid $\mathcal{T}$ is*

$$T_\infty(\mathcal{T}) = O\left(\sigma r h w^{1/\lg(2(r-1))}\right) ,$$

where h is the height of $\mathcal{T}$, w is its width, σ is the slope of the stencil, and r is the number of black trapezoids created by the procedure in the space-cut case.

Proof: To avoid cluttering the proof with the O-notation, assume that a call to the `kernel` procedure and a `spawn` cost at most one unit of critical path. Furthermore, let $\alpha = 1/\lg(2(r-1))$ for brevity. Because procedure `walk1` uses $r \geq 2$ to spawn at least two threads in the space cut, we have $\alpha \leq 1$.

```
0   const int σ;     /* assert(σ >= 1) */
1   const int r;     /* assert(r >= 2) */

2   cilk void walk1(int t0, int t1, int x0, int ẋ0, int x1, int ẋ1)
3   {
4      int h = t1 - t0, Δx = x1 - x0;
5      int x, i;

6      if (h >= 1 && Δx >= 2 * σ * h * r) {      /* parallel space cut */
7          int l = Δx / r;      /* base of a black trapezoid, rounded down */

8          for (i = 0; i < r - 1; ++i)
9               spawn walk1(t0, t1, x0 + i * l, σ, x0 + (i+1) * l, -σ);
10         spawn walk1(t0, t1, x0 + i * l, σ, x1, -σ);

11         sync;

12         spawn walk1(t0, t1, x0, ẋ0, x0, σ);
13         for (i = 1; i < r; ++i)
14              spawn walk1(t0, t1, x0 + i * l, -σ, x0 + i * l, σ);
15         spawn walk1(t0, t1, x1, -σ, x1, ẋ1);
16     } else if (h > 1) {                      /* time cut */
17         int s = h / 2;
18         spawn walk1(t0, t0 + s, x0, ẋ0, x1, ẋ1);
19         sync;
20         spawn walk1(t0 + s, t1, x0 + ẋ0 * s, ẋ0, x1 + ẋ1 * s, ẋ1);
21     } else if (h == 1) {                     /* base case */
22         for (x = x0; x < x1; ++x)
23             kernel(t0, x);
24     }
25  }
```

Figure 6: One-dimensional parallel stencil algorithm implemented in the Cilk language. The procedure is parametrized by two integers σ and r, whose meaning is described in the text. In lines 8–10, we spawn r black trapezoids. Because of the rounding of l in line 7, the length of the base of the last trapezoid is not necessarily l, and we handle this trapezoid separately in line 10. The `sync` statement in line 11 waits for the black trapezoids to complete, before spawning the gray trapezoids in lines 12–15.

We now prove that

$$T_\infty(h,w) \le 2\sigma r(2w^\alpha h - 1) \tag{7}$$

by induction on the area of the trapezoid.

Base case: If $h = 1$ and $1 \le w < 2\sigma r$, then the procedure enters its base case with a critical path $T_\infty(h,w) = w \le 2\sigma r \le 2\sigma r(2w^\alpha - 1)$, and Eq. (7) holds.

Inductive step: Otherwise, the procedure recursively cuts the trapezoid into strictly smaller trapezoids for which we assume inductively that Eq. (7) holds. Depending on whether the procedure executes a time cut or a parallel space cut, we distinguish two cases.

Time cut: If the procedure executes a time cut, we have

$$T_\infty(h,w) \le T_\infty(h/2,w_1) + T_\infty(h/2,w_2) + 1 \ ,$$

where w_1 and w_2 are the widths of the two trapezoids produced by the cut. By inductive hypothesis, we have

$$T_\infty(h/2,w_i) \le 2\sigma r(2w_i^\alpha h/2 - 1) \ .$$

Since $\alpha \le 1$ holds, w^α is a concave function of w. By Jensen's inequality, we have $w_1^\alpha + w_2^\alpha \le 2((w_1+w_2)/2)^\alpha = 2w^\alpha$. Consequently, the following inequalities hold:

$$\begin{aligned}
T_\infty(h,w) &\le 2\sigma r(2w_1^\alpha h/2 - 1) + 2\sigma r(2w_2^\alpha h/2 - 1) + 1 \\
&\le 2\sigma r\left((w_1^\alpha + w_2^\alpha)h - 2\right) + 1 \\
&\le 2\sigma r(2w^\alpha h - 2) + 1 \\
&\le 2\sigma r(2w^\alpha h - 1) \ ,
\end{aligned}$$

thereby proving Eq. (7) in the time-cut case.

Space cut: If the procedure executes a parallel space cut, it generates at least $r-1$ gray trapezoids of width $w_g = \sigma h$, and r black trapezoids of width w_b. The critical path is the sum of the critical paths of one black and one gray trapezoid, plus an additional critical path $2r$ for spawning the recursive subproblems. Therefore, we have

$$T_\infty(h,w) \le T_\infty(h,w_b) + T_\infty(h,w_g) + 2r \ .$$

The sum of the widths of the black trapezoids is at most $w - (r-1)w_g$, and therefore we have

$$\begin{aligned} w_b &\leq (w - (r-1)w_g)/r \\ &\leq (w - (r-1)w_g)/(r-1) \\ &\leq w/(r-1) - w_g \ . \end{aligned}$$

Consequently, we have

$$\begin{aligned} T_\infty(h, w) &\leq T_\infty(h, w_b) + T_\infty(h, w_g) + 2r \\ &\leq 2\sigma r(2(w_b^\alpha + w_g^\alpha)h - 2) + 2r \\ &\leq 2\sigma r(2((w/(r-1) - w_g)^\alpha + w_g^\alpha)h - 2) + 2r \ . \end{aligned}$$

Again by Jensen's inequality, we have

$$(w/(r-1) - w_g)^\alpha + w_g^\alpha \leq 2\left(w/(2(r-1))\right)^\alpha = w^\alpha \ ,$$

from which we conclude that

$$\begin{aligned} T_\infty(h, w) &\leq 2\sigma r(2w^\alpha h - 2) + 2r \\ &\leq 2\sigma r(2w^\alpha h - 1) \ . \end{aligned}$$

Since Eq. (7) holds in the base case, in the time-cut case, and in the space-cut case, the theorem follows by induction. Q.E.D.

4.2.2 Cache Complexity of the 1D Stencil Algorithm

We now analyze the cache complexity of our parallel stencil procedure `walk1`. We assume an ideal cache with line size $L = \Theta(1)$, because a general line size only complicates the analysis without yielding further insights. The analysis depends on two geometric invariants which we now state.

Lemma 8 (Aspect ratio) *If procedure `walk1` traverses a trapezoid of height h_0, then for each subtrapezoid of height h and width w created by the procedure, the invariant $h \geq \min(h_0, w/(4\sigma(r+1)))$ holds, where σ is the slope of the stencil and r is the number of black trapezoids created by the procedure in the space-cut case.*

Proof: The proof is by induction on the number of cuts required to produce a subtrapezoid. The invariant holds by definition of h_0 at the beginning of the execution. The base case produces no subtrapezoids, and therefore it

trivially preserves the invariant. A parallel space cut does not change h and does not increase w, thus preserving the invariant. In the time-cut case, $\Delta x = x_1 - x_0 \leq 2\sigma rh$ holds by construction of the procedure. Because $|\dot{x}_i| \leq \sigma$, we have $w \leq \Delta x + \sigma h$. The time cut produces trapezoids of height $h' = h/2$ and width $w' \leq w + \sigma h \leq \Delta x + 2\sigma h \leq 2\sigma(r+1)h$. Thus, a time cut preserves the invariant, and the lemma is proven. Q.E.D.

Lemma 9 (Aspect ratio after space cuts) *If procedure `walk1` traverses a trapezoid of height h_0, then each space cut produces trapezoids of height h and width w with $h \geq \min(h_0, \Omega(w/\sigma))$, where σ is the slope of the stencil . The constant hidden in the Ω-notation does not depend upon the parameter r of the procedure.*

Proof: Before applying a space cut to a trapezoid of width w and height h, Lemma 8 holds. The space cut produces trapezoids of width $w' = \Theta(w/r)$ and of the same height h, and therefore we have $h \geq \min(h_0, \Omega(w'/\sigma))$. Q.E.D.

Theorem 10 (Sequential cache complexity) *Let procedure `walk1` traverse a trapezoid of height h_0 on a single processor with an ideal cache of size Z and line size $L = \Theta(1)$. Then each subtrapezoid $\mathcal{T}$ of height h and width w generated by `walk1` incurs at most*

$$O\left(\frac{wh}{Z/(\sigma r)} + \frac{wh}{h_0} + w\right)$$

cache misses, where σ is the slope of the stencil and r is the number of black trapezoids created by the procedure in the space-cut case.

Proof: Let W be the maximum integer such that the working set of any trapezoid of width W fits into the cache. We have $W = \Theta(Z)$.

If $w \leq W$, then the procedure incurs $O(w)$ cache misses to read and write the working set once, and the theorem is proven.

If $w > W$, consider the set of maximal subtrapezoids of width at most W generated by the procedure while traversing $\mathcal{T}$. These trapezoids are generated either by a space cut or by a time cut. Trapezoids generated by a time cut have width $w' = \Omega(W)$ and height $h' = \Omega\big(\min(h_0, w'/(\sigma r))\big)$ by Lemma 8. Trapezoids generated by a space cut have width $w' = \Omega(W/r)$ and height $h' = \Omega\big(\min(h_0, w'/\sigma))\big)$ by Lemma 9. In either case, we have $h' = \Omega\big(\min(h_0, W/(\sigma r))\big) = \Omega\big(\min(h_0, Z/(\sigma r))\big)$.

Execution of each maximal subproblem visits $w'h'$ spacetime points incurring $O(w')$ cache misses. Hence, the ratio of useful work to cache misses

for the execution of the subproblem is $h' = \Omega\big(\min(h_0, Z/(\sigma r))\big)$. Thus, the same ratio holds for the entire execution of $\mathcal{T}$ which, therefore, incurs at most

$$\frac{wh}{\Omega\big(\min(h_0, Z/(\sigma r))\big)}$$

cache misses, from which the theorem follows. Q.E.D.

We are now ready to analyze the parallel cache complexity of our cache oblivious stencil algorithm. We first derive the sequential cache complexity of a trapezoid in terms of its area A, which is proportional to the work of the trapezoid. Since the cache complexity turns out to be a concave function of the work, we can then derive the Cilk cache complexity from Corollary 6.

Lemma 11 *Let procedure* `walk1` *traverse a trapezoid of height h_0 on a single processor with an ideal cache of size Z and line size $L = \Theta(1)$. Then each subtrapezoid $\mathcal{T}$ of area A generated by* `walk1` *incurs at most*

$$O\left(\frac{A}{Z/(\sigma r)} + \frac{A}{h_0} + \sqrt{A\sigma r}\right)$$

cache misses, where σ is the slope of the stencil and r is the number of black trapezoids created by the procedure in the space-cut case.

Proof: Let w be the width of $\mathcal{T}$. We first prove that

$$w = O\left(\frac{A}{h_0} + \sqrt{A\sigma r}\right) . \tag{8}$$

From Lemma 8, we have $h = A/w \geq \min\big(h_0, w/(4\sigma(r+1))\big)$. Depending on which of the two terms is smaller, we have two cases. If $h_0 \leq w/(4\sigma(r+1))$, then we have $A/w \geq h_0$. Consequently, we have $w \leq A/h_0$, which proves Eq. (8). Otherwise, we have $A/w \geq w/(4\sigma(r+1))$, and thus $w^2 \leq 4A\sigma(r+1)$, again proving Eq. (8).

The lemma then follows by substituting Eq. (8) in Theorem 10. Q.E.D.

Theorem 12 (Parallel cache complexity) *Assume a Cilk scheduler, an ideal distributed-cache machine with P processors and private caches of size Z and line size $L = \Theta(1)$, and memory consistency maintained by the Backer protocol. Let procedure* `walk1` *traverse a trapezoid of width w_0 and height h_0. Let σ be the slope of the stencil and r be the number of black trapezoids created by the procedure in the space-cut case. Then, the execution of the procedure incurs*

$$O\left(\frac{w_0 h_0}{Z/(\sigma r^2)} + r w_0 + \sigma h_0 \sqrt{P r^3 w_0^{1+\alpha}}\right)$$

cache misses with high probability, where $\alpha = 1/\lg(2(r-1))$.

Proof: Consider the trace of the execution with the recursive decomposition consisting of all segments corresponding to trapezoids completely executed by the procedure. We identify the length of a segment in the decomposition with the area of the trapezoid. Then, from Lemma 11, the cache complexity of a segment $\mathcal{B}$ in the recursive decomposition is bounded by $Q(\mathcal{B}) \leq f(|\mathcal{B}|)$, for some nondecreasing concave function f such that

$$f(A) \in O\left(\frac{A}{Z/(\sigma r)} + \frac{A}{h_0} + \sqrt{A\sigma r}\right) .$$

The critical path is $T_\infty = O\left(\sigma r h_0 w_0^\alpha\right)$, as proven in Theorem 7. The theorem then follows from Corollary 6. Q.E.D.

Remark: Practical instances of procedure `walk1` operate with a constant value σ, and a relatively large constant value r, such that $\alpha = 1/\lg(2(r-1)) = \epsilon$, where ϵ is a "small" constant. Then, the cache complexity of procedure `walk1` applied to a trapezoid of width and height n is with high probability

$$Q_P(n, Z) = O\big(n^2/Z + n + \sqrt{Pn^{3+\epsilon}}\big) . \tag{9}$$

4.3 Linear System Solver

In this section, we present and analyze a multithreaded cache oblivious linear system solver based upon Gaussian elimination without pivoting followed by back substitution.

Unlike the multithreaded cache oblivious algorithm for Gaussian elimination presented by Blumofe et al. [7], which employs mutually recursive procedures for LU decomposition, triangular solve, and Schur's complement, our algorithm consists of a single recursive procedure. In this respect, our algorithm is similar to the Gaussian Elimination Paradigm of Chowdhury and Ramachandran [12], but more general because it handles rectangular matrices of arbitrary integral size. Toledo [29] studies the related but harder problem of cache oblivious Gaussian elimination with pivoting.

4.3.1 Description of the Gaussian Elimination

Our solver can handle multiple right-hand sides simultaneously by solving the system $CX = B$, where C is an $N \times N$ matrix and B consists of

M column vectors, each corresponding to one right-hand side associated with one of M solution vectors. We store both matrices C and B in a $N \times (N+M)$ matrix A by concatenating the columns of B to the columns of C. Both the Gaussian elimination and the back substitution are implemented in place, and the solution X can be found in A in place of the right hand side B.

The Gaussian elimination shown in Fig. 7 consists of procedure `walk2`, which could be seen as variation of the 2D spacetime traversal for stencil computations specialized for $\sigma = 0$, and of the kernel procedure `gauss`. Except for the order in which it visits points in (k, i, j)-space, our algorithm is equivalent to the familiar iterative Gaussian elimination expressed as triply nested loop [20, Section 3.2]:

```
for (k=0; k<N-1; k++)
  for (i=k+1; i<N; i++)
    for (j=k; j<N+M; j++)
      gauss(k, i, j);
```

Here, variable k indexes the pivot element, and i and j traverse the submatrix in the lower right corner of A for the update operation.

Procedure `walk2` in Fig. 7 traverses all points (k, i, j) such that $k_0 \le k < k_1$, $i_0 + di_0\,(k - k_0) \le i < i_1$, and $j_0 + dj_0\,(k - k_0) \le j < j_1$. Unlike in the stencil algorithm, the slopes of the upper bounds are always zero and we do not represent them explicitly. To make procedure `walk2` equivalent to the iterative Gaussian elimination, we invoke it as shown in Fig. 8.

We now argue informally that procedure `walk2` is equivalent to the iterative Gaussian elimination. To establish correctness, we must argue that procedure `walk2` traverses the same (k, i, j)-space as the iterative procedure, and that the traversal occurs in an order that respects the kernel dependencies.

To see that the (k, i, j)-space is the same as the iterative procedure, observe first that this property holds trivially in the base case in lines 30–33 and that this property is preserved by the k-cut in lines 25–28 assuming inductively that the property holds for the two recursive calls. The i-cut in lines 15–18 partitions the (k, i, j) into two disjoint subspaces, which are not self-intersecting because of the condition $\Delta i \ge 2 * \Delta k$, which is analogous for the condition for the space cut in the stencil case. A similar argument applies to the j-cut in lines 20–23.

To see that the procedure respects the dependencies of the `gauss` kernel, observe that $i \ge k$ and $j \ge k$ hold by construction, and that the kernel requires that point (k, i, j) depend upon points $(k-1, i, j)$, (k, k, j), and (k, i, k). The first dependency is enforced by the `sync` statement in line 27, which, for fixed i and j, guarantees that the procedure visits points (k, i, j)

```
0   double A[N][N+M];
1   void (*kernel)(int k, int i, int j);

2   void gauss(int k, int i, int j)
3   {
4      if (j == k)
5         A[i][j] /= A[k][k];
6      else
7         A[i][j] -= A[i][k] * A[k][j];
8   }

9   cilk void walk2(int k0, int k1,
10                  int i0, int di0, int i1,
11                  int j0, int dj0, int j1)
12  {
13     int Δk = k1 - k0, Δi = i1 - i0, Δj = j1 - j0;

14     if (Δi >= 2*Δk && Δi >= Δj && Δi > 1) {
15        int im = (i0+i1)/2;
16        spawn walk2(k0, k1, i0, di0, im, j0, dj0, j1);
17        if (i0 < k1) sync;
18        spawn walk2(k0, k1, im, 0, i1, j0, dj0, j1);
19     } else if (Δj >= 2*Δk && Δj > 1) {
20        int jm = (j0+j1)/2;
21        spawn walk2(k0, k1, i0, di0, i1, j0, dj0, jm);
22        if (j0 < k1) sync;
23        spawn walk2(k0, k1, i0, di0, i1, jm, 0, j1);
24     } else if (Δk > 1) {
25        int k2 = Δk/2;
26        spawn walk2(k0, k0+k2, i0, di0, i1, j0, dj0, j1);
27        sync;
28        spawn walk2(k0+k2, k1, i0+di0*k2, di0, i1, j0+dj0*k2, dj0, j1);
29     } else if (Δk == 1) {
30        int i, j;
31        for (i = i0; i < i1; i++)
32           for (j = j0; j < j1; j++)
33              kernel(k0, i, j);
34     }
35  }
```

Figure 7: Multithreaded cache oblivious Gaussian elimination

```
kernel = gauss;
spawn walk2(0, N-1,             /* k0, k1 */
            1, 1, N,            /* i0, di0, i1 */
            0, 1, N+M);         /* j0, dj0, j1 */
sync;
```

Figure 8: How to invoke procedure `walk2` from Fig. 7 to compute Gaussian elimination.

in ascending order of k. The second dependency is enforced by the `sync` statement in line 17. If $i_0 \geq k_1$, then if (k, i, j) is in the region traversed by the procedure, then (k, k, j) is not in the region. Thus, the second dependency holds vacuously and it is safe to execute the two subproblems in parallel. Otherwise $i_0 < k_1$, and the procedure conservatively respects the dependency by traversing the two subproblems in ascending order of i. A symmetric argument holds for the third dependency in the j-cut case.

4.3.2 Work and Critical Path of Gaussian Elimination

We now determine the work, critical path, and sequential cache complexity of procedure `walk2`. Then we apply Corollary 6 to derive its parallel cache complexity.

Let $\Delta k = k_1 - k_0$, $\Delta i = i_1 - i_0$, and $\Delta j = j_1 - j_0$. We state without proof that the work T_1 of the procedure is

$$T_1 = O(\Delta k \Delta i \Delta j) \ .$$

The analysis of the critical path is more involved. One way to attack the problem is to recognize that, depending on the parameters, procedure `walk2` computes an LU decomposition, the solution of a triangular linear system, or a Schur's complement, and then apply the analysis from [7] to conclude that the critical path is $T_\infty = O(N \lg^2 N)$ when $M = 0$. Another possibility would be to coerce procedure `walk2` into the Parallel Gaussian Elimination Paradigm [12], leading to the same conclusion when $M = 0$. However, since these reductions would force us to multiply entities beyond necessity and they would anyway be limited to a special case, we analyze the critical path of procedure `walk2` directly in Theorem 14 for general M.

Lemma 13 *When procedure `walk2` is invoked as in Fig. 8, these invariants hold:*

1. $i_0 \geq k_0$.

2. $j_0 \geq k_0$.

3. If $di_0 = 0$, then $i_0 \geq k_1$.

4. If $dj_0 = 0$, then $j_0 \geq k_1$.

Proof: Invariants 1 and 2 hold by construction, because the initial conditions specify a region of (k, i, j) space such that $i \geq k$ and $j \geq k$.

Invariant 3 trivially holds initially because $di_0 = 1$. The recursive calls in lines 16, 21, and 23 trivially preserve the invariant. In the recursive call in line 26, we have $k_0 + k_2 \leq k_1$, and the invariant is preserved. In the recursive call in line 28, we have $i_0 + di_0\, k_2 \geq i_0$, and the invariant is preserved. We are left to prove that $i_m \geq k_1$ in line 18. We have $2(i_m - k_1) \geq i_0 + i_1 - 2k_1 \geq 2i_0 + \Delta i - 2k_1 \geq 2i_0 + 2\Delta k - 2k_1 = 2(i_0 - k_0) \geq 0$, where the last inequality follows from Invariant 1.

The proof of Invariant 4 is the same as Invariant 3 after swapping i and j. Q.E.D.

Theorem 14 *When procedure `walk2` is invoked as in Fig. 8, its critical path is*

$$T_\infty = O(N \lg N \lg(N + M)) \ .$$

Proof: Recall Iverson's APL notation [22, Section 2.1]:

$$[\mathcal{A}] = \begin{cases} 1 & \text{if } \mathcal{A} \text{ is true;} \\ 0 & \text{otherwise.} \end{cases}$$

We now prove that, when procedure `walk2` is invoked as in Fig. 8, a constant $c \geq 1$ exists such that the critical path is

$$\begin{aligned} &T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) \leq \\ &\quad c(2\Delta k - 1 + \lg \Delta i + \lg \Delta j)(1 + [i_0 < k_1] \lg(\Delta i))(1 + [j_0 < k_1] \lg(\Delta j)) \ , \end{aligned} \tag{10}$$

for all sufficiently large values of Δk, Δi, and Δj, where $\Delta k = k_1 - k_0$, $\Delta i = i_1 - i_0$, and $\Delta j = j_1 - j_0$.

The proof is by well-founded induction on the triple $(\Delta k, \Delta i, \Delta j)$ under the product order: $(\Delta k, \Delta i, \Delta j) \leq (\Delta k', \Delta i', \Delta j')$ iff $\Delta k \leq \Delta k'$, $\Delta i \leq \Delta i'$, and $\Delta j \leq \Delta j'$.

Base case: If $(\Delta k, \Delta i, \Delta j) \leq (1, 2, 2)$, a constant $c \geq 1$ can be found such that Eq. (10) holds.

Inductive step: Otherwise, depending on the relative values of Δk, Δi, and Δj, the procedure recursively cuts one dimension in half. We have therefore three cases.

***i*-cut:** (lines 15–18) We distinguish two subcases, depending upon whether the `sync` statement in line 16 is executed or not.

If $i_0 < k_1$, then the `sync` statement is executed and the critical path obeys the relation

$$T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) \leq T_\infty(k_0, k_1, i_0, i_m, j_0, j_1) + T_\infty(k_0, k_1, i_m, i_1, j_0, j_1) + 1 \,,$$

where the constant 1 accounts for the critical path of the constant number of instructions executed by the procedure excluding the recursive calls.

By Lemma 13, $i_0 \geq k_1$ holds inside the recursive call in line 18. Thus we have

$$\begin{aligned} &T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) \\ &\quad \leq c(2\Delta k - 1 + \lg(\Delta i/2) + \lg \Delta j)(1 + \lg(\Delta i/2))(1 + [j_0 < k_1] \lg(\Delta j)) + \\ &\qquad c(2\Delta k - 1 + \lg(\Delta i/2) + \lg \Delta j)(1 + [j_0 < k_1] \lg(\Delta j)) + 1 \\ &\quad \leq c(2\Delta k - 1 + \lg(\Delta i/2) + \lg \Delta j)(1 + \lg \Delta i)(1 + [j_0 < k_1] \lg(\Delta j)) + 1 \\ &\quad \leq c(2\Delta k - 1 + \lg(\Delta i) + \lg \Delta j)(1 + \lg \Delta i)(1 + [j_0 < k_1] \lg(\Delta j)) \,, \end{aligned}$$

where the last inequality holds because $c \geq 1$.

If $i_0 \geq k_1$, then the `sync` statement is not executed and the critical path obeys the relation

$$T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) \leq \max(T_\infty(k_0, k_1, i_0, i_m, j_0, j_1), T_\infty(k_0, k_1, i_m, i_1, j_0, j_1)) + 1 \,.$$

Thus we have

$$\begin{aligned} T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) &\leq c(2\Delta k - 1 + \lg(\Delta i/2) + \lg \Delta j)(1 + [j_0 < k_1] \lg(\Delta j)) + 1 \\ &\leq c(2\Delta k - 1 + \lg \Delta i + \lg \Delta j)(1 + [j_0 < k_1] \lg(\Delta j)) \,, \end{aligned}$$

where the last inequality holds because $c \geq 1$.

Thus, the inductive step is proven for both the $i_0 \leq k_1$ and the $i_0 > k_1$ cases of the i-cut.

***j*-cut:** (lines 20–23) This case is the same as the i-cut swapping the roles of i and j.

k**-cut:** (lines 25–28) The critical path obeys the relation

$$T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) \le T_\infty(k_0, k_m, i_0, i_1, j_0, j_1) + T_\infty(k_m, k_1, i_1, i_1, j_0, j_1) + 1 ,$$

where the constant 1 accounts for the critical path of the constant number of instructions executed by the procedure excluding the recursive calls.

Thus we have

$$\begin{aligned} & T_\infty(k_0, k_1, i_0, i_1, j_0, j_1) \\ & \quad \le 2c(\Delta k - 1 + \lg \Delta i + \lg \Delta j)(1 + [i_0 < k_1] \lg \Delta i)(1 + [j_0 < k_1] \lg(\Delta j)) + 1 \\ & \quad \le c(2\Delta k - 1 + \lg \Delta i + \lg \Delta j)(1 + [i_0 < k_1] \lg \Delta i)(1 + [j_0 < k_1] \lg(\Delta j)) , \end{aligned}$$

where the last inequality holds because $c \ge 1$.

Because Eq. (10) holds in the base case and in all the inductive cases, the theorem is proven. Q.E.D.

4.3.3 Cache Complexity of Gaussian Elimination

Lemma 15 (Aspect ratio) *If procedure* `walk2` *is invoked with initial parameters* Δk_0*,* Δi_0*, and* Δj_0*, then for all recursive subproblems the following invariants hold:*

$$\Delta k \ge \min(\Delta k_0, \Omega(\max(\Delta i, \Delta j))) ; \tag{11a}$$

$$\Delta i \ge \min(\Delta i_0, \Omega(\max(\Delta k, \Delta j))) ; \tag{11b}$$

$$\Delta j \ge \min(\Delta j_0, \Omega(\max(\Delta k, \Delta i))) . \tag{11c}$$

Proof: The invariants are true initially, and the procedure always cuts the dimension corresponding to the largest of $2\Delta k$, Δi, or Δj. Q.E.D.

Lemma 16 (Sequential cache complexity) *Let procedure* `walk2` *be invoked with initial parameters* Δk_0*,* Δi_0*, and* Δj_0*, on a single processor with an ideal cache of size* Z *and line size* $L = \Theta(1)$*. Then each subproblem generated by the procedure incurs at most*

$$O\left(V^{2/3} + V\left(\frac{1}{\sqrt{Z}} + \frac{1}{\Delta k_0} + \frac{1}{\Delta i_0} + \frac{1}{\Delta j_0}\right)\right)$$

cache misses, where $V = \Delta k \Delta i \Delta j$*.*

Proof: Procedure `walk2` visits a subsequence of the (k, i, j)-space visited by the cache oblivious matrix multiplication procedure from [16], and thus the sequential cache complexity of each subproblem is [16, Theorem 1]

$$O\left(\Delta k \Delta i + \Delta k \Delta j + \Delta i \Delta j + \frac{V}{\sqrt{Z}}\right) .$$

We now prove that $\Delta k \Delta i = O(V^{2/3} + V/\Delta j_0)$. From Lemma 15 Eq. (11c), we have that either $\Delta j \geq \Delta j_0$ or $\Delta j = \Omega(\max(\Delta k, \Delta i))$. If $\Delta j \geq \Delta j_0$ then $\Delta k \Delta i = V/\Delta j \leq V/\Delta j_0$ and we are done. Otherwise, $(\Delta k \Delta i)^2 = V^2/\Delta j^2 \leq O(V^2/(\Delta k \Delta i))$, and therefore $\Delta k \Delta i = O(V^{2/3})$.

A similar argument proves that $\Delta k \Delta j = O(V^{2/3} + V/\Delta i_0)$ and that $\Delta i \Delta j = O(V^{2/3} + V/\Delta k_0)$, from which the lemma follows. Q.E.D.

Theorem 17 (Parallel cache complexity) *We triangularize a system of N linear equations in N unknowns with M right-hand sides by Gaussian elimination. Assume a Cilk scheduler, an ideal distributed-cache machine with P processors and private caches of size Z and line size $L = \Theta(1)$, and memory consistency maintained by the Backer protocol. Let procedure* `walk2` *traverse the (k, i, j)-space with volume $V = O(N^2(N + M))$. Then, the execution of the Gaussian elimination procedure incurs*

$$O\left(\left(V^2 S\right)^{1/3} + V\left(\frac{1}{\sqrt{Z}} + \frac{1}{N} + \frac{1}{N+M}\right)\right)$$

cache misses with high probability, where $S = O(PN \lg N \lg(N + M))$.

Proof: Procedure `walk2` traverses the (k, i, j)-space with dimensions $\Delta k_0 = N$, $\Delta i_0 = N$, and $\Delta j_0 = N + M$ and volume $V = \Delta k_0 \Delta i_0 \Delta j_0 = O(N^2(N + M))$. The number of cache misses is proportional to the work associated with the (k, i, j)-space, which is proportional to its volume V. According to Lemma 16 and for $L = \Theta(1)$, the sequential cache complexity of each subproblem of the recursive procedure `walk2` is a concave function $f(V) \in O(V^{2/3} + V(1/\sqrt{Z} + 1/N + 1/(N + M))$. Hence, the theorem follows from Corollary 6, with the critical path length given by Theorem 14. Q.E.D.

Remark: For the degenerate cases $M = 0$ (LU decomposition without right-hand side) and $M = 1$ (single right-hand side), the parallel cache complexity of the Gaussian elimination is

$$Q_P(N, Z) = O\left(\frac{N^3}{\sqrt{Z}} + \left(PN \lg^2 N\right)^{1/3} N^2\right) . \qquad (12)$$

4.3.4 Description of the Back-Substitution

Given an upper triangular matrix, we can apply a column-oriented back-substitution [20, Section 3.1.3] to solve a linear system of equations. For multiple right-hand sides the iterative, column-oriented back-substitution consists of the triply nested loop:

```
for (k=N-1; k>=0; k--) {
  for (j=N; j<N+M; j++) {   /* for each right-hand side */
    A[k][j] /= A[k][k];
    for (i=k-1; i>=0; i--)
      A[i][j] -= A[i][k] * A[k][j];
  }
}
```

We implement a multithreaded cache oblivious version of the back-substitution by reusing traversal procedure `walk2` of Fig. 7. The new kernel procedure `backsub` is shown in Fig. 9, and the invokation of `walk2` to compute the back-substitution is shown in Fig. 10.

```
0   void backsub(int k', int i', int j)
1   {
2     int k = N-1-k';
3     int i  = N-1-i';
4     if (i == k)
5       A[i][j] /= A[k][k];
6     else
7       A[i][j] -= A[i][k] * A[k][j];
8   }
```

Figure 9: Kernel for multithreaded cache oblivious back substitution.

```
kernel = backsub;
spawn walk2(0, N,                /* k0, k1 (reversed) */
            0, 1, N,             /* i0, di0, i1 (reversed) */
            N, 0, N+M);          /* j0, dj0, j1 */
sync;
```

Figure 10: How to invoke procedure `walk2` from Fig. 7 to compute the back-substitution.

We can reuse procedure `walk2` by observing that the the back-substitution and the Gaussian elimination traverse similar iteration spaces. The iterative version traverses the space (k, i, j) such that $N > k \geq 0$, $N \leq j < N+M$, and $k > i \geq 0$. As shown in Fig. 10, we set $\Delta i_0 = 1$ and $\Delta j_0 = 0$, so

that procedure `walk2` invokes the `backsub` kernel with parameters (k', i', j) such that $0 \le k' < N$, $k' \le i' < N$, and $N \le j < N + M$. With the substitutions $k = N - 1 - k'$ and $i = N - 1 - i'$ in the `backsub` kernel, the iteration spaces of `walk2` and of the iterative routine are the same.

We state without proof that the work T_1 of the back-substitution is

$$T_1 = O(N^2 M) \ .$$

The critical path T_∞ of the back-substitution is similar to that of the Gaussion elimination determined in Theorem 14.

Theorem 18 *When procedure `walk2` is invoked as in Fig. 10, its critical path is*

$$T_\infty = O((N + \lg M) \lg N) \ .$$

Proof: Apply Eq. (10) from the proof of Theorem 14 with $\Delta k = N$, $\Delta i = N$, and $\Delta j = M$ according to the parameters of Fig. 10, and note that with these parameters we have $[j_0 < k_1] = 0$. Q.E.D.

4.3.5 Cache Complexity of the Back-Substitution

Theorem 19 (Parallel cache complexity) *Assume a Cilk scheduler, an ideal distributed-cache machine with P processors and private caches of size Z and line size $L = \Theta(1)$, and memory consistency maintained by the Backer protocol. Let procedure `walk2` traverse the (k, i, j)-space with volume $V = O(N^2 M)$. Then, the execution of the back-substitution procedure incurs*

$$O\left(\left(V^2 S\right)^{1/3} + V \left(\frac{1}{\sqrt{Z}} + \frac{1}{N} + \frac{1}{M} \right) \right)$$

cache misses with high probability, where $S = O(P(N + \lg M) \lg N)$.

Proof: Analogous to the proof of Theorem 17, the theorem is a consequence of Corollary 6 and the critical path length from Theorem 18. Q.E.D.

Remark: For a single right-hand side, $M = 1$, the parallel cache complexity of the back-substitution is

$$Q_P(N, Z) = O\left(N^2 + \left(P N^2 \lg N\right)^{1/3} N \right) \ . \qquad (13)$$

Because in this case the algorithm performs $\Theta(\mathrm{N}^2)$ work on $\Theta(\mathrm{N}^2)$ input elements, the cache is asymptotically useless. The algorithm benefits from the cache when solving for multiple right-hand sides, however.

4.4 Longest Common Subsequence

In this section, we analyze a multithreaded algorithm for computing the length of a longest common subsequence (LCS) of two sequences.

The length computation is a subproblem of the problem of computing a LCS of two sequences, and its textbook solution is a dynamic programming algorithm [13] that could in principle be formulated as a 1D-stencil computation. The stencil formulation is inconvenient, however, if one wishes to use the length computation as a subroutine in Hirschberg's algorithm [23] for computing the LCS. Here, we present a more direct Cilk program for computing the length of the LCS which can easily be incorporated into Hirschberg's algorithm. Our program is derived from the cache oblivious sequential algorithm of Chowdhury and Ramachandran [11], and its parallelization is based upon [25].

4.4.1 Description of the Longest Common Subsequence Algorithm

Cilk procedure `lcslen` in Fig. 11 computes the length of the longest common subsequence of two sequences, x of length $M-1$ and y of length $N-1$. The first sequence is stored in array x at index range $[1,\ldots,M-1]$, and the second in array y at index range $[1,\ldots,N-1]$. To compute the length of the longest common subsequence, initialize array c with zeros, and invoke procedure `lcslen` with the following arguments:

```
spawn lcslen(1, M, 1, N);  sync;
```

The work of the length computation is $T_1 = O(MN)$.

To compute the critical path and cache complexity, we assume for simplicity w.l.o.g. that $N \geq M$. The length of the critical path is $T_\infty = O(M^{\lg 3} N/M)$, as can be seen from the following argument.[5] Procedure `lcslen` splits both dimensions in half until the base case is reached, which occurs when the shorter of the two dimensions is 1. Because M is the shorter dimension by assumption the recursion depth is $\lg M$. The critical path of the base case is $O(N/M)$, corresponding to the rectanglar tableau of width N/M and height 1. Thus, the critical path obeys the recurrence

$$T_\infty(M) = \begin{cases} N/M, & \text{if } M = 1, \\ 3T_\infty(M/2) + \Theta(1), & \text{otherwise}, \end{cases}$$

which has the stated result.

[5] Shorter critical paths are possible by partitioning the tableau into more than just four quadrants [25].

```
0   int x[M], y[N];                       /* sequences */
1   int c[M+N-1];                         /* length array */

2   #define C(I,J)   c[(I)-(J)+N]     /* index projection into c */

3   cilk void lcslen(int i0, int i1, int j0, int j1)
4   {
5     int Δi = i1 - i0, Δj = j1 - j0;

6     if (Δi <= 1 || Δj <= 1) {          /* base case */
7       int i, j;
8       for (i = i0; i < i1; i++) {
9         for (j = j0; j < j1; j++) {
10          if (x[i] == y[j])
11            C(i,j) = C(i-1,j-1) + 1;
12          else
13            C(i,j) = MAX(C(i-1,j), C(i,j-1));
14        }
15      }
16    } else {                            /* recursion */
17      int im = (i0+i1)/2;
18      int jm = (j0+j1)/2;
19      spawn lcslen(i0, im, j0, jm);          /* Q0 */
20      sync;
21      spawn lcslen(i0, im, jm, j1);          /* Q1 */
22      spawn lcslen(im, i1, j0, jm);          /* Q2 */
23      sync;
24      spawn lcslen(im, i1, jm, j1);          /* Q3 */
25    }
26  }
```

Figure 11: Multithreaded cache oblivious length computation of a longest common subsequence computation.

We maintain the length values, which the naive algorithm [13] stores in a $M \times N$ tableau, in array c of length $M+N-1$. Intuitively, array c covers one row and one column of the $M \times N$ dynamic programming tableau spanned by the two sequences x and y. In particular, in the initial state, we assume that array c stores length value 0 in each element, and serves as the top row and leftmost column of the tableau. In the final state, array c contains the lengths of subsequences associated with the bottom row and rightmost column of the tableau. The 3-point stencil of the tableau prescribes the data dependencies such that point $C(i, j)$ depends on points $C(i-1, j-1)$, $C(i-1, j)$, and $C(i, j-1)$, cf. base case in Fig. 11.

4.4.2 Cache Complexity of the Longest Common Subsequence Algorithm

Chowdhury and Ramachandran [11] introduced a cache oblivious version of the longest common subsequence computation. They have shown that the sequential execution of the length computation has cache complexity $Q(M, N, Z) = O(MN/Z + M + N)$, where we assume that the cache line size $L = \Theta(1)$. This cache complexity is asymptotically optimal, and their result holds under the assumption of an ideal cache. Their analysis also applies to the sequential execution of procedure `lcslen` in Fig. 11.

The multithreaded version of the length computation in Fig. 11 consists of four subproblems corresponding to tableau quadrants $\mathcal{Q}_0$, $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$. Hence, the recursive computation of the quadrants produces a 4-recursive decomposition of the trace of procedure `lcslen`, which consists of four complete subtrees of the call tree, each of which incurs $O(MN/Z+M+N)$ cache misses on subproblems of size $M \times N$. The synchronization before and after the computations of quadrants $\mathcal{Q}_1$ and $\mathcal{Q}_2$ constrains the concatenation of the subsegments, yet does not affect the applicability of Corollary 6.

To bound the number of cache misses Q, let $w = MN$ be the length of a segment. Then $Q \leq f(w)$ for some concave function $f(w) \in O(w/Z + w^{1/2})$. For our 4-recursive decomposition, the cache complexity of the parallel execution of `lcslen` is according to Corollary 6

$$Q_P(M, N, Z) = O\big(MN/Z + S^{1/2}(M+N) + S\big) \ , \tag{14}$$

where $S = O(PM^{\lg 3}N/M))$ with high probability.

Remark: The sequential cache complexity of procedure `lcslen` for square problems of size $n = M = N$ is $Q(n, Z) = O(n^2/Z + n)$, and the Cilk cache

complexity is

$$Q_P(n, Z) = O\left(n^2/Z + \sqrt{P}n^{3.58}\right) , \tag{15}$$

if we approximate $2 + \lg 3 \approx 3.58$.

5 Conclusion

We presented a technique for analyzing the cache complexity of multithreaded cache oblivious algorithms on an idealized parallel machine.

While our technique yields stronger upper bounds than previously known, our bounds are not optimal. For example, the matrix multiplication procedure in Fig. 1 can be scheduled statically by partitioning the trace into $S = P^{3/2}$ segments, each computing a matrix multiplication of size $(n/\sqrt{P}) \times (n/\sqrt{P})$, thereby yielding cache complexity $O(n^3/\sqrt{Z} + \sqrt{P}n^2)$, which is lower than the bound $O(n^3/\sqrt{Z} + \sqrt{nP}n^2)$ that we derived for the work-stealing scheduler. Similarly, a matrix multiplication procedure that uses temporary arrays [7] has a shorter critical path and can be scheduled so as to incur $O(n^3/\sqrt{Z} + \sqrt[3]{P}n^2)$ cache misses. For the one-dimensional stencil computation, we conjecture that a smart scheduler should incur no more than $O(n^2/Z + Pn^{1+\epsilon})$ cache misses. Experiments suggest that these three expressions for the cache complexity constitute a more accurate model of the work-stealing scheduler than our upper bounds in Eqs. (4) and (9), but we lack proof that this is the case.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The Data Locality of Work Stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] Laszlo A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent Cache-Oblivious B-Trees. In *17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 228–237, Las Vegas, NV, July 2005.

[4] Gianfranco Bilardi and Franco P. Preparata. Upper Bounds to Processor-Time Tradeoffs Under Bounded-Speed Message Propagation.

In *7th Annual ACM Aymposium on Parallel Algorithms and Architectures*, pages 185–194, Santa Barbara, CA, July 1995.

[5] Guy E. Blelloch and Phillip B. Gibbons. Effectively Sharing a Cache Among Threads. In *16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, Barcelona, Spain, June 2004.

[6] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[7] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An Analysis of Dag-Consistent Distributed Shared-Memory Algorithms. In *8th Annunal ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[10] Richard P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[11] Rezaul A. Chowdhury and Vijaya Ramachandran. Cache-Oblivious Dynamic Programming. In *17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, Miami, FL, 2006.

[12] Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*, pages 71–80, New York, NY, USA, 2007. ACM Press.

[13] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[14] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 1993.

[15] Steven J. Fortune and James Wyllie. Parallelism in Random Access Machines. In *10th Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, May 1978.

[16] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–298, New York, USA, October 1999.

[17] Matteo Frigo, Keith H. Randall, and Charles E. Leiserson. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[18] Matteo Frigo and Volker Strumpen. Cache Oblivious Stencil Computations. In *International Conference on Supercomputing*, pages 361–366, Boston, Massachusetts, June 2005.

[19] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? *Theory of Computing Systems*, 32(3):327–359, 1999.

[20] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, 3rd edition, 1996.

[21] Ronald L. Graham. Bounds for Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, 45:1563–1581, November 1966.

[22] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Math.* Addison Wesley, Reading, Massachusetts, 1994.

[23] Daniel S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[24] Jia-Wei Hong and H. T. Kung. I/O Complexity: The Red-Blue Pebbling Game. In *13th Annual ACM Symposium on Theory of Computing*, pages 326–333, Milwaukee, Wisconsin, May 1981.

[25] Charles E. Leiserson. Minicourse on Miltithreaded Programming in Cilk, Lecture 2: Analysis of Cilk Algorithms. Summer School on Language-Based Techniques for Concurrent and Distributed Software at the University of Oregon, 2006.

[26] Girija J. Narlikar and Guy E. Blelloch. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, 1999.

[27] Daniel D. Sleator and Robert E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[28] Volker Strumpen and Matteo Frigo. Software Engineering Aspects of Cache Oblivious Stencil Computations. Technical Report RC24035, IBM Research, August 2006.

[29] Sivan Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM J. Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.

[30] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.