The Cache Complexity of Multithreaded Cache Oblivious Algorithms

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

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, which incurs $O(n^2/Z + n + \sqrt{Pn^{3+\epsilon}})$ cache misses with high probability.

Categories and Subject Descriptors

F.2 [**Theory**]: Analysis of Algorithms and Problem Complexity

General Terms

Algorithms

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 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 [14], although, as a special case, our bounds also apply to a sequential process migrated from one processor to another by an operating system.

This problem has been studied in 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 commonly referred to as (hardware) threads. In this paper, we focus 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, 20]. 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, 18] 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 also give 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. 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

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

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

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 while pathological series-parallel computations exist for which this bound is tight [1], 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 [14], and we give a technique for proving bounds stronger than Eq. (1) for cache oblivious algorithms [14]. 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 most existing cache oblivious algorithms, the cache complexity is indeed bounded by a concave function of the work, and therefore this analysis is applicable. 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.

For example, a multithreaded program for multiplying two $n \times n$ matrices without using additional storage has $T_1 = O(n^3)$, $T_\infty = O(n)$, 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 [16]. Our one-dimensional stencil algorithm for a square spacetime region has $T_1 = O(n^2)$, $T_\infty = O(n)$, and sequential cache complexity $Q_1 = O(n^2/Z + n)$. The multithreaded algorithm, when executed on P processors by the Cilk scheduler has cache complexity $Q_P = O\big(n^2/Z + n + \sqrt{Pn^{3+\epsilon}}\big)$ with high probability.

In Section 2 of this article 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 and stencil computations.

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 [14].

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 [14, 21] 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 data word 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 yet 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 [13] 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 [22] and LogP [12], our model is asynchronous, and processors operate independently most of the time. Like in the QSM model [17], 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 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 $Q(\mathcal{A}) \leq f(|\mathcal{A}|)$ 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

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

PROOF. Let $\mathcal{A}_i$, $0 \leq i < S$ 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 $Q_P(\mathcal{M}) = \sum_{0 \leq i < S} Q(\mathcal{A}_i)$ cache misses. By assumption, we have $\sum_{0 \leq i < S} Q(\mathcal{A}_i) \leq \sum_{0 \leq i < S} f(|\mathcal{A}_i|)$. By Jensen's inequality we have $\sum_{0 \leq i < S} f(|\mathcal{A}_i|) \leq Sf\left(\sum_{0 \leq i < S} |\mathcal{A}_i|/S\right) = Sf(|\mathcal{M}|/S)$, and the theorem follows. $\square$

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, 15] 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 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). □

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 $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_q$ 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. □

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. □

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 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 $Q(\mathcal{A}) \leq g(|\mathcal{A}|)$ for all segments $\mathcal{A}$ of $\mathcal{M}$. The corollary then follows from Theorem 2. □

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 $E[Sf(rT_1/S)] \leq E[S]f(rT_1/E[S])]$, 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 and stencil computations. The parallel stencil computation algorithm was not previously published, and therefore we discuss this algorithm in detail. All applications are programmed in Cilk [15]. A similar analysis could be applied to suitable parallelizations of other cache oblivious algorithms, such as the cache oblivious dynamic programming framework of Chowdhury and Ramachandran [11].

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 $T_\infty = O(n)$.[1]

[1]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, $A_{11}$, $B_{11}$, $C_{11}$);
        spawn matmul($n$/2, $A_{11}$, $B_{12}$, $C_{12}$);
        spawn matmul($n$/2, $A_{21}$, $B_{11}$, $C_{21}$);
        spawn matmul($n$/2, $A_{21}$, $B_{12}$, $C_{22}$);

        sync;

        spawn matmul($n$/2, $A_{12}$, $B_{21}$, $C_{11}$);
        spawn matmul($n$/2, $A_{12}$, $B_{22}$, $C_{12}$);
        spawn matmul($n$/2, $A_{22}$, $B_{21}$, $C_{21}$);
        spawn matmul($n$/2, $A_{22}$, $B_{22}$, $C_{22}$);
    }
}
```

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.

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 [14] 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 [19].

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 $w = n^3$ be the length of the segment. Then $Q \le f(w)$ 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

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

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

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

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

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

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 $n^3/\sqrt{Z} \ge (Pn)^{1/3}n^2$, 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 $n^3/\sqrt{Z} \le (Pn)^{1/3}n^2$, and thus $\sqrt{Z} \ge (Pn)^{-1/3}n$, from which $ZPn \ge (Pn)^{1/3}n^2$ 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.

We remark that the new bound is not optimal, however. For example, a smart scheduler could achieve linear speedup 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 $O(n^3/\sqrt{Z} + \sqrt{P}n^2)$.

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 [16], 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***. In practical implementations of stencils, there is often no need to store the entire spacetime; storing a bounded number of time steps per space point is sufficient. For example, consider a 3-point stencil in 1-dimensional space (2-dimensional spacetime): Because the computation of a point at time t depends only upon three points at time $t-1$, it is sufficient to store two time steps only. Our cache oblivious algorithm applies to such "in-place" computations that allocate and reuse spacetime points for a bounded number of time steps.

For brevity we restrict our discussion to one-dimensional stencils. The algorithm can be extended to stencils with arbitrary dimensions as discussed in [16].

4.2.1 Description of 1D Stencil Algorithm

Procedure walk1 in Fig. 5 visits all points (t, x) in a rectangular spacetime region, where $0 \le t < T$, $0 \le x < N$, and t and x are integers. The procedure visits point $(t+1, x)$ after visiting points $(t, x+k)$ for $|k| \le \sigma$, and thus it respects the dependencies imposed by the stencil. We assume in this paper that $\sigma \ge 1$.[2]

Although we are ultimately interested in traversing rectangular spacetime regions, the procedure does in fact operate on more general trapezoidal regions such as the one

[2]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.

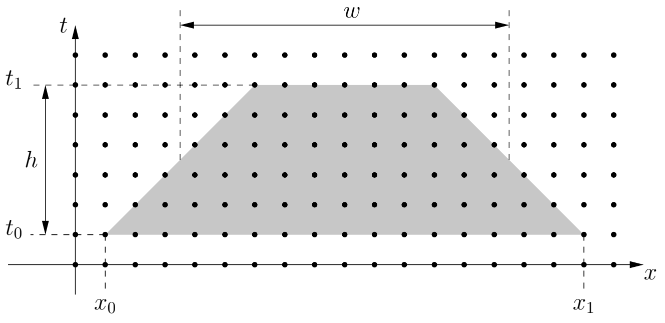

Figure 2: Illustration of the trapezoid $\mathcal{T}(t_0, t_1, x_0, \dot{x}_0, x_1, \dot{x}_1)$ for $\dot{x}_0 = 1$ and $\dot{x}_1 = -1$. The trapezoid includes all points in the shaded region, except for those on the top and right edges.

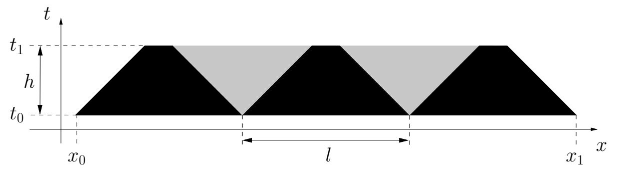

Figure 3: Illustration of a *parallel space cut*. The black trapezoids are independent of each other and can be visited in parallel. Once these trapezoids have been visited, the gray trapezoids can in turn be visited in parallel.

shown in Fig. 2. 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 $t_0 \le t < t_1$ and $x_0 + \dot{x}_0(t - t_0) \le x < x_1 + \dot{x}_1(t - t_0)$. (We use the Newtonian notation $\dot{x} = dx/dt$.) The ***height*** of the trapezoid is $h = t_1 - t_0$, 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: $t_1 \ge t_0$, $x_1 \ge x_0$, and $x_1 + h \cdot \dot{x}_1 \ge x_0 + h \cdot \dot{x}_0$.

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. 3. 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 not depend on points in a gray trapezoid. A black trapezoid is well-defined only if the condition $l \ge 2\sigma h$ holds, or else the trapezoid would be self-intersecting. Therefore, r black trapezoids fit into $\mathcal{T}$ only if $x_1 - x_0 \ge 2r\sigma h$, 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. 3, and two are located at the left and right edges of $\mathcal{T}$. In Fig. 3, 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.

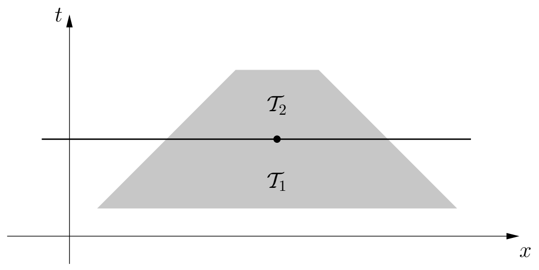

Figure 4: Illustration of a *time cut*. The algorithm cuts the trapezoid along the horizontal line through its center, it recursively visits $\mathcal{T}_1$, and then it visits $\mathcal{T}_2$.

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. 4. 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 space-time points (t_0, x) with $x_0 \le x < x_1$. 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 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 $T_1(w, h) \le 2rT_1(w/(2r), h) + O(h)$ in case of a space cut, and bound $T_1(w, h) \le 2T_1(w, h/2) + O(1)$ in case of a time cut. One can verify by induction that $T_1(w, h) \le c(wh - w - h)$ 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.

```
0   int $r$;  /* assert($r$ >= 2) */

1   cilk void walk1(int $t_0$, int $t_1$, int $x_0$, int $\dot{x}_0$, int $x_1$, int $\dot{x}_1$)
2   {
3      int $h$ = $t_1$ - $t_0$, $\Delta x$ = $x_1$ - $x_0$;
4      int $x$, i;

5      if ($h$ >= 1 && $\Delta x$ >= 2 * $\sigma$ * $h$ * $r$) {     /* parallel space cut */
6          int $l$ = $\Delta x$ / $r$;     /* base of a black trapezoid, rounded down */

7          for (i = 0; i < $r$ - 1; ++i)
8               spawn walk1($t_0$, $t_1$, $x_0$ + i * $l$, $\sigma$, $x_0$ + (i+1) * $l$, -$\sigma$);
9          spawn walk1($t_0$, $t_1$, $x_0$ + i * $l$, $\sigma$, $x_1$, -$\sigma$);

10         sync;

11         spawn walk1($t_0$, $t_1$, $x_0$, $\dot{x}_0$, $x_0$, $\sigma$);
12         for (i = 1; i < $r$; ++i)
13              spawn walk1($t_0$, $t_1$, $x_0$ + i * $l$, -$\sigma$, $x_0$ + i * $l$, $\sigma$);
14         spawn walk1($t_0$, $t_1$, $x_1$, -$\sigma$, $x_1$, $\dot{x}_1$);
15     } else if ($h$ > 1) {                        /* time cut */
16         int $s$ = $h$ / 2;
17         spawn walk1($t_0$, $t_0$ + $s$, $x_0$, $\dot{x}_0$, $x_1$, $\dot{x}_1$);
18         sync;
19         spawn walk1($t_0$ + $s$, $t_1$, $x_0$ + $\dot{x}_0$ * $s$, $\dot{x}_0$, $x_1$ + $\dot{x}_1$ * $s$, $\dot{x}_1$);
20     } else if ($h$ == 1) {                       /* base case */
21         for ($x$ = $x_0$; $x$ < $x_1$; ++$x$)
22             kernel($t_0$, $x$);
23     }
24  }
```

Figure 5: One-dimensional parallel stencil algorithm implemented in the Cilk language. In lines 7–9, we spawn r black trapezoids. Because of the rounding of l in line 6, the length of the base of the last trapezoid is not necessarily l, and we handle this trapezoid separately in line 9. The `sync` statement in line 10 waits for the black trapezoids to complete, before spawning the gray trapezoids in lines 11–14.

The critical path of procedure `walk1` is $T_\infty = O(\sigma r h w^{1/\lg(2(r-1))})$. Appendix A contains the laborious yet straightforward proof.

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 7 (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.*

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, $\Delta x = x_1 - x_0 \leq 2\sigma r h$ 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. □

Lemma 8 (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 the constant hidden in the Ω-notation does not depend upon r.*

Proof. Before applying a space cut to a trapezoid of width w and height h, Lemma 7 holds. The space cut produces trapezoids of width $w' = \Theta(w/r)$ and of the same height h, and therefore we have $h \geq \min(h_0, \Omega(w'/\sigma))$. □

Theorem 9 (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.

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 7. 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 8. 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 $w'h'$ space-time 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. □

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 10. *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.

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

$$w = O\left(\frac{A}{h_0} + \sqrt{A\sigma r}\right) . \qquad (7)$$

From Lemma 7, 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. (7). Otherwise, we have $A/w \geq w/(4\sigma(r+1))$, and thus $w^2 \leq 4A\sigma(r+1)$, again proving Eq. (7).

The lemma then follows by substituting Eq. (7) in Theorem 9. □

THEOREM 11 (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. Then the execution 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 10, 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

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

With critical path $T_\infty = O\left(\sigma r h_0 w_0^\alpha\right)$, as proven in Theorem 12 in Appendix A, the theorem follows from Corollary 6. □

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

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

5. Conclusion

We believe that programs that can cope with varying memory hierarchies as exemplified by cache oblivious algorithms, a provably efficient scheduler such as Cilk, and a scalable shared memory as advocated by the dag consistent memory model, provide a framework in which we can obtain scalable and reasonably high performance with reasonable programming effort. This framework suggests a scalable computer architecture based on processors with private caches as part of an automatically managed memory hierarchy. The theoretical results of this paper provide further support for this hypothesis.

6. REFERENCES

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

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

[3] M. A. Bender, J. T. Fineman, S. Gilbert, and B. 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] G. Bilardi and F. 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] G. E. Blelloch and P. 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] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[7] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. 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] 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 *5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995.

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

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

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

[12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. 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.

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

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache Oblivious Algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–298, New York, USA, Oct. 1999.

[15] M. Frigo, K. H. Randall, and C. E. Leiserson. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

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

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

[18] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

[19] J.-W. 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.

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

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

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

APPENDIX

A. Analysis of the Critical Path of Walk1

THEOREM 12. *The critical path of* `walk1` *when visiting trapezoid $\mathcal{T}$ is*

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

where h is the height of $\mathcal{T}$ and w is its width.

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 $\alpha \leq 1$.

We now prove that

$$T_\infty(h, w) \leq 2\sigma r(2w^\alpha h - 1) \tag{8}$$

by induction on the area of the trapezoid.

Base case: If $h = 1$ and $1 \leq w < 2\sigma r$, then the procedure enters its base case with a critical path $T_\infty(h, w) = w \leq 2\sigma r \leq 2\sigma r(2w^\alpha - 1)$, and Eq. (8) holds.

Inductive step: Otherwise, the procedure recursively cuts the trapezoid into strictly smaller trapezoids for which we assume inductively that Eq. (8) 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

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

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

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

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

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

thereby proving Eq. (8) 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 r for spawning the recursive subproblems. Therefore, we have

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

The total width of the black trapezoids it at most $w - (r-1)w_g$, and therefore we have

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

Consequently, we have

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

Again by Jensen's inequality, we have

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

from which we conclude that

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

Thus, Eq. (8) holds inductively in the space cut case as well, concluding the proof of the theorem. $\square$