Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms *

Guy E. Blelloch[1] Rezaul A. Chowdhury[2] Phillip B. Gibbons[3] Vijaya Ramachandran[2]
Shimin Chen[3] Michael Kozuch[3]

[1]Carnegie Mellon University [2]University of Texas, Austin [3]Intel Research Pittsburgh

Abstract

This paper presents a *multicore-cache model* that reflects the reality that multicore processors have both per-processor private (L_1) caches and a large shared (L_2) cache on chip. We consider a broad class of parallel divide-and-conquer algorithms and present a new on-line scheduler, CONTROLLED-PDF, that is competitive with the standard sequential scheduler in the following sense. Given any dynamically unfolding computation DAG from this class of algorithms, the cache complexity on the multicore-cache model under our new scheduler is within a constant factor of the sequential cache complexity for *both* L_1 *and* L_2, while the time complexity is within a constant factor of the sequential time complexity divided by the number of processors p. These are the first such asymptotically-optimal results for any multicore model. Finally, we show that a separator-based algorithm for sparse-matrix-dense-vector-multiply achieves provably good cache performance in the multicore-cache model, as well as in the well-studied sequential cache-oblivious model.

1 Introduction

Chip multiprocessors (CMPs) [25, 24, 9], or *multicores*, are emerging as the dominant computing platform. Nearly all chip manufacturers have made the paradigm shift to focus on producing chips containing multiple processors or *cores* [1, 3, 2]. In light of this new era of parallel processing on a chip, algorithms researchers have begun exploring models, algorithms, and scheduling policies for such machines [12, 17, 19, 22, 11]. Some work has focused on the fact that CMPs have a large on-chip cache, which is shared among the processors [12, 19]. This cache provides a low latency, high bandwidth means of communicating among the processors, by writing and reading memory locations that are currently cached. Algorithms and scheduling policies are designed to make good use of the shared cache, in order to avoid the high cost of accessing the off-chip memory. This work has modeled the shared cache using a natural generalization of the *disk access machine model* [6] to multiple processors. Namely, p processors share a cache of size M and a memory of unbounded size. If a processor reads or writes a memory word w currently residing in the cache, a *shared-cache hit* occurs. If w is not in the cache, a *shared-cache miss* occurs, and the cache line of size B that contains w is placed in the cache, causing another line to be evicted if the cache is full. We will refer to this as the *(parallel) shared-cache model*.

Other work [22, 11, 19, 23] has focused on the fact that CMPs also have private caches on chip—typically one per processor. This work has modeled these caches using a variety of *distributed-cache* models [22, 11, 23]. In these models, p processors each have a cache of size C_1, and there is a shared memory of unbounded size. Hits and misses to a private cache are defined analogously to the shared-cache model. Unlike the shared-cache model, however, the same cache line can appear simultaneously in multiple private caches, as long as all processors are only reading the line [11]. Before a processor can complete a write to the line, all other copies of that line get evicted. We will refer to this as the *(parallel) private-cache model*.

In contrast to this prior work, this paper studies the combined effect of having both private and shared caches on chip. We propose a *multicore-cache model*, which reflects the reality that CMPs have both small private (first level) caches and a large shared (second level) cache on chip. There have been no formal models or analysis of such effects.

Our goal is to design an on-line scheduler for multithreaded programs that obtains provably good performance on the multicore-cache model, despite the sometimes competing effects of private and shared caches. These competing effects were observed in previous experimental studies [18], which show that (1) the state-of-the-art scheduler for the private-cache model, *work-stealing (*WS*)*, can suffer from excessive shared-cache

*Blelloch was supported in part by NSF Grant CCR-0122581 and a gift from Intel. Chowdhury and Ramachandran were supported in part by NSF Grant CCF-0514876.

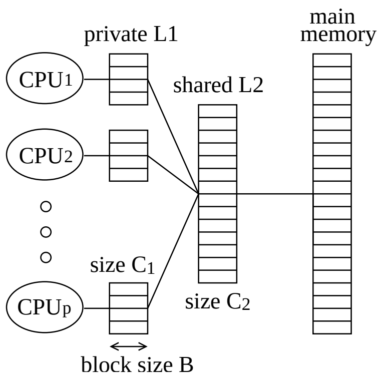


Figure 1: Multicore-cache model.

misses on CMP architectures, and conversely, (2) the state-of-the-art scheduler for the shared-cache model, *parallel depth-first (*PDF*)*, can suffer from excessive private-cache misses on CMP architectures.

In this paper, we consider a broad class of parallel divide-and-conquer algorithms, and show that a new on-line scheduler, CONTROLLED-PDF, achieves provably good cache performance on the multicore-cache model. This class includes mergesort, all-pairs shortest paths, recursive matrix addition, multiplication and inversion, as well as space-efficient parallel Strassen's matrix multiplication and matrix inversion. One of our key results is showing that a separator-based algorithm for sparse-matrix-dense-vector multiplication, an important computational kernel, also achieves provably good cache performance in the multicore-cache model (as well as in the well-studied sequential cache-oblivious model [21]).

2 A Multicore-Cache Model

Multicore-Cache Model. Our multicore-cache model is a simple combination of the private-cache and shared-cache models. There are $p > 1$ processors, where each processor has a private L_1 cache of size C_1 and all processors share an L_2 cache of size C_2, where $C_2 \geq p \cdot C_1$. There is also a shared memory of unbounded size, partitioned into data blocks of size B. Each L_1 cache has space for C_1/B data blocks; the L_2 cache has space for C_2/B data blocks. See Figure 1. Initially, all caches are empty and the input resides in the shared memory. Blocks are moved or copied between the various caches and the memory as a result of processors' read or write requests. A processor requesting to read a block must first fetch a copy of the block into its L_1 cache, if it is not already there. The same block may continue to reside in other caches and/or the memory. A processor requesting to write a block must first fetch the block into its L_1 cache, if it's not already there, and this copy must be the only copy of the block in the caches or memory—all other copies are removed (*invalidated*). (In CMPs, this rule is enforced automatically by the hardware.) A block can also be removed (*evicted*) from a cache to make room for another block. Before the sole copy of a block is evicted, it must be moved to either another cache or to the memory.

This model reflects nearly all existing CMPs, leaving unspecified several aspects where CMPs tend to differ. For example, some CMPs may enforce *inclusion*, which requires that every block in some L_1 is also in the L_2, and every block in the L_2 is also in memory (in the case of a block that is being written, the L_2 and memory blocks are marked as "stale"). In addition, the cache replacement policies, which determine what block gets evicted, may differ. For concreteness, we will assume a least-recently-used (LRU) replacement policy: For the L_1 cache, the least recently read or written block by its associated processor is evicted. For the L_2 cache, the block that has been least recently either added to the L_2 or copied to an L_1 is evicted. Note that this is one of several possible definitions of LRU for CMPs, and it idealizes real-world policies by assuming perfect LRU over the entire cache. Our asymptotic results hold (i) whether there is inclusion or not, and (ii) for a variety of LRU policies, as well as other well-studied replacement policies such as an optimal policy (as in [22]).

On-line Schedulers and PDF. As is common, we model a computation as a DAG that unfolds dynamically as the computation proceeds. Each node v in the DAG corresponds to a sequence of w_v operations (local computations, reads, or writes) to be executed by a processor; we say w_v is the *weight* of node v. Edges in the DAG correspond to serial dependences between the nodes. An *on-line scheduler* assigns nodes of the DAG to processors, as the DAG unfolds. A node is *ready* if all its ancestor nodes have executed. An assigned node may be executed on a processor if and only if it is ready.

The standard *sequential* execution of a computation DAG is according to a depth-first topological sort of the DAG (a 1DF-schedule). In a PDF-*scheduler* [13, 12], a processor completing a node is assigned the ready node that the 1DF-schedule would have executed earliest among the ready nodes. (Previous work [13, 14] has shown how the PDF-schedule can be generated on-line without having to precompute a 1DF-schedule, for multithreaded computations with nested fork-join parallelism, including divide-and-conquer algorithms, and even with synchronization variables.) In Section 3, we will define a new scheduler, called CONTROLLED-PDF, which is a hybrid of the 1DF and PDF schedulers.

Complexity Metrics. We consider three performance metrics for an algorithm (i.e., a multithreaded computation) scheduled according to a given on-line scheduler. The *time complexity* is the makespan of the schedule, measured in terms of the number of operations (i.e., assuming each operation takes unit time). It can be viewed as the number of parallel steps of the algorithm. A processor i incurs an L_1-*hit* (L_1-*miss*) if it requests to read or write a block that was (was *not*) already in i's L_1 cache. Our second metric, the L_1-*cache complexity*, is the number of L_1-misses *summed over all the* L_1 *caches*. Finally, an L_2-*hit* (L_2-*miss*) occurs if a processor requests a block not in its L_1 cache that is either (neither) in the L_2 cache or (nor) in another L_1 cache at the time of the request. (This definition reflects the fact that in real-world CMPs, a processor can grab a block residing in another L_1 cache without paying the cost of going to memory.) The L_2-*cache complexity* is the number of L_2-misses.

Although we do not explicitly compute an overall running time that incorporates all three metrics, we note that typical CMP latencies are roughly one cycle for an L_1-hit, 20 cycles for an L_1-miss that is an L_2-hit, and 300 cycles for a memory access (i.e., an L_2-miss).

In this paper, our goal is to design a parallel scheduler for the multicore-cache model that is *competitive* with the standard sequential scheduler for the standard 3-level (L_1/L_2/memory) sequential cache model, in the following sense. Given any computation DAG, the cache complexity under our parallel scheduler is within a constant factor of the sequential cache complexity for *both* L_1 *and* L_2, while the time complexity is within a constant factor of the sequential time complexity divided by p. Here, both models have the same cache sizes C_1 and C_2. This paper presents the first such results, for a broad class of DAGs defined by divide-and-conquer algorithms.

Related Work. While there have been a number of systems papers addressing the combined effects of private and shared caches on CMPs [28, 20, 30], prior theory papers have considered only private caches or shared caches in isolation. Our (3-level) multicore-cache model, in contrast, models both. Previous multi-level (i.e., $>$ 2-level) cache hierarchy models [5, 8] have studied one of two extremes: either all the levels are shared or all the levels (except for the last) are private. The former is amenable to study under the *cache-oblivious* framework [21], which shows that by optimizing for a simple 2-level sequential cache model with unknown cache size M and unknown block size B, one can optimize for an arbitrary sequential multi-level cache hierarchy. The latter reduces to independent sequential hierarchies sharing a memory, which again is amenable to cache-oblivious treatment. In contrast, it is an open question whether the combination of private and shared caches in CMPs, which we seek to model in our multicore-cache model, is amenable to a cache-oblivious framework. We observe, however, that the divide-and-conquer algorithms studied in this paper are indeed cache-oblivious—only the *scheduler* needs to know the machine's cache parameters, as detailed later.

Bender et al. [11] proposed a *concurrent cache-oblivious model*, which provides a 2-level parallel private-cache model comprised of p processors each with a private cache of unknown size M/p, and an unknown block size B. However, shared caches were not considered, and optimizing for this model has *not* been shown to optimize for a more multi-level hierarchy. Acar et al. [4] provided a strong bound on the private cache complexity of work-stealing (WS). Other 2-level private-cache models [15, 22, 23] assume a consistency model for data blocks that is significantly weaker than nearly all existing multiprocessors. As these models only reduce the number of misses compared to the model we propose, the upper bounds we show hold under such weaker consistency models.

Note that our multicore-cache model, which applies to the 3 levels of private-L_1/shared-L_2/shared-memory, also applies to the 3 levels of private-cache/shared-memory/shared-disk. Narlikar [27] presented a scheduler for this latter context, which combined PDF and WS to get the provably small memory usage of PDF together with the good cache performance of work-stealing (WS). However, the cache performance was only shown experimentally, for certain benchmarks, and it is possible to construct DAGs for which the scheduler will suffer excessive cache misses.

The challenge for scheduling in CMPs (and our multicore-cache model) is that the private and shared caches have sometimes competing demands. For good shared-cache performance, it is desirable to have the processors working on the *same* cache blocks at the same time—that way, the processors re-use blocks currently in the shared cache, instead of incurring misses caused by evicting one another's blocks. For good private-cache performance, in contrast, it is desirable to have the processors working on *disjoint* sets of cache blocks at any given time (unless all processors are reading) —this avoids misses caused by "ping-ponging" cache blocks back and forth between the private caches. None of the prior work showed how to achieve both provably-good shared-cache and private-cache performance from a single scheduler, which is the result we present in this paper for a large class of divide-and-conquer algorithms.

3 Cache-Efficient Scheduling for DC Algorithms

3.1 Hierarchical Divide-and-Conquer Algorithms. Divide-and-conquer (*DC*) algorithms often have good parallelism, and here we consider *hierarchical*

DC, where the divide and combine steps are themselves solvable using the divide-and-conquer technique. Thus, the sequential time complexity, $T_k(n)$, and the sequential cache complexity, $Q_k(C, n)$, of these algorithms can be described by the following set of *Hierarchical Recurrence Relations* (*HR*) of type k, for some $k \geq 1$.

$$T_k(n) = t_k(n) + a_k \cdot T_k(n/b_k) + \sum_{i=1}^{k-1} a_{k,i} \cdot T_i(n/b_i) \tag{3.1}$$

$$Q_k(C, n) = q_k(C, n) + a_k \cdot Q_k(C, n/b_k) + \sum_{i=1}^{k-1} a_{k,i} \cdot Q_i(C, n/b_i) \tag{3.2}$$

with base conditions $T_k(n) = \Theta(1)$ for $n \leq 1$, and $Q_k(C, n) = \Theta(S(n))$ for $S(n) \leq C$, where $S(n)$ is the total space needed to hold the input and output of a *DC* subproblem of size n, and C is the size of the cache. In our results, we assume that a constant factor change in the subproblem input size n results in a constant factor change in space $S(n)$, that is, $S(\Theta(n)) = \Theta(S(n))$.

In recurrences 3.1 and 3.2, $a_k \geq 1$, $a_{k,i} \geq 0$, $b_k > 1$ and $b_i > 1$ are integer constants, and $a_{k,k-1} \geq 1$ if $k > 1$. In the applications we present in this section 3, $t_k(n)$ and $q_k(n)$ are both $O(1)$. However, Observation 3.1 below holds for the much larger values of $t_k(n) = \mathcal{O}\left(1 + n^{\beta(k)-\epsilon} \cdot \log^{\gamma(k)-1} n\right)$ and $q_k(C, n) = \mathcal{O}\left(1 + C\left(\frac{n}{S^{-1}(C)}\right)^{\beta(k)-\epsilon} \log^{\gamma(k)-1}\left(\frac{n}{S^{-1}(C)}\right)\right)$, where $\beta(k) = \max_i \left\{\log_{b_i} a_i\right\}$, $\gamma(k) = \sum_{i=1}^{k} \left|\log_{b_i} a_i = \beta(k)\right|$, and $\epsilon > 0$ is an arbitrarily small constant. Here, $|\mathcal{E}|$ is used to denote the value of Boolean expression $\mathcal{E}$. These larger values are used in the algorithm for edge separable graphs given in section 4.

Observation 3.1. *The solutions to recurrences 3.1 and 3.2 are* $T_k(n) = \Theta\left(n^{\beta(k)} \cdot \log^{\gamma(k)-1} n\right)$, *and*
$$Q_k(C, n) = \Theta\left(C \cdot \left(\frac{n}{S^{-1}(C)}\right)^{\beta(k)} \cdot \log^{\gamma(k)-1}\left(\frac{n}{S^{-1}(C)}\right)\right).$$

Recurrences 3.1 and 3.2 describe *DC* algorithms that invoke any finite number of other *DC* algorithms as subroutines. This is more general than simple *DC* algorithms that invoke at most two types of other *DC* algorithms (for divide and combine steps). For instance, **FW-APSP** (mentioned below) is an instance of this more general class.

The *HR* class includes many important *DC* algorithms, including recursive matrix addition (**MA**) and cache-oblivious matrix multiplication (**CO-MM**) [21], both type 1; divide-and-conquer merge (**Merge**) [7] and Strassen's matrix multiplication (**St-MM**) [29], both type 2; and merge-sort (**Msort**), Strassen's matrix inversion (**St-MI**) [29] and cache-oblivious Floyd-Warshall APSP (**FW-APSP**) [19], all type 3. In all of these applications, $t_k(n)$ and $q_k(C, n)$ are $O(1)$.

Performance on Multicores. In order for an HR algorithm to perform well on multicores, three ingredients are required:

1. *Parallelism.* Some of the recursive subproblems should be computable in parallel with each other, otherwise no speed-up is achievable.

   In the multicore setting, the number of processors $p \leq \frac{C_2}{C_1} \ll$ input size, since we assume the input does not fit in L_2. Thus we are not concerned here with the polylog time for parallel algorithms considered in the **NC** setting, but rather with a more moderate level of parallelism.
2. *Space Usage.* The space needed to solve the HR algorithm with p processors should be within a constant factor of the sequential space. Otherwise not only would the space requirements for the computation increase, but the cache complexity would increase as well.

   The naive parallelization of **St-MM** is an example HR computation whose space usage increases as a function of number of processors used.
3. *Cache Efficiency.* Given an HR algorithm with good parallelism and parallel space efficiency, the cache complexity of both L_1 and L_2 caches should be comparable to the sequential cache complexity.

In the following, we will use the term *multicore HR algorithm* to denote a parallel HR algorithm whose parallel space complexity is within a constant factor of its sequential space complexity.

All of the example HR algorithms mentioned earlier have a fair amount of parallelism in them. All of them, except for the straightforward parallelization of Strassen's MM and MI, also are multicore HR algorithms, i.e., they have good parallel space complexity. Table 1 lists the recurrences for sequential running time, and the inherent parallelism (i.e., critical path length) in the algorithms we consider. The parallel running time of an algorithm with a given number of processors can be obtained by combining the solutions to these recurrences using Brent's principle [16].

Under the multicore-cache model, the number of misses for these algorithms for either the L_1 or L_2 cache could be quite large unless the parallel tasks are properly scheduled. In the next section we describe the CONTROLLED-PDF scheduler for multicore HR algorithms which schedules tasks on multicore processors so that both the L_1 and L_2 cache complexities are within a constant factor of the sequential cache complexity,

Algorithm	HR Type	Recurrences for Running Time
MA	1	$T_1^{\mathrm{MA}}(n) = 4T_1^{\mathrm{MA}}\left(\frac{n}{2}\right)$, $T_{1,\infty}^{\mathrm{MA}}(n) = T_{1,\infty}^{\mathrm{MA}}\left(\frac{n}{2}\right)$
CO-MM	1	$T_1^{\text{CO-MM}}(n) = \mathcal{O}(1) + 8T_1^{\text{CO-MM}}\left(\frac{n}{2}\right)$, $T_{1,\infty}^{\text{CO-MM}}(n) = \mathcal{O}(1) + 2T_{1,\infty}^{\text{CO-MM}}\left(\frac{n}{2}\right)$
St-MM St-MI	2 3	see Section 3.3.1
n^3-MI	2	$T_2(n) = 4T_1^{\mathrm{MA}}\left(\frac{n}{2}\right) + 6T_1^{\text{CO-MM}}\left(\frac{n}{2}\right) + 2T_2\left(\frac{n}{2}\right)$, $T_{2,\infty}(n) = 4T_{1,\infty}^{\mathrm{MA}}\left(\frac{n}{2}\right) + 5T_{1,\infty}^{\text{CO-MM}}\left(\frac{n}{2}\right) + 2T_{2,\infty}\left(\frac{n}{2}\right)$
FW-APSP (IGEP)	3	$T_1(n) = \mathcal{O}(1) + 8T_1\left(\frac{n}{2}\right)$, $T_{1,\infty}(n) = \mathcal{O}(1) + 2T_{1,\infty}\left(\frac{n}{2}\right)$ $T_2(n) = \mathcal{O}(1) + 4T_1\left(\frac{n}{2}\right) + 4T_2\left(\frac{n}{2}\right)$, $T_{2,\infty}(n) = \mathcal{O}(1) + 2T_{1,\infty}\left(\frac{n}{2}\right) + 2T_{2,\infty}\left(\frac{n}{2}\right)$ $T_3(n) = \mathcal{O}(1) + 2T_1\left(\frac{n}{2}\right) + 4T_2\left(\frac{n}{2}\right) + 2T_3\left(\frac{n}{2}\right)$, $T_{3,\infty}(n) = \mathcal{O}(1) + 2T_{1,\infty}\left(\frac{n}{2}\right) + 2T_{2,\infty}\left(\frac{n}{2}\right) + 2T_{3,\infty}\left(\frac{n}{2}\right)$
Merge	2	$T_1(n) = \mathcal{O}(1) + T_1\left(\frac{n}{2}\right)$, $T_{1,\infty}(n) = \mathcal{O}(1) + T_{1,\infty}\left(\frac{n}{2}\right)$ $T_2^{\mathrm{Merge}}(n) = T_1\left(\frac{n}{2}\right) + 2T_2^{\mathrm{Merge}}\left(\frac{n}{2}\right)$, $T_{2,\infty}^{\mathrm{Merge}}(n) = T_{1,\infty}\left(\frac{n}{2}\right) + T_{2,\infty}^{\mathrm{Merge}}\left(\frac{n}{2}\right)$
Msort	3	$T_3(n) = \mathcal{O}(1) + T_2^{\mathrm{Merge}}\left(\frac{n}{2}\right) + 2T_3\left(\frac{n}{2}\right)$, $T_{3,\infty}(n) = \mathcal{O}(1) + T_{2,\infty}^{\mathrm{Merge}}\left(\frac{n}{2}\right) + T_{3,\infty}\left(\frac{n}{2}\right)$

Table 1: Recurrences for time bounds of the *DC* algorithms considered in this section. Here, $T_{k,\infty}(n)$ is the inherent parallelism in a *DC* algorithm of type k, i.e., the number of parallel steps executed by the algorithm, when given an unlimited number of processors (ignoring caching effects). As shown in recurrences 3.1 and 3.2 (Section 3.1), recurrence for $Q_k(C, n)$ has structure similar to $T_k(n)$, and hence not included in this table.

and where optimal parallelism is achieved provided the multicore HR algorithm has sufficient parallelism. We show that this scheduler achieves optimal speed-up and cache-efficiency for several HR algorithms, including the ones mentioned earlier. For Strassen's algorithms, we achieve this by presenting an alternate parallelization to the naive method.

3.2 Controlled-PDF Scheduler for Multicore *HR* Algorithms.

Consider the multicore-cache model in which $C_2 \geq \alpha \cdot C_1$ (the model requires $\alpha \geq p$). The scheduling algorithm uses knowledge of C_1 and p, but the algorithm written by the user does not make use of these parameters, and specifies parallelism through forks and joins. The user needs to specify the space complexity function $S(n)$ mentioned earlier as well as r, the ratio between the parallel and sequential space usage of the algorithm, which is assumed to be a constant.

Let G denote the computation DAG of the given *HR* algorithm. The scheduler first transforms G as follows which can be performed on-the-fly during execution.

Let $\mu = 1/r$. The scheduler chooses $n_1 = S^{-1}(C_1)$ and $n_2 = S^{-1}(\alpha \cdot \mu \cdot S(n_1))$. It contracts each subDAG of G corresponding to a recursive function call on an input of size n_2 to an L_2-*supernode*. We denote the contracted graph by $\mathcal{C}_2(G)$. During this process the scheduler contracts all subDAGs corresponding to type-j recurrence before contracting any subDAG corresponding to type-$(j-1)$, for any j. For each L_2-supernode v the scheduler considers its corresponding subDAG in G, and contracts each subsubDAG of this subDAG corresponding to a recursive function call on an input of size n_1 to an L_1-*supernode*. The contracted subDAG is denoted by $\mathcal{C}_1(v)$. As before, it contracts all type-j subsubDAGs before contracting any type-$(j-1)$ subsubDAG.

Now the CONTROLLED-PDF scheduler is defined on G as follows. The scheduler considers the nodes (i.e., L_2-supernodes) of $\mathcal{C}_2(G)$ one at a time, in order of a 1DF-schedule. Within each L_2-supernode v, it schedules the L_1-supernodes of $\mathcal{C}_1(v)$ according to a PDF-schedule, using all p processors. Each L_1-supernode scheduled on a processor is executed entirely on that processor. After all L_1-supernodes of an L_2-supernode have been executed, the scheduler moves on to the next L_2-supernode.

We now show that under this schedule, the number of cache misses in both the L_1 and L_2 caches is within a constant factor of the sequential cache complexity.

LEMMA 3.1. *Consider the multicore-cache model in which $C_2 \geq \alpha \cdot C_1$, where $\alpha \geq p$ is a constant. If a multicore* HR *algorithm of type $k \geq 1$ incurs $\mathcal{Q}_{L_1}(n)$ L_1 cache-misses and $\mathcal{Q}_{L_2}(n)$ L_2 cache-misses under the* CONTROLLED-PDF*-scheduler, then (a) $\mathcal{Q}_{L_1}(n) = O(Q_k(C_1, n))$, and (b) $\mathcal{Q}_{L_2}(n) = O(Q_k(C_2, n))$.*

Proof. (*a*) The proof is by induction on k. For the base case $k = 1$, consider recurrence 3.2 for sequential cache misses, unraveled to a subproblem of size n_1: $Q_1(C, n) = (n/n_1)^{\log_{b_1} a_1} \cdot Q_1(C_1, n_1) + \sum_{j=0}^{\log_{b_1}(n/n_1)-1} a_1^j \cdot q_1(C_1, n/{b_1}^j)$.

The CONTROLLED-PDF scheduler schedules each subproblem of size n_1 entirely within the L_1 cache of a single processor, hence the total number of cache misses for these problems, which is equal to the cache misses incurred by the computation corresponding to first term in the recurrence above, is the same in both the sequential execution and in the CONTROLLED-PDF schedule. The second term represents computation outside the L_1 supernodes in the CONTROLLED-PDF schedule. However, by assumption, $q_1(C_1, n) = \mathcal{O}\left(1 + C_1\left(\frac{n}{S^{-1}(C_1)}\right)^{(\log_{b_1} a_1)-\epsilon}\right)$, hence the summation in the second term is $\mathcal{O}(Q_1(C_1, n))$. Hence, $\mathcal{Q}_{L_1}(n) = \mathcal{O}(Q_1(C_1, n))$

Assume inductively that any *HR* algorithm of type up to $k-1$ satisfies part (a) of the Lemma. Then, if we again unravel recurrence 3.2 for a type k *HR* algorithm down to size n_1, we can partition the terms into two parts: (i) $(n/n_1)^{\log_{b_k} a_k} \cdot Q_k(C_1, n_1) + \sum_{j=0}^{\log_{b_k}(n/n_1)-1} a_k^j \cdot q_k(C_1, n/{b_k}^j)$; plus (ii) a large number of terms that are a linear combination of terms for *HR* of type $k-1$ or less. By the induction hypothesis, computations corresponding to part (ii) have cache complexity under the CONTROLLED-PDF scheduler bounded by their sequential cache complexity. The terms in part (i) are of the same form as in the base case, and by that same argument the number of L_1 misses under the CONTROLLED-PDF schedule is the same as in the sequential case.

(b) Observe that while $S(n_2) = \alpha \cdot \mu \cdot C_1$ space in the L_2 cache suffices for the sequential execution on a subproblem of size n_2, we will need $(r-1) \cdot S(n_2) = (1-\mu) \cdot \alpha \cdot C_1$ additional L_2 cache space for its parallel execution. By our choice of n_2, we have $C_2 \geq \alpha \cdot C_1 = S(n_2) + (r-1) \cdot S(n_2)$, and hence the cache is large enough to accommodate the extra space needed by the parallel algorithm. Therefore, the parallel L_2 cache complexity of the algorithm is given by recurrence 3.2 for Q_k on a cache of size $C_2 - (r-1) \cdot S(n_2) \geq \mu \cdot C_2$. Hence, using Observation 3.1, and letting $S^* = S^{-1}(\mu \cdot C_2)$, we obtain, $\mathcal{Q}_{L_2}(n) = \mathcal{O}(Q_k(\mu \cdot C_2, n)) = \mathcal{O}\left(\mu \cdot C_2 \cdot (n/S^*)^{\beta(k)} \cdot \log^{\gamma(k)-1}(n/S^*)\right) = \mathcal{O}\left(C_2 \cdot \left(n/S^{-1}(C_2)\right)^{\beta(k)} \cdot \log^{\gamma(k)-1}\left(n/S^{-1}(C_2)\right)\right) = \mathcal{O}(Q_k(C_2, n))$ (since $S(\Theta(n)) = \Theta(S(n))$). □

Note that unlike PDF and WS, CONTROLLED-PDF is not a greedy schedule—processors may idle (waiting for the completion of an L_2-supernode) in order to guarantee the bounds in Lemma 3.1. We show next that under certain general conditions, this idling does not effect the parallel time complexity by more than a constant factor. Given the high cost of cache misses (e.g., 300 cycles for fetching from memory), reducing misses can often lead to faster overall execution times in practice, even with a modest amount of imposed idling.

With CONTROLLED-PDF a p-fold speed-up is achievable *provided* the HR algorithm has sufficient parallelism within it to counteract the slowdown caused by the serialization within the L_1 supernodes. To achieve this, it is necessary to have a sufficiently large n_2, i.e., a sufficiently large α, as described below.

In recurrence 3.1, let $f_k(n) = t_k(n) + \sum_{i=1}^{k-1} a_{k,i} \cdot T_i\left(\frac{n}{b_i}\right)$ (hence $T_k(n) = f_k(n) + a_k \cdot T_i\left(\frac{n}{b_k}\right)$).

Let $T_{i,\infty}$ denote the inherent parallelism in the type-i HR and let $f_{i,\infty}$ represent the inherent parallelism in its divide and combine steps. Then, $T_{i,\infty}$ will satisfy the following recurrence for some $a_{i,\infty} \leq a_i$:

$$T_{i,\infty}(n) = a_{i,\infty} T_{i,\infty}(n/b_i) + f_{i,\infty}(n)$$

On p processors this type-i HR will run in time $\mathcal{O}(T_i(n)/p + T_{i,\infty}(n))$ using a standard Brent-type schedule. We now analyze the number of parallel steps under the CONTROLLED-PDF-schedule.

Let $T_{i,\infty}(n_2,\ n_1)$ denote the inherent parallelism of a type-i L_2-supernode, $1 \leq i \leq k$, under the CONTROLLED-PDF scheme. Then we have

$$T_{i,\infty}(n,\ n_1) \leq \begin{cases} T_i(n) & \text{if } n \leq n_1, \\ a_{i,\infty} T_{i,\infty}(n/b_i) + f_{i,\infty}(n) & \text{otherwise} \end{cases}$$

On p processors the CONTROLLED-PDF-scheduler will execute this type-i L_2-supernode in $T_{i,p}(n_2) = T_i(n_2)/p + T_{i,\infty}(n_2,\ n_1)$ parallel steps. Thus the CONTROLLED-PDF scheduler achieves optimal parallel speed-up if

$$\frac{T_i(n_2)}{p} = \Omega\left(T_{i,\infty}(n_2,\ n_1)\right). \tag{3.3}$$

All of the parallelism is realized during the process of reducing the input size from n_2 to n_1, and subproblems of size n_1 are executed sequentially under the CONTROLLED-PDF schedule. Since n_2 depends on $\alpha \cdot C_1$, the ability to achieve optimal speed-up depends on whether α is sufficiently large to satisfy equation 3.3, and is dependent on the inherent parallelism in the *HR* algorithm. The multi-core cache model requires $C_2 \geq \alpha \cdot C_1$ for $\alpha = p$, and we may sometimes need to use a value of $\alpha > p$ in order to achieve full parallelism, as we shall see in some of the applications considered below. It should be noted that in practice α is much larger than p.

3.3 Applications. Applications of our scheme for *HR* algorithms include the following.

Type 1:

- **MA**: Recursive matrix addition
- **CO-MM**: Cache-oblivious matrix multiplication (does not assume associativity of additions)

Type 2:

- **St-MM**: Strassen's matrix multiplication

Algorithm	HR Type	$T(n)$	$Q(C,n)$	$T_\infty(n)$	$S(n)$	n_1	$T_\infty(n_2\ ,\ n_1)$	$\alpha \geq$
MA	1	n^2	$\frac{n^2}{B}$	1	n^2	$\sqrt{C_1}$	$C_1 + \log \alpha$	p
CO-MM	1	n^3	$\frac{n^3}{B\sqrt{C}}$	n	n^2	$\sqrt{C_1}$	$C_1^{\frac{3}{2}} \cdot \sqrt{\alpha}$	p
St-MM	2	n^ζ	$\frac{n^\zeta}{B\sqrt{C}}$	$n^{\zeta-2+\epsilon} = n^{.81}$ arb. $\epsilon > 0$	n^2	$\sqrt{C_1}$	${C_1}^{\frac{\zeta}{2}} \cdot \alpha^{\frac{\zeta-2+\epsilon}{2}}$	$p^{1+\epsilon}$, arb. $\epsilon > 0$
St-MI	3	n^ζ	$\frac{n^\zeta}{B\sqrt{C}}$	n	n^2	$\sqrt{C_1}$	${C_1}^{\frac{\zeta}{2}} \cdot \sqrt{\alpha}$	$p^{\frac{2}{\zeta-1}}$
n^3-MI	2	n^3	$\frac{n^3}{B\sqrt{C}}$	$n \log n$	n^2	$\sqrt{C_1}$	${C_1}^{\frac{3}{2}} \cdot \sqrt{\alpha} \cdot \log \alpha$	$p \log p$
FW-APSP (IGEP)	3	n^3	$\frac{n^3}{B\sqrt{C}}$	$n \log^2 n$	n^2	$\sqrt{C_1}$	$C_1^{\frac{3}{2}} \cdot \sqrt{\alpha} \cdot \log^2 \alpha$	$p \log^2 p$
Merge	2	n	$\frac{n}{B}$	$\log^2 n$	n	C_1	$C_1 + \log \alpha \cdot \log(\alpha C_1)$	$p \cdot \left(1 + \frac{\log \alpha \cdot \log(\alpha C_1)}{C_1}\right)$
Msort	3	$n \log n$	$\frac{n}{B} \log \frac{n}{B}$	$\log^3 n$	n	C_1	$C_1 \log C_1$ $+ \log^2 \alpha \cdot \log(\alpha C_1)$	$p \cdot \left(1 + \frac{\log^2 \alpha}{C_1}\right)$

Table 2: Parameter values for the applications mentioned Section 3.3. The expression for n_2 is the same as that for n_1 with C_1 replaced by $\alpha \cdot C_1$. The dependence of the cache complexity on block size B is given for clarity. All values are to within a constant factor, and $\zeta = \log_2 7$.

- n^3**-MI**: Strassen's associative matrix inversion method using CO-MM and MA for the matrix multiplications and additions.
- **Merge**: Divide & conquer merge of two sorted lists

Type 3:

- **St-MI**: Strassen's associative matrix inversion
- **Msort**: Merge-sort using the type-2 Merge.
- **FW-APSP**: Cache-oblivious Floyd Warshall APSP, Gaussian elimination without pivoting, and certain other applications of IGEP.

The key parameters for these applications are summarized in Tables 1 and 2. All of these are multicore *HR* algorithms that achieve optimal L_1 and L_2 cache efficiency as well as full p-fold speed-up using linear space. There is some variation in the value of α needed to achieve full parallelism: MA and CO-MM achieve optimal performance for any $\alpha \geq p$. Merge and Msort have additional requirements on α to achieve full parallelism that depend inversely on C_1, but these are very likely to be much smaller than p in practice; all other applications need α to be at most an $O(p^{1+\epsilon})$, where $\epsilon < 0.11$ for St-MI, and ϵ is an arbitrarily small constant > 0 for the other algorithms.

The results given in Table 2 for CO-MM and FW-APSP follow directly from their *HR* algorithms [21, 19], while MA is a trivial recursive algorithm for matrix addition. The results for n^3-MI follow by considering Strassen's matrix inversion algorithm [29], and using CO-MM for the matrix multiplications. The divide-and-conquer Merge recursively divides a merge problem of total size n into two merge problems of sizes $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ [7], which is then used in mergesort (Msort).

3.3.1 St-MM and St-MI.

Strassen's matrix multiplication algorithm [29], which recursively multiplies seven matrices with half the number of rows is a type-2 *HR* algorithm that uses the trivial type-1 *HR* for matrix addition in its divide and combine steps. Since all of the seven recursive calls can be performed in parallel, the algorithm achieves $O(\log n)$ parallel time given sufficient number of processors. The time bounds $T_1(n)$ for MA and $T_2(n)$ for St-MM and their parallel time bounds satisfy the following recurrences: (i) $T_1(n) = 4 \cdot T_1(n/2)$; (ii) $T_2(n) = 7 \cdot T_2(n/2) + c \cdot T_1(n/2)$; (iii) $T_{1,\infty}(n) = T_{1,\infty}(n/2)$; and (iv) $T_{2,\infty}(n) = T_{2,\infty}(n/2) + T_{1,\infty}(n/2)$,

Hence the parameters of this computation are:

- $T_2(n) = O(n^{\log_2 7})$ and $T_{2,\infty}(n) = O(\log n)$; and sequential space requirement, $S(n) = \Theta(n^2)$.
- On p processors $T_{1,p}(n) = n^2/p$ and $T_{2,p}(n) = O(\frac{n^\zeta}{p} + \log n)$, where $\zeta = \log_2 7$.

This parallelization does not lead to a multicore *HR* since the space requirement with p processors is $p^{1-\frac{2}{\zeta}} n^2$. We consider instead the following alternate parallelization of Strassen's algorithm. We unravel the recursion a constant number of times k so that we recursively multiply $\frac{n}{2^k} \times \frac{n}{2^k}$ matrices. There are 7^k recursive matrix multiplications, which we perform in $\tau = \lceil \frac{7^k}{4^k - 1} \rceil$ steps, each step performing at most $4^k - 1$ matrix multiplications in parallel, with the matrices to be multiplied chosen to maximize the number of matrix additions that can be performed in that step. This leads to the following recurrences for the parallel time T_∞ and parallel space complexity S_∞:

$$T_\infty(n) = \tau \cdot T_\infty(n/2^k) + c' \cdot T_1(n/2^k)$$

$$S_\infty(n) \leq (4^k - 1) \cdot S_\infty(n/4^k) + \Theta(n^2)$$

This leads to the solution $S_\infty(n) = O(n^2)$ for the space. For $T_\infty(n)$ we have

$$T_\infty(n) = O(n^{\frac{1}{k} \log \tau}) = O(n^{\log_2 7 - \log_2 4 + \epsilon})$$

for an arbitrarily small constant $\epsilon > 0$.

Hence this gives a multicore *HR* algorithm with $T_\infty(n) = O(n^{0.81})$.

We also need to investigate the constraint on α implied by equation 3.3. Since $S(n) = \Theta(n^2)$ we have $n_1 = \sqrt{C_1}$ and $n_2 = \sqrt{\alpha \cdot C_1}$. Recall $\zeta = \log_2 7$. We have $T_2(n_2) = (\sqrt{\alpha \cdot C_1})^\zeta$ and $T_\infty(n_2\ ;\ n_1) = (\sqrt{\alpha})^{\frac{1}{k}\log\tau}(\sqrt{C_1})^\zeta$, hence since $\frac{1}{k}\log\tau = \zeta - 2 + \epsilon$, by equation 3.3 we need

$$(\sqrt{\alpha \cdot C_1})^\zeta / p \geq (\sqrt{\alpha})^{\zeta - 2 + \epsilon}(\sqrt{C_1})^\zeta \quad \text{or} \quad \alpha \geq p^{1+\epsilon}$$

for any arbitrarily small $\epsilon > 0$ (using a suitable choice of k).

In St-MI, Strassen's divide-and-conquer algorithm for matrix inversion, both divide and combine steps use MA and St-MM and the matrix inversion complexity T_3 satisfies the following recurrences (where T_2 and T_1 are the sequential complexities of St-MM and MA given above, and the c_i and d_i are suitable constants).

$$T_3(n) = 2 \cdot T_3(n/2) + c_1 \cdot T_2(n/2) + c_2 \cdot T_1(n/2)$$

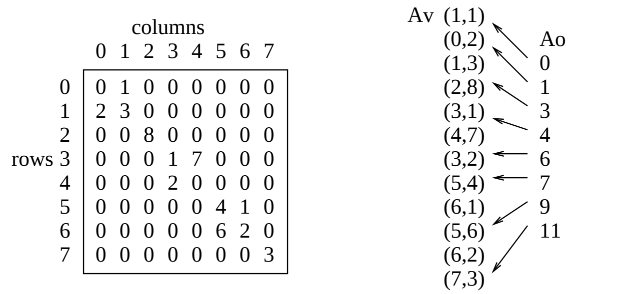
$$T_{3,\infty}(n) = 2 \cdot T_{3,\infty}(n/2) + d_1 \cdot T_\infty(n/2) + d_2 \cdot T_{1,\infty}(n/2)$$

Since the two recursive computations of inverses are performed sequentially the space bound is given by $S(n) = S(n/2) + O(n^2)$, and hence $S(n) = O(n^2)$. Hence St-MI has the parameters listed in Table 2.

4 Cache Efficiency of Edge Separable Graphs

We consider the problem of multiplying a sparse matrix by a dense vector. We describe a simple cache-oblivious algorithm which has good cache performance when the matrix (viewed as a graph) has small separators. We then show that a parallel variant has good cache performance under the multicore-cache model when using a CONTROLLED-PDF scheduler. Real-world matrices (graphs) often have small separators because either they are embedded in some low-dimensional space (*e.g.*, planar graphs or 3d meshes), or they represent some form of community or locality (*e.g.*, a web graph).

Let S be a class of graphs closed under the subgraph relation. We say that S satisfies a $f(n)$-*edge separator theorem* if there are constants a between $\frac{1}{2}$ and 1 and $b > 0$ such that every graph $G = (V, E)$ in S with n vertices can be partitioned into two sets of vertices V_1 and V_2 each of size at most an such that $|\{(u, v) \in E | (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)\}| \leq bf(n)$ [26]. Let the support graph $G = (V, E)$ of an $n \times n$ matrix be the graph with $V = \{1, \ldots, n\}$ and $E = \{(i, j) | A[i, j] \neq 0\}$. We say that a matrix satisfies an $f(n)$-edge separator theorem if its support graph satisfies such a theorem. We say a π-reordering of a matrix is a permutation of



(a) a sparse matrix (b) row−major representation

Figure 2: Row-major representation of a sparse matrix.

both the rows and columns by π. In the support graph this corresponds to a permutation π of the vertex labels. We note that graphs that belong to a class that satisfies an n^ϵ-edge separator for $\epsilon < 1$ have bounded degree.

The *row-major representation* of a sparse matrix A is the pair (A_v, A_o) where A_v is a vector of all the non-zero elements stored adjacently sorted first by row and then by column, and each element A_{ij} is stored as a pair consisting of the value (A_{ij}) and the column number (j), as shown in Figure 2. A_o is a vector containing the start location (index) of each row in A_v. The *row-major matrix algorithm* for multiplying a row-major representation of a sparse matrix A by a dense vector x is defined as follows:

for $i = 1$ to $|A_o|$
  $y[i] = 0$
  for $k = A_o[i]$ to $A_o[i+1] - 1$
    $(j, a) = A_v[k]$;
    $y[i] = y[i] + a * x[j]$;

We first consider the cache performance of this sequential algorithm.

LEMMA 4.1. *Any $n \times n$ matrix A satisfying an n^ϵ-edge separator theorem for $\epsilon < 1$ can be reordered so that the row-major algorithm generates at most $O(n/B + n/M^{1-\epsilon})$ cache misses in the cache-oblivious model with block size B and cache size M.*

Proof. The row-major algorithms scans the rows in order. The number of non-zeros is $O(n)$ because of the separator theorem. Since A_v and A_o are laid out in order, the number of cache misses caused by reading these is $O(n/B)$. Furthermore, y is scanned in order creating another $O(n/B)$ cache misses.

We now consider the accesses to $x[j]$. Let G be the support graph for A. Consider a separator tree T for G constructed by applying the separator theorem to the whole graph to get two components, and then recursively applying the separator theorem to each component until only a single node remains at each leaf of the tree. We refer to the left to right ordering of the leaves as a *separator ordering* of the vertices of G. We show that the row-major algorithm π-reordered to the separator ordering satisfies the desired bounds.

As we scan through the rows of the matrix in the row-major algorithm consider keeping a window of width M of the vector x loaded in the cache (technically we need three additional blocks for A_v, A_o and y). This window is centered as the current row number. The total number of misses to load the window as we scan along the rows is n/B. Consider any subtree T' of T that contains i and is of size (number of leaves) at most $M/2$. When the algorithm is at a row i consider a non-zero element A_{ij} that is processed causing a read of $x[j]$. This corresponds to an edge (i, j) in the graph. If j is within the subtree then $x[j]$ is a cache hit (*i.e.*, j is within the window). We therefore only need to consider edges that separate a tree that is larger than $M/2$, and we assume all of these will cause a cache miss. The number of such edges is bounded by the following recurrence:

$$R(n) = \begin{cases} \max_{1/2 \le a' \le a} \{R(a'n) + R((1-a')n) \\ \quad +bn^\epsilon\}, \quad \text{if } n > M/2 \\ 0, \quad \text{otherwise,} \end{cases}$$

where a and b are the parameters of the separator theorem. By induction we can verify that $R(n) \le k(n/(M/2)^{1-\epsilon} - n^\epsilon)$, where $k = b/(a^\epsilon + (1-a)^\epsilon - 1)$. The total number of misses is therefore bounded by $O(n/B)$ for scanning A, y, and the window of x, and $O(n/M^{1-\epsilon})$ for any accesses to x outside the window. Since we assume an LRU cache we note that we can simply charge each miss outside the window twice—once for accessing the $x[i]$ and once for refilling the window slot. $\Box$

This bound can be compared to the bounds presented by Bender *et al.* [10]. For their version that can choose the memory layout and for $m = O(n)$ the bounds are $\Theta\left(\min\left\{\frac{n}{B}\left(1 + \log_{M/B}\frac{n}{M}\right), n\right\}\right)$. They show this is optimal. Because of the separator properties, however, our bounds are significantly better with only an additive term with respect to the scanning time n/B instead of the multiplicative term.

We now consider the following simple divide-and-conquer parallel version of the row major algorithm. The *parallel row-major algorithm* divides the rows in half (first $n/2$ and second $n/2$), and in parallel calculates the results for each half. Recurse until there is a single row. As in Section 3, we use a CONTROLLED-PDF to schedule the algorithm. Here we use n_1 and n_2 to be the number of rows that are grouped in an L_1 and L_2 supernode, respectively. Specifically, we use a 1DF-schedule on the L_2 supernodes until we reach a problem of size n_2 or less, then we use a PDF-schedule on the L_1 supernodes, and within the L_1 supernodes (n_1 or fewer rows) we use a sequential 1DF-schedule. Lemma 3.1 can easily be extended to consider this algorithm, but here we show the cache complexity directly using the separator properties.

LEMMA 4.2. *Any $n \times n$ matrix A satisfying an n^ϵ-edge separator theorem with $\epsilon < 1$ can be reordered so that parallel row-major algorithm using the* CONTROLLED-PDF*-scheduler for some suitable choice of supernode sizes, incurs $O(n/B + n/C_1^{1-\epsilon})$ L_1 cache misses and $O(n/B + n/C_2^{1-\epsilon})$ L_2 cache misses.*

Proof. We use the same reordering as in Lemma 4.1. The bounds on the L_1 cache follow from the Lemma because we can apply the window argument as we sweep the rows in individual processors in an L_1 supernode, but we need to account of the fact that each group might start with an empty L_1 cache and have to fill the L_1 cache. Since there are n/n_1 L_1 supernodes each needing to load $O(C_1/B)$ blocks, the filling will create $O(nC_1/(n_1B))$ L_1 misses. If we pick $n_1 \ge C_1$ this $O(n/B)$ term can be counted against scanning the rows of the matrix. For the L_2 cache we select $n_2 \le C_2/2$ and can therefore fit the L_2 supernode in half the cache. We can make a similar argument as in Lemma 4.1 but with the window centered at the middle of the L_2 supernode (in row order). If we consider the leftmost and rightmost L_1 supernode within the current L_2 supernode they are both at least $C_2/4$ away from the end of the window. Therefore their cache characteristics are no worse than in the sequential case with a cache half as big ($C_2/2$). Furthermore as we move from one L_2 supernode to the next L_2 supernode, the windows overlap so we do not create additional misses during the transitions. $\Box$

Comparing Lemmas 4.1 and 4.2, we see that the L_1 and the L_2 cache complexities on the multicore-cache model are within constant factors of the sequential L_1 and L_2 cache complexities. The parallel time for the parallel row-major algorithm is efficient if we pick $n_1 = C_1$ and $n_2 = C_2/2$ so $n_2/n_1 = C_2/(2C_1)$ and as long as $\alpha \ge p(1 + \frac{\log \alpha}{C_1})$.

In practice, the same matrix is typically used repeatedly, e.g., as part of an iterative solver or with different vectors. Thus, as in [10], we do not account for the cost of preprocessing the matrix into a desired layout, as this cost is amortized over many repeated uses.

5 Conclusion

In this paper we have proposed the multicore-cache model, which captures the cache configuration of current and proposed chip multiprocessors by including a shared (L_2) cache and associating a private (L_1) cache with each processor. Previous schedulers that tried to optimize cache complexity in the parallel setting have worked very differently depending on whether the caches are private or shared. We have presented the CONTROLLED-PDF scheduler, which schedules a large class of divide-and-conquer algorithms such that both the private L_1 and shared L_2 cache misses match the sequential cache complexity while maintaining full parallel speed-up. An important topic for further research is to extend these results to other types of algorithms.

We note that current chip multiprocessors have several levels of (off-chip) memory larger than the L_1 and L_2 caches (e.g., RAM and disk). However, these larger levels are all shared, and thus the hierarchy starting at L_2 can be modeled by the cache-oblivious model [21]. All of the algorithms we have analyzed under the CONTROLLED-PDF scheduler have good sequential cache-oblivious bounds, and they maintain this property for these larger levels of the memory hierarchy when scheduled by the CONTROLLED-PDF scheduler. One can also consider hierarchies of private and shared caches. Modeling such caches and developing algorithms and schedulers for them is left as a topic for further research.

References

[1] www.sun.com/processors/UltraSPARC-T1/, 2007.

[2] www.tilera.com, 2007.

[3] Intel shows off 80-core processor. www.news.com/2100-1006_3-6158181.html, 2007.

[4] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3), 2002. Springer.

[5] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *ACM STOC*, 1987.

[6] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988.

[7] S. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, 36(11), 1987.

[8] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierachy model of computation. *Algorthmica*, 12(2/3), 1994. Springer.

[9] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *ACM ISCA*, 2000.

[10] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *ACM SPAA*, 2007.

[11] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *ACM SPAA*, 2005.

[12] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM SPAA*, 2004.

[13] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2), 1999.

[14] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *ACM SPAA*, 1997.

[15] 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 *ACM SPAA*, 1996.

[16] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974.

[17] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *ACM SPAA*, 2007.

[18] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *ACM SPAA*, 2007.

[19] R. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *ACM SPAA*, 2007.

[20] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Ann. Tech. Conf.*, 2005.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE FOCS*, 1999.

[22] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *ACM SPAA*, 2006.

[23] M. T. Goodrich, M. Nelson, and N. Sitchinava. Sorting in parallel external-memory multicores. Technical report, U.C. Irvine, 2007.

[24] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2), 2000.

[25] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), 1997.

[26] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2), 1979.

[27] G. J. Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2), 2002. Springer.

[28] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *ACM ISCA*, 1996.

[29] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), 1969. Springer.

[30] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *ACM EuroSys*, 2007.