Program-Centric Cost Models for Locality

Guy E. Blelloch*, Jeremy Fineman†, Phillip B. Gibbons‡, Harsha Vardhan Simhadri*

*Carnegie Mellon University †Georgetown University ‡Intel Labs Pittsburgh

{guyb,harshas}@cs.cmu.edu, jfineman@cs.georgetown.edu, phillip.b.gibbons@intel.com

Abstract

In this position paper, we argue that cost models for locality in parallel machines should be *program-centric*, not machine-centric.

Categories and Subject Descriptors F.2 [*Analysis of Algorithms and Problem Complexity*]; D.2.8 [*Metrics*]: Complexity Measures and Performance Measures

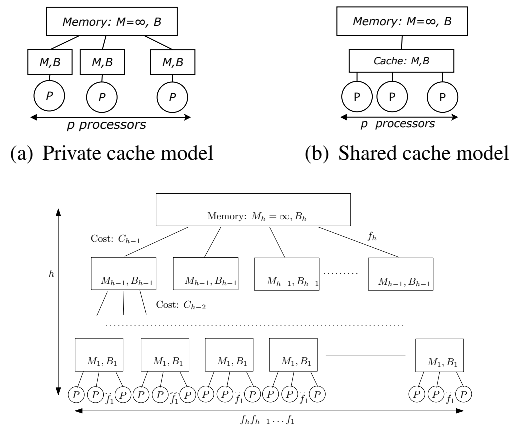
Keywords Parallelism, Locality, Program-Centric Models

1. Introduction

For good scaling performance, parallel programs should be designed to make efficient use of communication and caches. However, requiring the programmer to hand-tune her program for locality on individual machines is counter-productive. It takes considerable effort to understand a new machine and optimizing for a specific machine draws attention away from portability and correctness. One approach is to model locality for a class of parallel machines using a machine-centric model such as the Parallel External Memory model (PEM) [1], Bulk-Synchronous Parallel (BSP) or the Multi-BSP model [13]. Such models explicitly view the machine as a specific organization of a collection of processors, caches and memory. Program costs are then analyzed, roughly, in terms of data transfers between the components (e.g., between processors, or between caches and memory). This approach to capturing locality is specific to a particular machine organization and requires that the user carefully schedules tasks onto processors.

To avoid this problem, we propose that locality in programs should be quantified in an abstract programming model rather than with regards to a machine-centric model. In this methodology, expressions for communication costs are derived without thinking about how a program is scheduled on particular machines or caches. We suggest two models for this purpose: an adaptation of the sequential Cache-Oblivious model, and a more general Parallel Cache-Oblivious (PCO) model [5]. Both are *program-centric* (a.k.a. *multicore-oblivious* [9]) in that they are defined in terms of the DAG representing a program execution and have no notions of processors or cache hierarchy specifics.

Such models have little utility if they do not represent costs on real machines. Therefore, in addition to the model itself, it is critical to show general bounds on performance when mapping the program onto particular machine organizations such as the tree of caches (fig. 1(c)). Ideally, one high-level program-centric model for locality can replace many machine-centric models, allowing portability of programs and the analysis of their locality across machines via machine-specific mappings. Such mappings make use of a scheduler designed for a particular cache organization. Schedulers should come with program performance guarantees that are based solely on program-centric metrics (e.g., locality, work, depth) for the program and parameters of the machine.



(a) Private cache model (b) Shared cache model

(c) A Parallel Memory Hierarchy, modeled by an h-level tree of caches, with $\Pi_{i=1}^{h} f_i$ processors.

Figure 1. Machine-centric models

In this paper, we highlight two program-centric models for locality and corresponding schedulers that provably preserve the locality on single- and multi-level cache models with shared global memory (fig. 1). More details can be found elsewhere [4, 5].

2. Analyzing Locality

Locality in sequential programs. The Cache-Oblivious (CO) model [11] is a popular model for measuring locality of sequential programs. In this model, programs are measured against an abstract machine with two levels of memory: a slow unlimited RAM and a fast "ideal" cache with limited size. Locality for a sequential program is expressed in terms of the cache complexity function $Q(M, B)$, defined to be the number of cache misses on an ideal cache of size M and cache line size B. As long as a program does not use the parameters M and B, the bounds for the single level are valid simultaneously across all levels of a multi-level cache hierarchy. This model allows easy comparison of locality in programs and has encouraged careful data layout and space management in algorithms to maximize locality [10].

Analyzing locality in parallel programs using sequential cache complexity and depth. Parallel programs can be represented by directed acyclic graphs (DAGs) that impose a partial order on the instructions. The DAG may unfold dynamically as the program runs. One approach to analyzing the locality/cache complexity of DAGs is to impose a sequential (total) order on the instructions consistent with the DAG, say according to a depth-first order (fig. 2(l)). The cache complexity can then be analyzed in the CO model with respect to this order. Here, we focus only on nested parallel DAGs—programs with dynamic nesting of fork-join constructs but no other synchronizations—as they are more tractable and sufficiently expressive for many algorithmic problems [6]. It turns out that the depth-first order based cache complexity Q_1, the depth D (the critical path, or span), and the work W of the DAG—all program-centric metrics—can be used to bound the performance

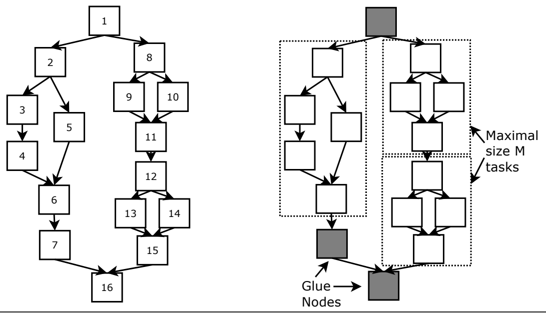


Figure 2. (l) Depth-first schedule, (r) PCO analysis

of nested-parallel programs on machine models with one level of caches, as follows.

Mapping Q_1, D, and W for one level hierarchies. Two natural parallel machine models with a single cache level are the private cache (as in PEM model [1]) and the shared cache machine models [2]. The private model (fig. 1(a)) is a collection of p processors each with its own cache of dimensions M and B, all sharing a RAM. In the shared cache model (fig. 1(b)), p processors share one cache of dimensions M and B. In both models, processors can access their own caches in unit time and cache misses go to RAM and take C units of time. In both cases, the total number of cache misses Q_P and running time T can be bounded in terms of Q_1, D and W by appropriate choice of scheduler: Work-Stealing (WS) [8] for the private cache model and Parallel Depth-First (PDF) [3] for the shared cache model:

WS: $Q_P \le Q_1(M,B) + O(pD_lM/B)$; $T \le (W + C \cdot Q_P)/p + D_l$,
PDF: $Q_P(M,B) \le Q_1(M - pBD_l, B)$; $T \le (W + C \cdot Q_P)/p + D_l$,

where $D_l = DC$. For low depth programs (polylog in input size) with good locality according to Q_1, these schedulers guarantee good performance—Q_P is close to Q_1, T is close to optimal [4].

Analyzing locality in parallel programs using the PCO model. Although Q_1, D, and W are good program-centric metrics for machines where one level of cache (either private or shared) dominates the program's running time, it has proven to be hard to generalize these results to multiple levels (Sec. 3, [5]). The problem arises because the depth-first order based Q_1 accounts for cache line reuse between instructions unordered in the DAG. The Parallel Cache-Oblivous (PCO) model [5] overcomes this problem and seems to be more suitable for analyzing performance on cache hierarchies. In the PCO model, locality, denoted by $Q^*(M,B)$, is defined based solely on the composition rules used to construct nested parallel programs. The model and analysis have no notions of processors, schedulers, or cache hierarchy specifics.

We refer to any segment of a DAG between a fork and the corresponding join as a task. We refer to the memory footprint of a task (total number of distinct cache lines it touches) as its size. We say a task is *size M maximal* if its size is less than M but the size of its parent task is more than M. Roughly speaking, the PCO analysis decomposes the program into a collection of maximal subtasks that fit in M space, and "glue nodes"—instructions outside these subtasks (fig. 2(r)). For a maximal size M task $\mathtt{t}$, the PCO cache complexity $Q^*(\mathtt{t}; M, B)$ is defined to be the number of distinct cache lines it accesses, counting accesses to a cache line from unordered instructions multiple times. The model then pessimistically counts all memory instructions that fall outside of a maximal subtask (i.e., glue nodes) as cache misses. The total cache complexity of a program is the sum of the complexities of the maximal tasks, and the memory accesses outside of maximal tasks. Although it may seem that this is overly pessimistic, it turns out that for many well-designed parallel algorithms, $Q^* = O(Q_1)$ [5].

Mapping Q^* to multi-level hierarchies. To understand the effectiveness of the PCO model when mapped to a multi-level cache hierarchy we consider the Tree of Caches model (fig. 1(c)). The model is parameterized by the number of cache levels h, the dimensions of the cache M_i, B_i and fan-out f_i at level-i, and transfer cost C_i from level i to $i+1$. The total number of processors is $p = \Pi_{i=1}^{h} f_i$. To obtain a good schedule for this model, a careful mix of ideas from WS, PDF and space bounded [9] schedulers can be applied at different scales in the program. However, it may not be possible to effectively schedule "irregular" programs on the model, i.e., programs with vastly different work-size balances across different segments within the DAG. The greater the irregularity, the lesser the exploitable parallelism. In [5], we quantified the notion of exploitable parallelism by extending the PCO model to a new metric $Q^*_\alpha(M,B)$ to account for the cost of irregularity. Further, for most "reasonable" algorithms, especially polylog-depth algorithms, we showed that $Q^*_\alpha(M,B) = O(Q^*(M,B))$. Finally, we constructed an optimal scheduler based on the Q^*_α metric that can execute any "reasonable" program within time $O((\Sigma_{i=1}^{h} Q^*(M_i, B_i) \cdot C_i)/p)$.

3. Conclusion

This paper highlighted our efforts to design program-centric locality measures that can be provably mapped to various machine models. We have also designed algorithms for various problems that are optimal based on these program-centric measures [4, 7]. Implementations of these algorithms have shown great scalability on shared memory machines with deep hierarchies, validating our approach [6, 7, 12]. Extrapolating from these successes, we advocate using this approach for principled design and analysis of programs for new machines with increasingly complex memory hierarchies.

Acknowledgments. This work is partially supported by the National Science Foundation under grants CCF-1018188 and CCF-1218188, by Intel Labs Academic Research Office, and by the Intel Science and Technology Center for Cloud Computing.

References

[1] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache multiprocessors. In *SPAA*, 2008.

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

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

[4] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

[5] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, 2011.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, 2012.

[7] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and i/o efficient set covering algorithms. In *SPAA*, 2012.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), 1999.

[9] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, 2010.

[10] E. D. Demaine. Cache-oblivious algorithms and data structures. In *Summer School on Massive Data Sets*, LNCS. Springer-Verlag, 2002.

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

[12] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *SPAA*, 2012.

[13] L. G. Valiant. A bridging model for multi-core computing. In *ESA*, 2008.