Coupling Memory and Computation for Locality Management

Umut A. Acar[1,2], Guy Blelloch[1], Matthew Fluet[3], Stefan K. Muller[1], and Ram Raghunathan[1]

1 **Carnegie Mellon University, Pittsburgh, PA, USA,**
{umut,blelloch,smuller,ram.r}@cs.cmu.edu
2 **Inria, Paris, France**
3 **Rochester Institute of Technology, Rochester, NY, USA,** mtf@cs.rit.edu

Abstract

We articulate the need for managing (data) locality automatically rather than leaving it to the programmer, especially in parallel programming systems. To this end, we propose techniques for coupling tightly the computation (including the thread scheduler) and the memory manager so that data and computation can be positioned closely in hardware. Such tight coupling of computation and memory management is in sharp contrast with the prevailing practice of considering each in isolation. For example, memory-management techniques usually abstract the computation as an unknown "mutator", which is treated as a "black box".

As an example of the approach, in this paper we consider a specific class of parallel computations, nested-parallel computations. Such computations dynamically create a nesting of parallel tasks. We propose a method for organizing memory as a tree of heaps reflecting the structure of the nesting. More specifically, our approach creates a heap for a task if it is separately scheduled on a processor. This allows us to couple garbage collection with the structure of the computation and the way in which it is dynamically scheduled on the processors. This coupling enables taking advantage of locality in the program by mapping it to the locality of the hardware. For example for improved locality a heap can be garbage collected immediately after its task finishes when the heap contents is likely in cache.

1998 ACM Subject Classification D.3.3. Dynamic Storage Management

Keywords and phrases Parallel computing, locality, memory management, parallel garbage collection, functional programming, nested parallelism, thread scheduling

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.1

1 Introduction

A confluence of hardware and software factors have made the cost of memory accesses and thus management of data locality an important theoretical and practical challenge.

On the hardware side, we have witnessed, over the past two decades, an increasing performance gap between the speed of CPU's and off-chip memory, a.k.a., the memory wall [59]. Starting with sequential architectures, the growing CPU-memory gap led to the development of deep memory hierarchies. With the move from sequential to parallel (multi- and many-core) architectures, the CPU-memory gap has only grown larger due to parallel components competing for limited bandwidth, the larger latency caused by the increased scale, and the potential need for memory coherence [15]. To help close the increasing gap, modern parallel architectures rely on complex memory hierarchies with multiple levels of caches, some shared among cores, some not, with main memory banks associated with specific

1st Summit on Advances in Programming Languages (SNAPL'15).
Eds.: Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett; pp. 1–14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

processors leading to non-uniform memory access (NUMA) costs, and complicated coherence schemes that can cause various performance anomalies. To obtain good performance from such hardware, programmers can employ very low-level machine-specific optimizations that carefully control memory allocation and the mapping of parallel tasks to specific processors so as to minimize communication between a processor and remote memory objects. While such a low-level approach might work in specific instances, it is known that this approach leads to vast inefficiencies in programmer productivity, and leads to software that is error-prone, difficult to maintain, and non-portable. It may appear that given these facts of hardware, we are doomed to designing and coding low-level machine-specific algorithms.

Developments on the software side limit the effectiveness of even such heroic engineering and implementation efforts: with the broad acceptance of garbage-collection frameworks and managed runtime systems, it may not even be possible for the programmer to exert much control over memory allocation, confining the applicability of locality optimizations to low-level languages such as C. Even in low-level languages, the challenges of locality management are amplified in the context of parallel programs, where opportunities for parallelism are expressed by the programmer and utilized by the run-time system by creating parallel threads as needed and mapping the threads to processors by using a thread scheduler. Such thread schedulers often make non-deterministic decisions in response to changes in the work load, making it very difficult if not impossible for the programmer to determine where and when a piece of computation may be performed.

Based on these observations, we conclude that locality should be managed by the compiler and the run-time system of the programming languages. Given the importance of locality in improving efficiency and performance, automatic management of locality in parallel programs can have significant scientific and broad impacts. But, such automatic management may seem hard: after all, how can the compiler and the run-time system identify and exploit locality? Fortunately, it is well known that programs exhibit natural locality. According to Denning, the reason for this is the "human practice of divide and conquer – breaking a large problem into parts and working separately on each." [24]. Research in the last decade shows that this natural locality of programs extends to nested parallelism as supported by languages such as Cilk [31], Fork/Join Java [41], NESL [14], parallel Haskell [39], parallel ML [29, 37, 30, 53], TPL [42], and X10 [16], Habanero Java [36]. This is because these languages take advantage of the same divide-and-conquer problem-solving methodology that leads to good locality. As a result, the compiler or runtime does not need to find the locality, just take advantage of it.

For example, consider the matrix multiply code in Figure 1. The eight recursive calls as well as the additions (recursively), can be performed in parallel, leading to abundant parallelism. Furthermore, the computation has abundant natural "temporal" locality: an $m \times m$ matrix multiply deep in the recursion will do $O(m^3)$ work but only need to load $O(m^2)$ memory allowing for significant reuse. There is also high "spatial" locality because quadrants can be laid out in a spatially coherent array. While this simple example considers only a highly structured algorithm, more irregular parallel algorithms also exhibit similar locality properties.

Previous work on (thread) scheduling has already shown how such natural locality in parallel software can automatically be exploited, and has proved bounds on a variety of machine models, such as multiprocessors with private caches using work stealing [2, 32], with shared caches using DFS scheduling [12, 18], or with trees of caches using space bounded schedulers [20, 11, 23]. A limitation of the previous results, however, is that they make very restrictive assumptions about memory allocation. The work-stealing results, for example,

```
function MM(A, B) =
  if A and B are small
    return SmallMM(A, B)
  else
    R_TL = MM(A_TL, B_TL) + MM(A_TR, B_BL)
    R_BL = MM(A_BL, B_TL) + MM(A_BR, B_BL)
    R_TR = MM(A_TL, B_TR) + MM(A_TR, B_BR)
    R_BR = MM(A_BL, B_TR) + MM(A_BR, B_BR)
    return compose(R_TL, R_BL, R_TR, R_BR)
```

Figure 1 Parallel Matrix Multiply; X_{YZ} refers to the four quadrants of the matrix X with Y and Z denoting the left/right and bottom/top parts. The + indicates matrix addition.

assume all memory is allocated on the stack, and the space bounded schedulers assume all space is preallocated and passed in. The results therefore do not apply to systems with automatic memory management and remain low-level in how the user has to carefully lay out their memory, making fine-grained dynamically allocated data structures particularly difficult to implement efficiently.

We believe that it is possible (and desirable) to manage locality automatically within the run-time system – specifically the memory manager and the scheduler – of a high-level garbage-collected parallel language. Why? At a fundamental level, such automatic management of locality is feasible because of the natural locality of certain software, and because all the information needed to manage it, such as the structure of the machine memory, the position of individual objects in memory, is readily available to and possibly controlled by the runtime system. But how?

In this paper, we suggest and briefly explore the following two ideas that we believe are important for taking advantage of locality for parallel programs.

1. Memory management and scheduling should be integrated as part of the same runtime system, such that management decisions are tightly linked with scheduling decisions.
2. The runtime heap structure should match the structure of the computation itself making it possible to position data and computation closely in hardware.

We describe an application of these ideas by considering purely functional nested-parallel programs executed on shared memory architectures such as modern multicore machines. Nested parallel computations, represented by fork-join programs is the mainstay of many parallel languages such as Cilk [31], Fork/Join Java [41], Habanero Java [36], NESL [14], parallel Haskell [39], parallel ML [29, 37, 30, 53], TPL [42], and X10 [16]. In this paper, we further limit ourselves to purely functional programs. While purely functional programming may seem like a significant restrictions, it often comes at no or little cost. For example, at Carnegie Mellon University, we teach the introductory undergraduate algorithms class [1], which covers a whole range of algorithms and data structures, including algorithms on sequences (arrays), binary search trees, and graphs by using the purely functional, nested-parallelism paradigm. The work presented in this paper was partly motivated from a desire to develop a compiler that generates fast parallel executables for modern multicore architectures from nested parallel programs written in functional languages.

Our approach to managing locality is to structure memory in a way to reflect the structure of the computation, as shaped by the scheduling decisions, and to couple memory management and computation via memory, on which they both operate. A key observation

behind our approach is that many parallel programs, both purely functional and impure, observe a *disentanglement* property, where parallel threads avoid side-effecting memory objects that might be accessed by other threads. Disentanglement is often quite natural, because entanglement often leads to race conditions. Taking advantage of disentanglement, we structure the memory as a tree (hierarchy) of heaps, also by observing scheduling decisions, thus coupling the computation, scheduling, and the memory. Taking advantage of disentanglement, we also couple garbage collection with the computation, completing the circle.

While we assume purely functional code here, there are relaxations to purity that appear to remain compatible with the techniques proposed here. We discuss some possible directions for future research in Section 7.

2 Preliminaries

We consider shared-memory, nested parallel computations expressed in a language with managed parallelism, where the creation and balancing of parallelism is performed automatically by the run time system of the host language. We allow arbitrary dynamic nesting of fork-join (a.k.a., "par-synch" or "spawn-synch") constructs (including parallel loops), but no other synchronizations. A nested-parallel computation can be represented as a Directed Acyclic Graph (*DAG*) of *threads* (a.k.a., "strands"), each of which represent a sequential computation uninterrupted by parallel fork and join operations. The vertices of the DAG represent threads and the edges represent control dependencies between them. The DAG can be thought of as a "trace" of the computation, in that all threads evaluated during a computation along with the dependences between them are represented in the DAG. We call a vertex a *fork-vertex* if it has out-degree two and a *join-vertex* if it has in-degree two. Much of the literature on parallel computing uses this characterization.

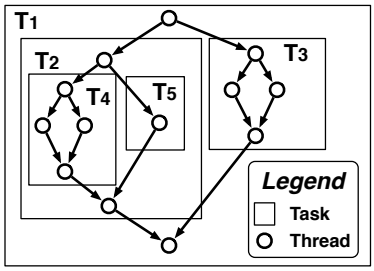

Figure 2 Tasks and threads.

We exploit an important hierarchical structure of nested-parallel computations. To see this structure, note first that in a nested-parallel computation, each fork-vertex matches with a unique join-vertex where the subcomputations started by the fork come together (join). We refer to such a subcomputation that starts at the child of a fork-vertex and completes at the parent of the matching join-vertex as a *task*. We also consider the whole computation as a task. We say that two threads are *concurrent* if there is no path of dependencies (edges) between them. We say that two tasks are concurrent if all their threads are pairwise concurrent. Figure 2 illustrates the DAG of an example fork-join computation along with its threads and some tasks. Tasks T_2 and T_3 are concurrent, whereas T_2 and T_4 are not.

At any given time during the computation we have a *task tree*, in which each node corresponds to a task. We can also associate with each leaf of the tree the thread that is currently active within that task. All these threads are said to be *ready* and a computational step can be taken on any subset of them. As expected, a fork operation executed on a leaf tasks T will create some number of new tasks that are children of T, and a join operation executed on a leaf task will remove the task from the tree. When the last child of a task T is removed, T becomes a leaf and hence has a *ready* thread. Note that although the DAG represents the trace of a computation when done, the task tree represents a snapshot in time

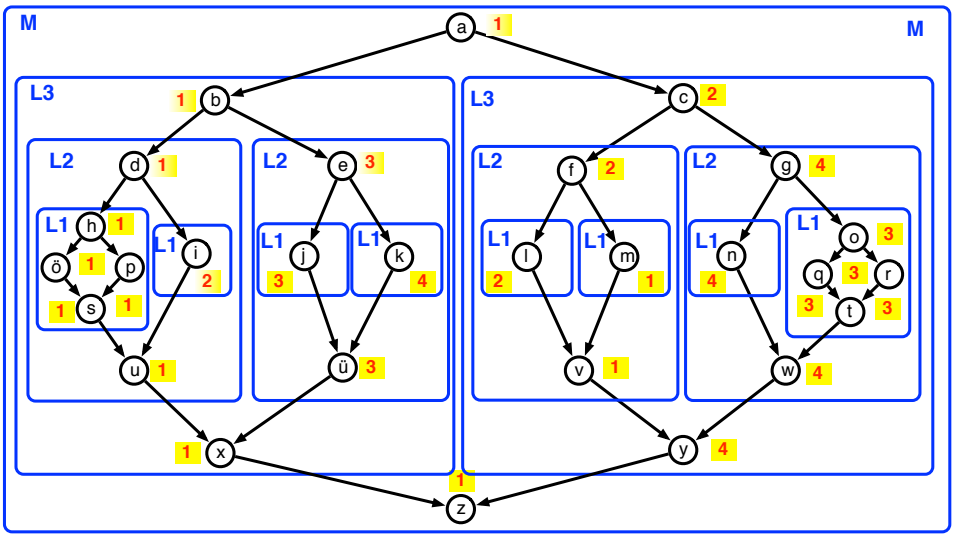

Figure 3 An illustration of how a parallel computation can be mapped processors and levels in the memory hierarchy. The label next to each thread (vertex) illustrates the processors, numbered from 1 to 4, executing the thread. Boxes illustrate parallel tasks. Each task is mapped to the lowest (fastest) level in the hierarchy that is large enough to hold its live data set; M denotes a memory bank and L1,L2, and L3 denote the levels of the cache.

of the computation. Assuming the computation is deterministic, the DAG is deterministic and does not depend on the schedule, but the task tree very much depends on the schedule.

3 Observations

We start by making some observations, using the following (hypothetical) example to help motivate them. Consider a nested-parallel program written with fork-join constructs. Suppose now that we run the program on an 4-processor machine, where each processor has its own L_1 and L_2 cache and four processors share an L_3 cache, by using a thread scheduler that performs load balancing so as to minimize completion time. Assume that when run, the program produces the computation DAG shown in Figure 3. Each thread is labeled with a letter, a through z (also with accents). We illustrate the schedule by labeling each thread with the number of the processor that executes it. For example, processor 1 executes thread a; processor 3 executes threads o, q, r and t. We illustrate a task by drawing a rectangular box around it. For the purposes of discussion, we assign each task a level in the memory hierarchy based on the size of the maximum memory needed by the live objects in that task, i.e., *its memory footprint*. For example, the task with root d is assigned to $L2$ because its live set does not fit into a level-1 cache but it does fit into a level-2 cache.

Disentanglement

Our approach to automatic locality management is based on a key property of nested-parallel computations, which we call *(memory) disentanglement*. In a nested-parallel computation, memory objects allocated by concurrent tasks are often *disentangled*: they don't reference each other. In purely functional languages such as NESL and pure Parallel ML, because of the lack of side effects, all objects allocated by concurrent tasks are disentangled. For example, in Figure 3, the memory object allocated by tasks rooted at b and c would be disentangled (they

can point to objects allocated by their parent a, but not to objects allocated by each other). In the presence of side effects, disentanglement is also the common form because side effects naturally harm parallelism and therefore are rarely used in a way to cause entanglement. For example, in Cilk, where imperative programming is readily available, disentanglement is encouraged, is the common case, and can be checked with a race detector.

Task Local Heaps

A key property of generational collection with sequential garbage collection is that the generations can be sized so they take advantage of the various levels of the cache. If the newest/smallest generation fits within the L1 cache, for example, and is flushed or compacted whenever it fills, then most accesses to that space will be L1 cache hits. Short lived data will therefore rarely cause an L1 miss. Similarly if the second generation fits within L2, slightly longer lived data will rarely cause an L2 miss. Our goal is to take advantage of this property in the context of parallel computations in which caches might be local or might be shared among subsets of processors. In our example, all the tasks that have L1 around them are scheduled on a single processor so as long as we use a generation within each one, most accesses will be L1 hits even if the total memory allocated is much larger than L1 (recall that the size refers to the live data at any given time). More interestingly the task rooted at b fits within L3 (which is shared). If we allocate a heap proportionally and share that heap among the subtasks, then their footprint will fit within L3. However this only works if we do not simultaneously schedule any of the tasks under the root c since together they would not (necessarily) fit into L3. We therefore would want to link the scheduling decision to heap sizes.

Independence

With disentanglement it is always safe to garbage collect any task in the task graph by only synchronizing descendant tasks. This allows garbage collection to proceed independently without the complications of a concurrent collector at various levels of the hierarchy. This is true even for a moving (copying or compacting) collector. It is even possible to collect an ancestor task with a non-moving collector when a descendant task is still running as long as the root set is properly accounted for.

GC Initiation

Since scheduling is performed at task boundaries, pieces of computation move at those points. Thus, if we coordinate the garbage collector with the scheduler, we can initiate garbage collections at scheduling points. For example when finishing the task rooted at o all the data that was accessed in that task (nodes o, q, r and t) is presumably still in the L1 cache. It would now be a good time to run a GC on the L1 heap to compact it so a smaller footprint has to be flushed from the cache, and later reloaded into another cache when used.

4 Hierarchical Memory Management

Based on the aforementioned observations we now outline a particular set of techniques for managing memory for locality. This is meant to be a concrete example and there are surely many variants. As mentioned previously, the key components are integrating the memory management with scheduling, and using a hierarchy (tree) of heaps. Organizing the

memory as a hierarchy of heaps will enable two key outcomes: 1) it will enable performing scalable, parallel and concurrent garbage collection, and 2) it will make it possible to map the hierarchy of heaps onto the caches of a hierarchical memory system, and 3) it allows the memory manager to collaborate with the scheduler to make effective use of shared caches by associating heaps with shared caches in the hierarchy.

Hierarchical, task- and scheduler-mapped heaps

We take advantage of the invariant that parallel tasks are disentangled – they create memory graphs that are disjoint (don't point to each other) – and partition memory into *subheaps* or *heaps*, which will be managed independently. Heaps can be of any size and can match the sizes in the cache hierarchy. For good locality, heaps are associated with *separately scheduled tasks* that start executing at a processor due to a scheduler task migration action, and heaps are merged to reflect the structure of the computation. Each heap contains all the live data allocated by its associated task, except for live data living in the heap of a migrated descendant task. The heaps are therefore properly nested, becoming smaller towards the leaves of the task tree. Each child heap is newer than the parent and therefore references only go up the heap hierarchy. A heap at the leaf corresponds to a computation that runs sequentially and thus can be garbage collected independently. We note that such a computation might have nested parallel tasks within it as long as the scheduler has not migrated them. When a task terminates its heap can be merged into its parent's heap. This is a logical merge and might or might not involve actually moving data.

Knot sets

The heaps in the hierarchy are "tied together" with knots; a *knot* is a pointer that points from the heap of a separately scheduled task H_c into its parent heap H_p. The knots can be used as a root set into H_p to allow for separate collection of H_p. Initially the knot set from H_c consists of only the closure for the task that created H_c. While the computation in the child runs this is a conservative estimate of what the child can reach. Whenever the child does a collection on its heap H_c the knot set to its parent can be updated, allowing more to be collected in the parent. Maintaining the knots therefore requires no read barriers (special code for every read), and allows garbage collection to proceed independently within each heap as long as internal heaps do not move their data. The only required synchronization is when accessing the knots, which is easy to implement with an atomic switch by the child.

An example

As an example, let's consider a hypothetical task A that starts as shown in Figure 4. We assume the task is migrated by the scheduler (perhaps to start the computation) and has a heap H_A of a size that fits in the L2 cache. The roots of the task point to two trees that are stored in the heap. For illustrative convenience, we draw the task (A) within its heap. We draw memory objects as squares and references between them as edges. We draw unreachable objects that can be reclaimed in green (appears gray in grayscale). After it starts executing, A performs some computation and forks two new tasks B and C that take as argument subtrees from H_A (Figure 5). Supposing that these tasks are executed in parallel by a scheduling action, we will create two separate heaps H_B and H_C for them. The arguments to B and C create the initial knot-set of H_A (depicted as dark squares). When forking, A creates the suspended join thread D (continuation) for when B and C complete,

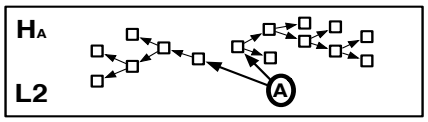

Figure 4 Start task A and its heap H_A.

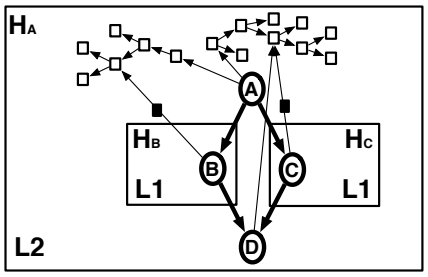

Figure 5 Task A forks two new tasks B and C. Their heaps H_B and H_C can be managed and collected independently.

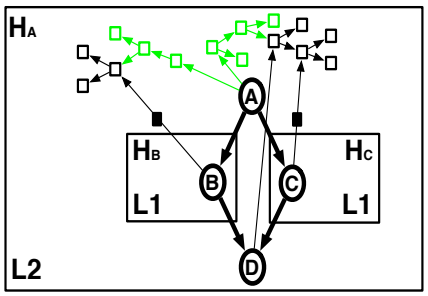

Figure 6 The suspended heap H_A is collected some time after B and C starts running.

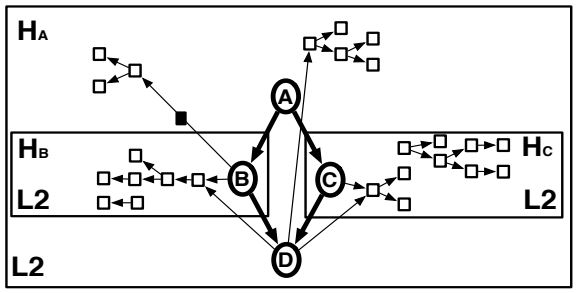

Figure 7 By the time tasks B and C complete, they have grown their heap by adding new objects.

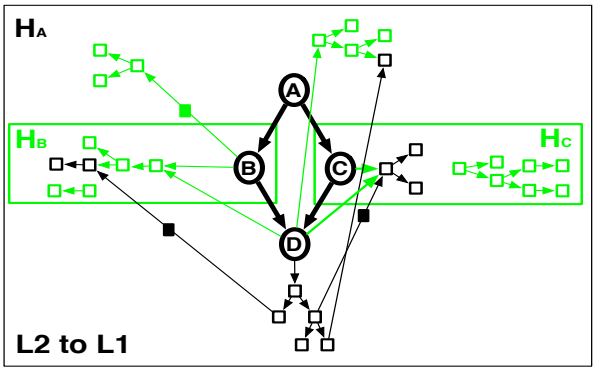

Figure 8 When B and C join D, their heaps are merged with H_A, the heap of A. A garbage-collection of H_A is shown some time after the merge.

which can also refer to memory in H_A. As B and C run, if needed, the suspended heap H_A can be collected by performing a non-moving garbage collection using the knot sets as roots. Figure 6 illustrates the reclaimed memory objects in green (light in grayscale).

The heaps H_B and H_C start out empty (trivially fitting into an L_1 cache) and grow as B and C allocate objects, triggering garbage collection. Since B (C) is independent from all other tasks running in parallel and its knot-set is empty, its heap is completely independent from other heaps. It can therefore collect anytime using any algorithm it chooses. In particular it can use a copying or compacting collector to ensure the footprint of the remaining live data is minimized. A good time for garbage collection is when the heap size approaches L_1; this would help keep the working set in L_1. If significant live memory remains after a collection, the heap size might be promoted to the next cache size as illustrated in Figure 7, where the tasks have been promoted to fit into the level-2 cache. When B and C complete (not necessarily at the same time), they pass their results to the join thread D. At this point, the heap H_B (or H_C) can be merged with its parent heap H_A and its knot sets can be dropped. In the example, after thread D starts running it creates a pair of the results returned to it by B and C. As D runs, its heap can be collected (Figure 8). Such a collection can reclaim many unreachable objects (essentially any object that does not contribute to the final result), possibly making the heap fit into L1.

5 Implementation and Evaluation

We are in the process of developing a compiler and run-time system for an extension of the Standard ML language that supports nested parallelism and the presented hierarchical memory management techniques (Section 4). The starting point for our implementation effort is the MLton compiler infrastructure [48, 58] and the shared-heap-multicore branch developed by Daniel Spoonhower as part of his dissertation work [55, 54]. MLton is an open-source, whole-program, optimizing compiler for Standard ML (SML) [47]. The current release of MLton includes a self-tuning garbage collector that uses copying, mark-compact, and generational collection algorithms, automatically switching between them at run time based on both the amount of live data and the amount of physical memory. MLton's combination of garbage collection algorithms is inspired by Sansom's dual-mode garbage collection [51]. The current release of MLton does not include support for using multiple processors or processor cores to improve the performance of SML programs.

Using our implementation, we plan to evaluate the quantitative effectiveness of our proposed techniques by developing a purely-functional nested-parallelism parallel benchmark suite modeled after the recently announced Problem Based Benchmark Suite (PBBS) [52], which include standard parallelism benchmarks as well as less traditional benchmarks using graph algorithms. We also plan to evaluate the approach more quantitatively by using it in our teaching.

6 Related Work

Scheduling for Locality

There have been thousands of papers that deal with the issue of locality in some way. In the following discussion we focus on techniques that (1) apply to parallel computations where tasks are created dynamically and mapped to processors by the runtime, (2) are useful for reasonably general purpose programs and data structures (e.g. not just dense arrays), (3) are suited for shared memory (or at least shared address space) cache-based architectures, and (4) for which at least something theoretical can be said about their properties. There has been plenty of work on models where the user maps their algorithms directly to processors (e.g. [56, 4, 57]), or for specific domains such as dense matrices (e.g. [17, 9, 7]).

There has been significant work over the past decade on trying to model and understand locality in a way that is independent of how a computation is mapped to processors, and then have a runtime scheduler somehow preserve this locality [2, 12, 32, 21, 10, 22, 13, 20, 23, 50, 11]. By relying an specific scheduling policies, the results from this work can asymptotically bound cache misses on a concrete parallel machine model in terms of abstract costs derived from the program. For example, earlier work showed that for a nested parallel computation with depth (span, critical path length) D and sequential cache complexity Q_1, a work-stealing scheduler on P processors with private caches will incur at most $Q_1 + O(PDM/B)$ cache misses [2], where M is the total size of each cache and B is the block size. Similarly a PDF scheduler on the same computation with a shared cache of size $M + O(PDB)$ will incur at most Q_1 cache misses [12]. A key point of these results is that cost metrics for a program that have nothing to do with the particular machine, Q_1 and D, can be used to bound costs on those machines. In addition to theory the approaches have been shown to work well in practice [18].

More recent work has considered deeper hierarchies of caches, involving levels of private and shared caches, and has shown that a class of space-bounded schedulers [20, 11, 23]

are well suited for such hierarchies. These schedulers try to match the memory footprint of a subcomputation with the size of a cache/cluster in the hierarchy and then run the computation fully on that cluster. Under certain conditions these schedulers can guarantee cache miss bounds at every level of the cache hierarchy that are comparable to the sequential misses at the same level. However in this case the abstract cache complexity is not the sequential complexity, but a relatively natural model [11] that considers every access a miss for subcomputations that don't fit in the cache and every access a hit for those that do. Cole and Ramachandran [23] describe a scheduler with strong asymptotic bounds on cache misses and runtime for highly balanced computations. Recent work [11] describes a scheduler that generalizes to unbalanced computations. Several heuristic techniques have also been suggested for improving locality with dynamic scheduling on such cache hierarchies [28, 40, 50].

Our work builds on ideas developed in this previous work, but as far as we know, none of this work has considered linking memory management with scheduling to improve locality, and the previous work assumes very limited memory allocation schemes.

Memory Management

There has also been significant work on incorporating locality into memory management schemes. Several explicit memory allocators, such as Hoard [8], TCMalloc [33], and SSMalloc [43], have been designed to be multithreading-friendly. These schemes create local pools of memory on each processor so that lock contention is reduced and so that freed memory is reused while still in the cache. All the approaches are heuristic; in our work, we have found that they can do exactly the wrong thing in certain contexts. A recent blog discussion [46] shows just how subtle memory allocation and management can be for a multicore system. The PIs have independently experienced many of the same issues discussed in the blog as well as several others. The problem is that the schemes know nothing about the scheduler or even programming methodology in which they are being embedded. The only information they have is the allocation request sequence at each thread.

For implicit memory allocation with garbage collection, maintaining locality becomes even more complicated, although it does give the runtime more flexibility because of the ability to move data. There have been dozens of proposed techniques for parallel garbage collection ([38]), and many of these suggest the use of processor or thread local heaps [34, 35, 25, 19, 27, 49, 3, 5, 45, 60]. Some of this work uses the idea of creating a nursery for each thread that handles recent allocations, and then a global heap for shared data [25, 27, 3, 5, 45]. If the nursery is appropriately sized, this organization helps keep recently generated and accessed data in local cache. The problem is that once promoted to the global heap, locality is lost. Furthermore since the work uses no knowledge of the scheduler, when a new task is scheduled on a processor, what is left in the cache will likely be evicted while still fragmented and not collected. Finally, as far as we know, none of this work has looked at multi-level hierarchies with private and shared caches nor has it tried to show any theoretical bounds on cache misses.

Another property of almost all memory management systems is that they try to be independent of the computation that is running on them and are only accessed through a minimal interface. Most work on garbage collection has treated the mutator (the program) as a black box that runs on a fully general purpose GC, perhaps with some heuristics that work well with typical programs (e.g. generations, more reads than writes) [38]. Although this might be a benefit for porting a GC across different programming languages, in practice most GCs are designed for a specific language. We believe that by making them generic

is giving up significant benefit that might be achieved by tying them more directly to the program or other aspects of the runtime such as the scheduler.

7 Discussions

In this paper, we only considered purely functional, nested parallel programs. While probably not straightforward, it appears possible to extend these ideas to more relaxed models of parallelism to include impure programs with side effects. This is because the primary assumptions that we make, disentanglement, is guaranteed by purely functional programs but it is, in general, a weaker condition. For example, side effects that remain local to a thread do not violate disentanglement. More generally, it appears possible to extend the work presented here to allow for arbitrary side effects by using techniques such as those used by two-level garbage collectors employed in memory managers of functional languages [26, 6, 44, 53]. For example, all objects reachable by mutable memory locations can be kept in a separate region of memory and treated specifically to ensure that memory management does not alter program invariants.

8 Conclusion

In this paper, we propose techniques for automatic locality and memory management in nested parallel programs. The basic idea behind these techniques is to take advantage of a key invariant of much parallel computations – their independence from each other, which then leads to disentangled memory regions – and structure memory hierarchically to reflect the structure of the computation and hardware, both of which are also usually hierarchical.

Acknowledgments. This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940, and by the European Research Council under grant number ERC-2012-StG-308246.

References

1 Umut A. Acar and Guy Blelloch. 15210: Algorithms: Parallel and seqential, 2015. `http://www.cs.cmu.edu/~15210/`.

2 Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

3 Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM*, pages 21–30, 2010.

4 Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, 2008.

5 Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage collection for multicore NUMA machines. In *MSPC'11: Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 51–57. ACM Press, June 2011.

6 Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI'11, San Jose, CA, USA, June 5, 2011*, pages 51–57, 2011.

7 Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph expansion and communication costs of fast matrix multiplication: regular submission. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 1–12, 2011.

8 Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS*, pages 117–128, 2000.

9 Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–57, 2006.

10 Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *In the Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, 2008.

11 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'11, pages 355–366, 2011.

12 Guy E. Blelloch and Phillip B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

13 Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

14 Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

15 S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and platform evolution for the next decade., 2005.

16 Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'05, pages 519–538. ACM, 2005.

17 Siddhartha Chatterjee. Locality, communication, and code generation for array-parallel languages. In *PPSC*, pages 656–661, 1995.

18 Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *ACM Symposium on Parallel Algorithms and Architectures*, SPAA'07, pages 105–115, 2007.

19 Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *PLDI*, pages 125–136, 2001.

20 R.A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

21 Rezaul Alam Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA*, 2007.

22 Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *SPAA*, 2008.

23 Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *ICALP*, 2010.

24 Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, July 2005.

25 Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL*, pages 70–83, 1994.

26 Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 70–83, 1994.

27 Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. In *ISMM*, pages 76–87, 2002.

28 Kayvon Fatahalian, Timothy Knight, Mike Houston, Mattan Erez, Daniel Horn, Larkhoon Leem, Ji Park, Manman Ren, Alex Aiken, William Dally, and et al. Sequoia: Programming the memory hierarchy. *ACMIEEE SC 2006 Conference SC06*, 0(November):4–4, 2006.

29 Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP'08: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, pages 119–130. ACM Press, September 2008.

30 Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. *The Journal of Functional Programming*, 20(5–6):537–576, November 2010.

31 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

32 Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA*, 2006.

33 Sanjay Ghemwat and Paul Menage. TCMalloc : Thread-caching malloc, 2010.

34 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

35 Maurice Herlihy and J. Eliot B Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992.

36 Shams Mahmood Imam and Vivek Sarkar. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ'14, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014.

37 Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. The design rationale for Multi-MLton. In *ML'10: Proceedings of the ACM SIGPLAN Workshop on ML*. ACM, 2010.

38 Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook : The Art of Automatic Memory Management*. CRC Press, 2012.

39 Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 261–272, 2010.

40 Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 217–228, 2008.

41 Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA'00, pages 36–43, 2000.

42 Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA'09, pages 227–242, 2009.

43 Ran Liu and Haibo Chen. SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proceedings of the Third ACM SIGOPS Asia-Pacific conference on Systems*, APSys'12, pages 15–15, Berkeley, CA, USA, 2012. USENIX Association.

44 Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, pages 21–32, 2011.

45 Simon Marlow and Simon L. Peyton Jones. Multicore garbage collection with local heaps. In *ISMM*, pages 21–32, 2011.

46 Apurva Mehta and Cuong Tran. Optimizing linux memory management for low-latency / high-throughput databases. `http://engineering.linkedin.com/performance/optimizing-linux-memory-management-low-latency-high-throughput-databases`, 2013.

47 Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised)*. The MIT Press, 1997.

48 MLton web site. `http://www.mlton.org`.

49 Takeshi Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA*, pages 377–390, 2009.

50 Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

51 Patrick Sansom. Dual-mode garbage collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 1991.

52 Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA'12, pages 68–70, New York, NY, USA, 2012. ACM.

53 K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Multimlton: A multicore-aware runtime for standard ML. *J. Funct. Program.*, 24(6):613–674, 2014.

54 Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

55 Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

56 Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

57 Leslie G. Valiant. A bridging model for multicore computing. In *Proc. 16th European Symposium on Algorithms*, 2008.

58 Stephen Weeks. Whole-program compilation in MLton. In *ML'06: Proceedings of the 2006 workshop on ML*, pages 1–1. ACM, 2006.

59 Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

60 Jin Zhou and Brian Demsky. Memory management for many-core processors with software configurable locality policies. In *ISMM*, pages 3–14, 2012.