The Myrmics Memory Allocator: Hierarchical, Message-Passing Allocation for Global Address Spaces

Spyros Lyberis Polyvios Pratikakis
Dimitrios S. Nikolopoulos [*]
Institute of Computer Science (ICS),
Foundation for Research and Technology (FORTH)
{lyberis,polyvios,dsn}@ics.forth.gr

Martin Schulz Todd Gamblin
Bronis R. de Supinski
Center for Applied Scientific Computing (CASC),
Lawrence Livermore National Laboratory
{schulzm,tgamblin,bronis}@llnl.gov

Abstract

Constantly increasing hardware parallelism poses more and more challenges to programmers and language designers. One approach to harness the massive parallelism is to move to task-based programming models that rely on runtime systems for dependency analysis and scheduling. Such models generally benefit from the existence of a global address space. This paper presents the parallel memory allocator of the Myrmics runtime system, in which multiple allocator instances organized in a tree hierarchy cooperate to implement a global address space with dynamic region support on distributed memory machines. The Myrmics hierarchical memory allocator is step towards improved productivity and performance in parallel programming. Productivity is improved through the use of dynamic regions in a global address space, which provide a convenient shared memory abstraction for dynamic and irregular data structures. Performance is improved through scaling on many-core systems without system-wide cache coherency. We evaluate the stand-alone allocator on an MPI-based x86 cluster and find that it scales well for up to 512 worker cores, while it can outperform Unified Parallel C by a factor of 3.7–10.7×.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Allocation/Deallocation strategies; D.4.7 [*Operating Systems*]: Organization and Design—Distributed systems, Hierarchical design

Keywords Parallel Memory Allocator, GAS

1. Motivation

The many-core era poses substantial challenges for programmers to harness the processing power of emerging hardware architectures. A dominant hardware paradigm is the cache-coherent shared memory machine, often paired with thread-based programming models. However, the complexity of hardware cache coherency protocols and the inefficiency of shared-memory programming models may limit this paradigm's lifetime, restricting coherency to a subset of cores on a processor or eliminating it altogether.

The main reason for this trend is that hardware cache coherency protocols do not scale as the number of on-chip cores increases. Further, at larger core-counts, verifying the hardware against timing-sensitive failures, such as race conditions, becomes so difficult that some researchers advocate abandoning cache coherency altogether [18]. However, programming non-coherent architectures can be tedious and difficult. Significant programmer expertise is required to structure code around message-passing protocols, and errors in these programs are notoriously hard to debug.

Cache-coherent, shared memory architectures with high core counts are, against general belief, also hard to program. The dominant shared-memory programming model uses threads, which resemble sequential programs and "feel" easier to programmers. However, threading requires the programmer to reason about implicit communication and interactions through shared memory. This complex, tedious and error-prone process makes threaded programs hard to test, debug and maintain. In general, writing good-quality, race-free, well performing and scalable multithreaded code is challenging [24].

Many commercial and academic programming models attempt to increase programmer productivity by abstracting away the difficult parts of parallelization and communication. However, notable examples like Intel TBB [23], OpenMP [28], and Cilk [5, 12] target cache-coherent architectures that do not scale well to high core counts. The most well-known programming models oriented towards non-coherent architectures belong to the Partitioned Global Address Space (PGAS) family of languages, such as Unified Parallel C [10], Titanium [17], X10 [8, 15], Co-array Fortran [27] and Chapel [7].

To schedule and to manage parallel tasks, existing PGAS and distributed programming models require strict control of the task footprint in memory. To meet this requirement, they restrict the use of dynamic memory in the program by making all dynamic allocation local to a task [11], or by statically limiting the available dynamic memory[1]. Conversely, an MPI program that uses dynamically allocated data structures, such as trees, must manually marshal and unmarshal them for communication. The programmer must allocate adequate space before a transfer takes place, and must rewrite any pointers after the transfer. Essentially, programmers implement the near equivalent of a PGAS or RPC runtime within the application. Therefore, dynamic memory management in the existing non-coherent programming models tends to be either:

[*] Also with the School of Electronics, Electrical Engineering and Computer Science, Queen's University of Belfast.

[1] The Berkeley UPC 2.14.0 we used in the evaluation imposes a static limit of 64 MB per thread for dynamically allocated memory.

- *Easy and expensive,* because the runtime does not know about application-specific structures and must assume the worst; or
- *Efficient and difficult,* because the programmer must control all transfers and synchronization — MPI, the *de facto* winner, is like "assembly" for parallel programming.

We bridge this gap by providing the programmer a middle ground. We propose a distributed memory allocation system that supports control of communication at a much higher level of abstraction than MPI-like programming. For this purpose, we borrow a well-known and well-studied construct in memory management literature: region-based memory management [30]. Regions are intuitive. They have been used to increase locality and to accelerate bulk allocation and deallocation. Successful coherent implementations of regions include stand-alone libraries and built-in programming language support. Regions offer the best of both worlds because they preserve the shared-memory abstraction while providing a mechanism to describe the desired structure of memory and control the locality and placement of memory objects. Gay and Aiken [13] have measured up to 58% faster execution times on memory-intensive benchmarks that use region-based memory management versus a conventional garbage collector.

Regions can be used to augment parallel programming models to express dynamic data structures more intuitively and efficiently. By allocating structure members in the same region, the programmer can express accessing, locking or transferring the whole structure simply by referring to the region itself. This capability enables reduced communication overhead for transferring complex, irregular, pointer-based data structures, which can be tightly packed by an underlying region-based memory allocator.

The contributions of this work are:

1. We design a distributed, Global Address Space (GAS), region-based memory allocator for non-coherent machines, which targets both MPI clusters and future many-core architectures with no support for cache coherency. To our knowledge, we are the first to introduce distributed region-based memory allocation.
2. We implement our algorithms in a runtime library that offers region-based GAS on large supercomputers, hiding MPI communication from the programmer.
3. We evaluate our techniques and implementation using representative benchmarks.

Our experimental results demonstrate that our distributed, region-based memory allocator can improve program performance by a factor of 3.7–10.7× compared to equivalent Unified Parallel C programs without support for regions.

The rest of the paper is organized as follows. Section 2 outlines the Myrmics runtime system and its stand-alone memory allocator. Section 3 presents the lowest software layer, the SLAB allocator, which manages a set of discrete heaps. Section 4 explains how the SLAB allocator is used to realize regions of the programming model. Section 5 discusses how multiple memory allocator instances cooperate to serve requests in parallel. Section 6 describes the development of benchmarks for the stand-alone version of the Myrmics memory allocator. We also measure benchmark runs in an MPI cluster and compare the Myrmics allocator to UPC.

2. The Myrmics Memory System

This paper presents the memory allocation system of the *Myrmics* runtime system, a task-based runtime for non-coherent many-core architectures. Myrmics uses distributed scheduler cores to manage memory and schedule parallel tasks on worker cores, as in other task-based programming models [2]. The Myrmics runtime is work-in-progress, but its memory allocator system is fully usable in a stand-alone fashion with an MPI backend. For the remainder of the paper, we will use the term "scheduler" to refer only to the memory allocation functionality of the Myrmics scheduler cores.

In the Myrmics runtime system, programs are written as collections of possibly recursive tasks. Each task is atomic and cannot communicate with other tasks during its execution. However, the tasks define their memory footprint and the runtime guarantees that this memory is made available locally to the core that will execute the task code before the execution starts. The runtime resolves task dependencies and ensures the tasks will have exclusive access to the memory that they request. The programming model has been proved to execute parallel programs in a deterministic order [29].

In this work we present the stand-alone Myrmics memory allocator, which does not handle task dependency analysis or scheduling. Instead, parallel programs using the stand-alone allocator are written as follows:

- A number of *worker* cores run the application code. Each worker executes as a separate MPI process.
- A number of *scheduler* cores cooperate to serve requests from workers to implement a global address space. Each scheduler also executes as a separate MPI process.
- Workers do not make local heap calls; instead, they send requests to a designated scheduler, which answers with objects that have unique system-wide addresses.
- Workers may send or receive data to/from other workers only by specifying heap objects that have been allocated previously by a scheduler. Data sent in this way is visible on the other end at the same addresses.
- Objects can be arbitrarily allocated in *regions,* which are used by workers for efficient bulk transfers of multiple objects. Regions can be hierarchical in nature, supporting sub-regions.

A large chunk of virtual address space is memory mapped at all worker cores during the initialization phase. Allocation in this space is done in parallel by the scheduler cores. The Myrmics memory allocator, therefore, implements a *global address space,* much like the shared-memory aspects of the PGAS languages. Scheduler cores are organized in a hierarchy and cooperate using messages to serve the worker cores, which run the parallel application. Workers run only a small part of the allocator functionality and communicate with schedulers to service all allocation and data transfer requests. Any allocation in a worker core returns a globally unique pointer for new objects or a globally unique region ID for new regions and sub-regions.

Worker cores can access any memory location in the global address space, but the data for the location must first be received. To receive the data, workers use data objects and regions in point-to-point communication calls. A sender and a receiver worker both specify which objects or regions they want to exchange. The Myrmics memory allocator ensures that data is sent to the correct locations. The receiver can then safely access the data at the same memory addresses as the sender. This mechanism enables pointer-based data structures to be traversable using the same pointers locally, assuming all needed data has been transferred. In the stand-alone memory allocator, the application must ensure that all needed data is requested before any pointers are accessed.

To scale to a large number of cores, Myrmics runs the main parts of the runtime, including its distributed memory allocator, on multiple dedicated scheduler cores. We use dedicated scheduler cores to optimize for locality of the runtime's metadata, remove allocator-related communication from the application's critical path, and amortize much of the allocator's cost.

This work bridges the gap between limited use of dynamic memory in PGAS systems and manual message-passing distributed programming, by providing support for distributed, region-based dynamic memory management and implementing a global address space from dynamic regions. Regions, also called *arenas* [16], are growable memory pools that contain objects or other sub-regions. Traditionally, regions are used for fast allocation and deallocation of objects that share the same lifetime, and to control and improve locality. We reuse this abstraction to enable distributed programs to use dynamic memory without requiring explicit data marshaling and unmarshaling or restricting the scope of dynamically allocated objects.

The Myrmics memory allocator allows the programmer to create regions and sub-regions, allocate objects dynamically within them, and use arbitrary pointers to create dynamic data structures. The distributed allocator guarantees a global address space among all cores so any region can be transferred to any core without translating any pointers in the allocated data. Consider, for instance, a dynamically linked list: to transfer the whole list in MPI, the program would have to traverse it, serialize the "next" pointers, to send each data item, and re-establish the pointers after the transfer. Conversely, a UPC program would also traverse the list and send each item, while using an expensive "fat" pointer to the next element. In comparison, the Myrmics memory allocator allows the programmer to allocate the whole list in a region (and possibly sub-regions). Thus, the whole region can automatically be packed and sent efficiently. After the transfer, the receiver can traverse the list using the existing, all-local, list pointers.

3. SLAB Allocator

3.1 Overview

The lowest layer of the Myrmics memory management system is the SLAB allocator. It manages the dynamic allocation and freeing of memory objects of any size organized in *slabs,* which are packed groups of same-sized objects.

SLAB allocation is a well-established method [6], widely employed for memory allocation in operating system kernels. Its primary advantages are that it has a simple implementation — allowing fast, constant-time allocate and free operations — and that it avoids external fragmentation because the kernel allocates a small variety of object sizes. Typically, an operating system will also benefit from caching objects that use slabs. For instance, if all allocations and frees of mutexes happen from the same set of memory addresses, then reinitialization of all fields of a freshly allocated mutex is often unnecessary.

In the taxonomy of memory allocation policies [21], SLAB allocation belongs to the simple segregated storage family. To minimize the code and to maximize cache efficiency, we use the same allocator for runtime system heap management and to implement the system calls for application heap management. The Myrmics allocator differs from existing segregated storage allocators in several ways. First, we observe that the runtime kernel uses only a few size classes. Applications in general tend not to use too many classes[2]; we target high-performance applications that tend to have even more disciplined memory requirements. Therefore, we relax the requirement that size classes must be a power of two and instead we support as many classes as requested with the restriction that every size must be aligned to the size of a cache line, such as 64 B. This versatility reduces fragmentation and usually leads to better cache utilization. Second, since we target message-passing architectures, we design the slabs so that their metadata are carefully separated from the data, which increases the efficiency of hardware transfers and facilitates moving whole regions with fewer operations.

[2] Johnstone and Wilson [21] measured that for typical applications 90% of all objects allocated were of just 6.12 different sizes, 99% of all objects were of 37.9 sizes, and 99.9% of all objects were of 141 sizes.

3.2 Design

The system uses two configurable sizes for the basic quantities of allocation. The *slab size,* set to 4 KB, is the basic unit used internally in each allocator instance to allocate a chunk of memory. The *page size,* set to 1 MB, represents the basic unit at which different allocator instances trade free address ranges. It is also the basic unit at which schedulers request memory from the operating system (or directly from the hardware, when the Myrmics allocator is used in bare-metal setups). Whenever a memory allocation request is completed, the requested size is adjusted upwards to a 64-B aligned *slot size.* Objects belonging to the same slot size are serviced from the same set of slabs.

To index memory, the allocator uses a custom 8-degree Trie library, which is tuned to fit into the minimum 64-B slot size. Tries support fast, constant-time searches. We prefer them over hash tables for their deterministic performance as well as their added abilities to offer approximate searches and ordered walks. Three different tries are used in the allocator: the *Used Trie* holds an entry for each full or partial slab that is in use, keyed by the slab starting address. The *Partial Trie* holds the head of a linked list for each slot size that is currently active, keyed by the slot size. The *Free Trie* holds an entry for each free range of slabs available in the allocator, keyed by the starting address of the range. We employ a number of performance optimizations, such as *(i)* preallocating empty slabs for commonly used slot sizes, *(ii)* avoiding frequent Trie updates through lazily returning free slabs and *(iii)* eliminating the referencing of intermediate slabs to support efficient allocation and freeing of arbitrarily large slot sizes.

Each allocator supports multiple *slab pools* that operate independently using their own sets of slabs. Moreover, upon creation of each pool, we specify which other pool will be used for its metadata. The separation of metadata from data is crucial to support efficient region-based communication. The "recursion" of slab pool metadata stops at the runtime kernel slab pool, which handles its own metadata, as we explain below.

The SLAB allocator, including the trie library, occupies roughly 4,000 lines of C code.

3.3 Usage for Runtime Kernel Heap

In order for Myrmics to be portable to distributed memory machines — which may have no operating system, as with accelerator processors — we use the same SLAB allocator for the application and for the runtime kernel heap management, through the use of separate slab pools. The kernel heap slab pool is an exception, in the sense that its metadata are kept in the same slab pool along with the heap data under allocation. This combination is not straightforward. For example, allocating a new 64-B object in the kernel may require new 64-B trie nodes that are recursively allocated by the same code path into the same memory space. Specifically, this behavior may run into two problems: where to allocate the dynamically allocated metadata (like trie nodes) for the kernel heap pool and how to bootstrap the system.

To solve the first problem, we treat the kernel heap slab pool specially, by imposing additional constraints for preallocating empty slabs. For all object sizes necessary for the allocator data structures, we ensure that a minimum of empty slabs is left after any allocation is finished. If there are too few, we raise a flag, and as soon as the (possibly recursive) allocation/free requests are served we replenish the empty slabs from the Free Trie as needed. This procedure guarantees that we can satisfy any kernel slab pool

```
// Region management
rid_t sys_ralloc(rid_t parent, int level_hint);
void  sys_rfree(rid_t region);

// Object management
void *sys_alloc(size_t size, rid_t region);
void  sys_free(void *ptr);
void  sys_realloc(void *old_ptr, size_t new_size,
                  rid_t new_region);
void  sys_balloc(size_t size, rid_t region,
                 int num_objects, void **objects);

// Communication
void  sys_send(int peer_worker_id,
               rid_t *regions, int num_regions,
               void **objects, int num_objects);
void  sys_recv(int peer_worker_id,
               rid_t **regions, int num_regions,
               void ***objects, int num_objects);
void  sys_barrier();
```

Figure 1. The stand-alone Myrmics memory allocator API

request solely from the preallocated empty slabs by setting bitmap bits and without perturbing trie structures, which could require further allocator requests. Thus, we allow allocator requests to recurse as needed, knowing that they can be fulfilled without further recursion when they reach the lowest pool.

We bootstrap the kernel heap slab pool by initially assigning the needed number of preallocated empty slabs in a linear fashion. During boot, kernel heap allocations receive objects from the predefined slabs and the kernel tracks which slots are allocated. To leave the bootstrap mode, we perform normal allocation calls for all tracked objects, which set up all needed data structures with new linearly allocated objects. Eventually, this process converges[3] and when all objects are accounted for, the system is bootstrapped and the linear allocation is abandoned in favor of the normal one.

4. Local Memory Allocation

4.1 Overview

The intermediate layer in the Myrmics memory allocator uses the SLAB allocator to support hierarchical regions that are local to a scheduler instance. Figure 1 lists the basic set of system calls that implement the programming model application interface. The user allocates a new region with the `sys_ralloc()` call under an existing parent region or the default top-level root region. A unique, non-zero *region ID*, which represents the new region, is returned. We explain the use of the optional `level_hint` in Section 5. A region is freed using the `sys_rfree()` call, which destroys the region, all objects belonging to it and its children regions.

A new object is allocated by the `sys_alloc()` system call, which returns a pointer to its base address. The object may belong to any user-created region or the default top-level root region, represented by region ID 0. Objects are destroyed by the `sys_free()` call and can also be resized and/or relocated to other regions by the `sys_realloc()` call. Since the programming model requires all memory allocation to be done through system calls that induce worker-scheduler communication, we also provide the `sys_balloc()` call, which allocates a number of same-sized objects in bulk and returns a set of pointers. This call minimizes communication for common cases like the allocation of table rows.

[3] It converges because kernel objects are smaller than 4 KB: most allocations complete using the empty slabs and only a few need new slabs that require new trie nodes.

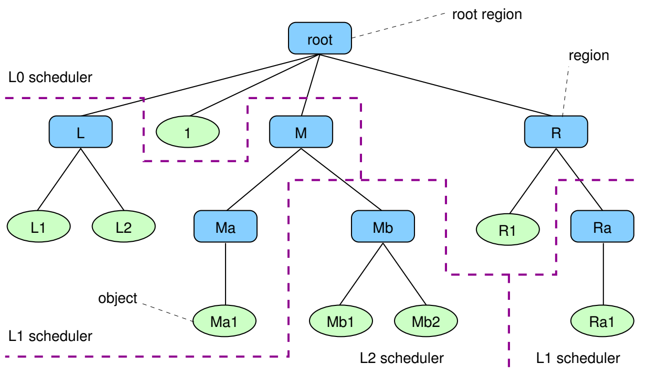


Figure 2. An example of a Region Tree. Dotted lines show how the Region Tree can be split among multiple schedulers.

We globally construct a *Region Tree,* such as the one shown in Figure 2, based on the relationship of user-allocated regions and objects. When the application starts, only the default "root" region exists. A scheduler core handles a part of the global region tree. This portion includes whole regions and any objects that belong to them, but not necessarily all of their descendants. The latter may belong to another scheduler core deeper in the hierarchy.

4.2 Design

We use a new slab pool to build each local region when it is created. We dedicate the equivalent of a separate heap to each region for many reasons. Our model hinges on communicating whole regions rather than individual objects, and the transfer of regions should therefore be as compact as possible. Packing region objects in dedicated slabs helps to isolate them from other regions and to enable communication on slab-based quantities. Further, a future design choice of migrating region responsibility among schedulers becomes feasible because different slab pools have carefully separated metadata. Allocating a new slab pool per region increases fragmentation, because partially filled and preallocated empty slabs are dedicated for the new region. We consider this tradeoff to be acceptable since many future object allocations in the region will happen quickly and will be compacted with other region objects, increasing communication efficiency and locality of region objects.

Apart from the creation of a new slab pool and the basic bookkeeping for the part of the Region Tree that is local, each scheduler contains four main data structures, which are also based on the same trie library. The first two are the *Used Ranges* and the *Free Ranges* Tries. The former tracks which local region uses which ranges of slabs. The latter contains ranges of slabs that the allocator can give to local slab pools that request more memory. These tries enable the allocator to determine in constant time which region is responsible for freeing an arbitrary pointer or which is the nearest set of free slabs to give to a slab pool under pressure, in order to keep region addresses as compact as possible.

For similar reasons, and using similar code paths, we use two more tries: the *Used Region IDs* tracks which region IDs are handled locally and the *Free Region IDs* contains the IDs that can be assigned to new regions. These tries enable quick translation of the globally unique, programmer-visible region IDs to the slab pools and region data structures, which are internal to each scheduler.

We use an adaptive mechanism that is based on watermarks to control the limit of external fragmentation. Initially, when the allocator is not under memory pressure, the number of slabs that

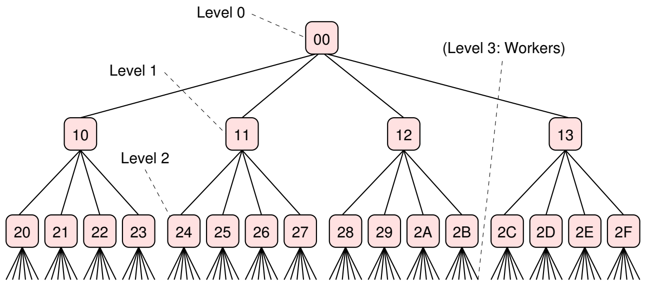


Figure 3. Organization of three scheduler levels, with a 4→1 scheduler-to-scheduler ratio. Assuming a 8→1 scheduler-to-worker ratio, each level 2 scheduler owns eight workers (not shown).

populate a new region's free pool is set to the high watermark. If and when many regions are requested by the application, the allocator reclaims increasing numbers of free slabs from the regions that have free memory above the current threshold. These are then used for the new regions. This process stops when all local regions have free memory equal to the low watermark, at which point the scheduler will communicate with its parent to request more pages. This policy reduces communication and balances increased locality of region objects with increased fragmentation.

4.3 Complexity

The local region layer adds a single trie lookup to most common-case operations for allocating and freeing objects. We consult the Used Ranges or Used Region IDs Tries to translate the pointer or region ID of the programming model API to a slab pool.

To allocate a new region, when the local allocator has enough memory and region IDs, it takes a new region ID from the Free Region IDs Trie and a range of slabs from the Free Ranges Trie. We create a new slab pool, initializing all related structures described in Section 3. In the case of increasing memory pressure between the high and low watermarks, local regions are visited — starting from the one last visited — possibly to trim free slabs. If we have already performed this process and still need more memory, we use inter-scheduler communication to request more memory.

Freeing regions is fast and independent of the number of allocated objects. We destroy the region slab pool by discarding its data structures altogether and returning all used and free slab ranges to the scheduler Free Ranges Trie. The programming model assumes that if the region has children regions, all of them are also destroyed. Thus, the complexity of hierarchical region freeing grows linearly with the number of children regions.

Transferring regions adds a new intra-scheduler operation on top of those required by the programming model API. *Region packing* accumulates a list of starting addresses and sizes that correspond to the memory usage of a region. The list encompasses all region objects as well as all children region objects. For each region involved, all slabs in the pool that are full or partially full are traversed in order[4] and a list is built by coalescing adjacent slabs on the fly, further increasing communication efficiency.

The local regions layer is 2,000 lines of C code.

[4] The Used Trie enables fast traversals by remembering the last visited node and following the appropriate turns on the trees to find the next one.

5. Distributed Allocation

5.1 Overview

A single memory allocator instance can service a limited number of requests from workers efficiently. All memory allocation calls involve scheduler-worker communication, so we must keep the latency of these operations low. As we have not yet implemented the rest of the runtime system, we do not know the full impact of other important scheduler duties, such as task arguments dependency analysis. Thus, the memory allocation system must scale to a high number of schedulers, the number and organization of which we will finalize in the future.

We organize the multiple scheduler instances, which contain only the memory allocator and basic interprocessor communication mechanisms, in a tree hierarchy, as Figure 3 shows. The tree has one top-level scheduler with a configurable number (equal to the *scheduler-to-scheduler ratio*) of next-level children. The scheduler tree descends for some levels. We attach multiple worker cores (equal to the *scheduler-to-worker ratio*) to each lowest-level scheduler. Processor cores can communicate directly only with cores that are one level above or below them in the hierarchy. This restriction targets future mesh-based, many-core hardware messaging layers by localizing communication patterns. It also helps to expose hardware locality constraints to the software architecture.

We divide work between schedulers based on the regions that are local to them. Figure 2 shows an example of how we can split the Global Region Tree among four schedulers. The mechanisms that we described in Section 4 handle all objects that belong to local regions. Worker access to objects and regions that are not local to the lowest-level scheduler incurs inter-scheduler communication so that the scheduler that is responsible for the region can handle the request. In Myrmics, the scheduling system primarily attempts to minimize this cost. The workers closest to the schedulers that handle the related regions should run the tasks. Another alternative, which we leave for future work, would migrate region metadata among schedulers in order to balance the load of irregular cases.

5.2 Design

The highest layer of the memory allocator is an expandable, generic, asynchronous, event-based server. If an incoming event refers to a local region, the server immediately processes it and responds. Otherwise, the server forwards the event to its parent or child schedulers. Replies from other schedulers are intercepted if they refer to pending actions for which the local scheduler awaits reply. Otherwise we forward them to the original requesters. Finally, we support reentrant events with saved local state for more complex situations in which we can handle part of the request locally or the final response should be assembled from multiple remote responses.

We assign regions to schedulers using both an optional level hint from the programmer and load-balancing criteria. The application may know how many levels of regions it will create, so it can help position the new region at an appropriate level within the region hierarchy, which the runtime translates to a scheduler level. Thus, we use the hint to estimate the "vertical" positioning of a region on the scheduler hierarchy. If the user does not supply a level hint, we assign new regions to lower-level schedulers. We use load balancing to determine the "horizontal" positioning; a non-leaf scheduler that must assign a new region to one of its children does so by selecting the one with the lowest region load. Schedulers periodically exchange upstream load information messages, whenever the previous reported load differs by a configurable threshold. Thus, higher-level schedulers always know the load status of their entire subtrees with a programmable degree of certainty.

The top-level scheduler initially owns all memory and all region IDs. During boot, middle- and low-level schedulers request chunks of both from their parent schedulers. The chunk, which represents a high watermark, is proportional to the total number of descendant schedulers. When a scheduler cannot service more requests by the internal balancing mechanism described in Section 4, or when a local request brings the free pools below a low watermark, the scheduler requests and receives more memory and/or region IDs from its parent. Extra memory pages and/or IDs are piggybacked to the last request to bring the scheduler back to the high watermark level without additional messages. Memory among schedulers is always traded in whole 1-MB page boundaries.

Schedulers know how to route requests for remote regions and objects by extending the Used Ranges and Used Region IDs Tries of non-leaf schedulers to include which child is responsible for the next hop. We tightly couple this mechanism to the memory and region ID assignment described above, so the information is readily available and does not require extra communication.

5.3 Complexity

When a worker core issues a memory allocation request that its leaf scheduler cannot handle, the request must pass through a number of schedulers in the hierarchy before they reach the the scheduler that can handle it. For each hop, we access the Used Ranges or Used Region IDs Tries to determine if (part of) the request can be handled locally. If not, but the tries contain an entry, we forward the request to the appropriate child scheduler, which is either directly responsible or knows to which of its children to delegate the request. If the address or region ID is not in the tries, we forward the request to the parent scheduler. Finally, if the top-level scheduler does not contain a corresponding entry, then we propagate error handling responses down the tree to indicate a programmer error. Programmer errors include freeing an invalid pointer and allocating an object in a nonexistent region. Thus, all non-local memory allocation requests incur a cost that is logarithmically proportional to the distance to the responsible scheduler in network hops. This cost is generally low since we assign tasks to workers as close to the data as possible.

The boundary cases of scheduler responsibilities present slightly more complex cases. In the example of Figure 2, creating region Mb as a child of region M requires a few additional messages between the two schedulers, since the L1 scheduler cannot fully complete the delegation to a child region for which the region ID is unknown at creation time. Handling of this and similar cases, such as deleting boundary regions or hierarchically packing regions that multiple schedulers own, is straightforward but generally requires more inter-scheduler communication. We create reentrant, stateful events that track each scheduler's local progress, until the operation completes successfully.

The distributed memory layer is written in 3,000 lines of C code.

6. Evaluation

6.1 Benchmark design

To evaluate the Myrmics memory allocator, we developed a number of microbenchmarks as well as two larger, application-quality benchmarks. We used a number of small test programs to test the allocator and to verify its correctness. Apart from these, we also present the results on four benchmarks: *(i)* a non-MPI, single-core, random object allocator that analyzes the fragmentation inside a region slab pool, *(ii)* a parallel, region-based, Barnes-Hut N-body simulation application, *(iii)* a parallel, region-based, Delaunay triangulation application and *(iv)* a comparison to Unified Parallel C for dynamically allocated lists.

In this section, MPI processes request all memory in advance from the Linux kernel, through large `mmap()` calls. This memory is subsequently managed by the memory allocator by intercepting all glibc allocation calls: we use a single runtime kernel slab pool for both the allocator and for MPI. The runtime kernel slab pool is private per processor, but we do not otherwise separate address spaces or vary privilege levels. For the stand-alone Myrmics allocator, the API of Figure 1 provides three calls. The `sys_send()` and `sys_recv()` calls take a target MPI rank and a variable number of region IDs or object pointers as arguments. Internally, we translate these arguments to lists of addresses and sizes (by packing regions and querying pointers) and then wrap around the respective `MPI_Send()` and `MPI_Recv()` calls. Also, a `sys_barrier()` call performs an MPI barrier among all worker cores.

All MPI-based measurements are done on the Lawrence Livermore National Laboratory *Atlas* cluster. Atlas has 1,152 nodes, each of them equipped with four Dual core AMD Opteron 2.4 GHz processors and 16 GB of main memory. The machines are interconnected with an Infiniband DDR network.

6.2 Fragmentation measurements

Our first benchmark, a serial program, allocates and frees objects within a single region. The application tracks all allocated pointers and randomly either allocates an additional object or frees a randomly chosen existing object. Figure 4a presents an execution for single-sized 192-B objects, with a 60% probability of allocating a new one and 40% probability to free one. The dotted gray line shows the application-requested size of all active objects with units on the left Y axis. The right Y axis shows the number of full and partial slabs. While the total number of objects grows, the allocator can compact most objects into full slabs; the number of partially filled slabs is kept constantly low.

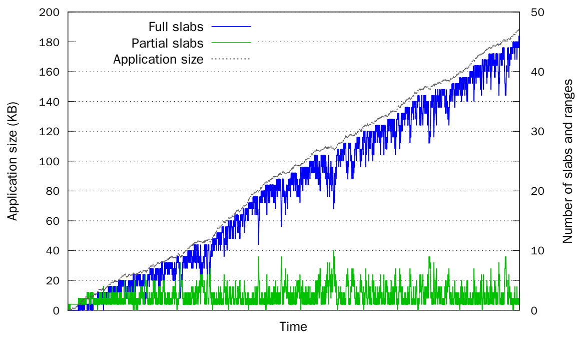
Full slabs are demoted to partial ones whenever a free is performed. Figure 4b, in which we vary the alloc/free probability in phases, shows this issue more clearly. The phases can be allocation-intensive or free-intensive as indicated by the slope of the application size curve. When frees are more common, full slabs become partial as they develop "holes" of 192 bytes. When the application returns to an allocation-intensive phase, first all holes in the partial slabs are discovered and plugged. We observe that this behavior is consistent with our prime concern to keep a region as packed in full slabs as possible, so that a region communication operation can access few address/size pairs.

In Figure 4c, the application runs with the same alloc/free phases, but uses six object sizes randomly during allocation. Three sizes are aligned to the slab size (64, 1024 and 4096 bytes), and the other three are not (192, 1536 and 50048 bytes). Behavior is similar to the previous measurements, although the large objects (4096 and 50048 bytes) consume correspondingly more full slabs and thus the ratio of full to partial slabs is much greater.

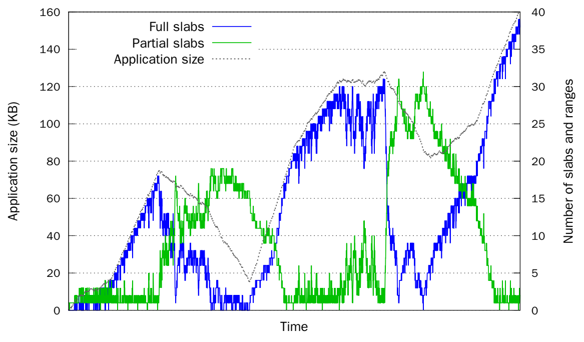
6.3 Barnes-Hut N-Body simulation

The second benchmark is a 3D N-body simulation application that calculates the movement of a number of astronomical objects in space as affected by their gravitational forces. To avoid $O(N^2)$ force computations, the Barnes-Hut method approximates clusters of bodies that are far enough from a given point by equivalent large objects at the clusters' centers of mass.

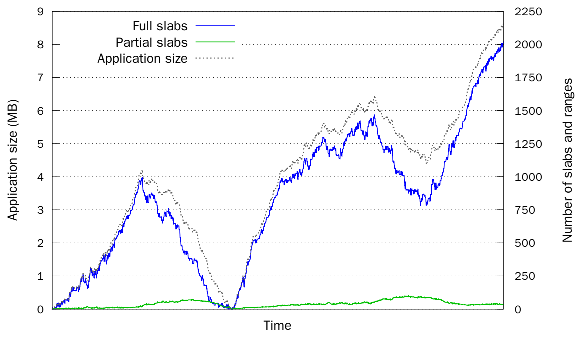
From the many variations of the Barnes-Hut method in the literature, we choose to start with the Dubinski 1996 approach [9], which is a well-known MPI-based implementation. This approach dynamically allocates parallel trees, parts of which are transferred among processors. Thus, it is a prime candidate for region-based memory allocation. Dubinski employs index-based structures with non-trivial mechanisms to allow for efficient transfer, pruning and



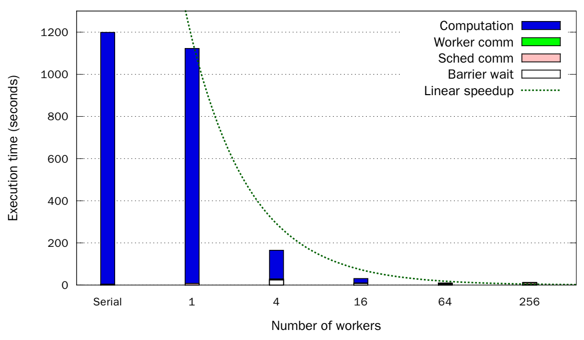
(a) Fragmentation, 192-B objects, 60% allocs, 40% frees



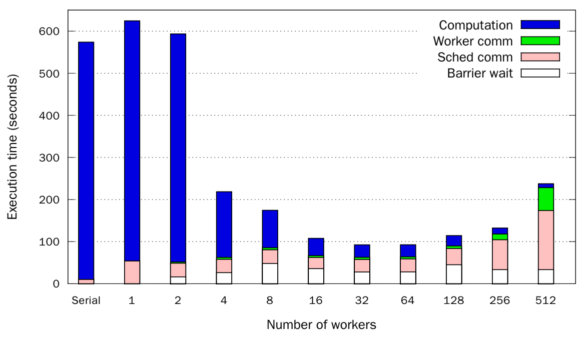
(b) Fragmentation, 192-B objects, varying probabilities



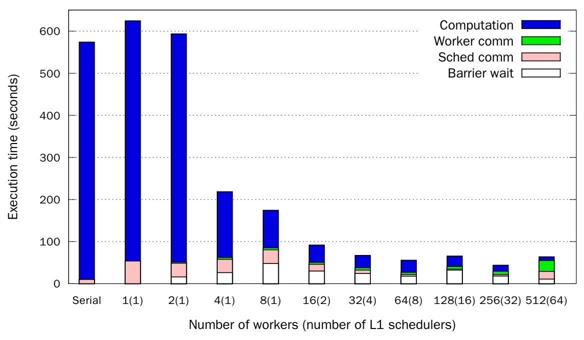
(c) Fragmentation, 6 object sizes, varying probabilities



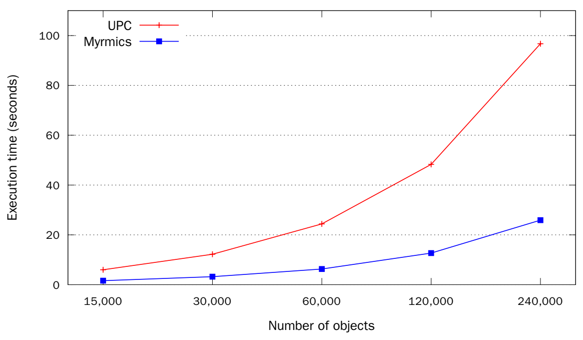
(d) Delaunay, 5M points, Single scheduler



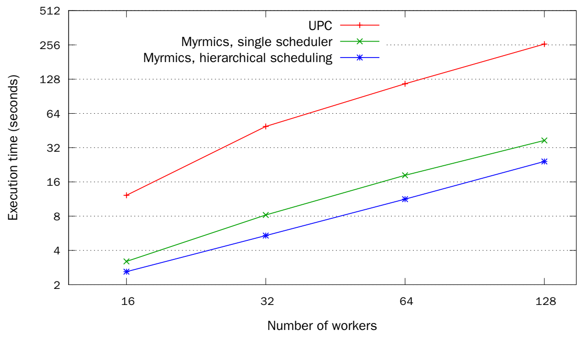
(e) Barnes-Hut, 1.1M bodies, Single scheduler



(f) Barnes-Hut, 1.1M bodies, Multiple schedulers



(g) UPC, 16 workers, varying number of objects



(h) UPC, 30K objects, varying number of workers

Figure 4. Evaluation of the Myrmics memory allocator using an MPI communication layer

grafting of subtrees over MPI. Our programming model supports a much more intuitive, pointer-based implementation in which we can easily graft trees since pointers retain their meaning after data transfers. MPI-based applications commonly resort to complex, "assembly-like" techniques to marshal data efficiently for transfers. The Myrmics allocator automates this tedious task.

In our benchmark, each worker core builds a local oct-tree in each simulation step for each body that it owns. We build the tree with each level belonging to a different memory region. The bounding box of the local bodies is communicated in pairs with all other workers, which compute based on that portion (*i.e.*, number of levels) of the local tree that must be sent to the communicating peer. We send the respective regions en masse. After we fetch and graft the portions of the remote trees, we perform the force simulation and body movement in isolation. A recursive bisection load-balancing stage in which we split processors into successively smaller groups follows each simulation step. We cut and exchange bodies along the longest dimension, balancing the load based on the number of force calculations that each body performed in the previous iteration. The recursive bisection load-balancer requires that the number of worker cores is a power of two.

Figure 4e presents application scaling on up to 512 worker cores with a single scheduler core. For each run, stacked bars show the average time of each worker. Time is spent either communicating with other workers ("Worker comm"), communicating with the scheduler via any other API call ("Sched comm") or doing local work ("Computation"). The first bar, marked "Serial" on the X axis, shows a single-core run in which the scheduler and a single worker are on the same processor. The next bar shows the scheduler and the single worker on separate processors. This distribution increases the cost of scheduler communication for the same work from 2% to 8%. For more worker cores, scaling is irregular, which is a data-dependent feature of the recursive bisection load balancer and the Barnes-Hut cell opening criterion, which needs more tree levels when any cell dimension exceeds certain quality constraints. Replacing the cell opening criterion gives much smoother scaling results, but unfortunately sacrifices simulation accuracy. A second observation concerns the scheduler communication time. As the application scales, each worker requires fewer allocations for its own tree, but the scheduler services more workers and its latency increases. This increase becomes a problem as early as in 32 cores, after which it grows worse. Last, worker communication becomes a bottleneck after 256 cores and overtakes the simulation time at 512 cores. Thus, the given problem size cannot scale further, which is a known limitation of this Barnes-Hut algorithm [9].

Figure 4f verifies our hypothesis that we can successfully distribute the memory allocation on multiple schedulers. In these experiments, up to eight workers are dedicated to a single scheduler. When multiple schedulers are present, they are organized in a two-level tree with a single top-level scheduler. The parenthesized number in the X axis specifies the number of leaf schedulers that we use. As expected, the scheduler communication time drops consistently as the application scales up to 128 workers. After that, the increased work needed to fetch all remote trees also involves the scheduler to pack all the remote regions; this work appears as scheduler communication time.

6.4 Delaunay triangulation

Our third benchmark, a Delaunay triangulation algorithm, creates a set of well-shaped triangles that connect a number of points in a 2D plane. Delaunay triangulation is a popular research topic with many serial and parallel algorithms. We base our code on the implementation of the serial Bowyer-Watson algorithm by Arens [1]. The algorithm adds each new point into the existing triangulation, deletes the triangles around it that violate the given quality constraints and re-triangulates the convex cavity locally. We use the optimization that Arens described to walk the neighboring triangles in order to determine the triangles that build the cavity quickly.

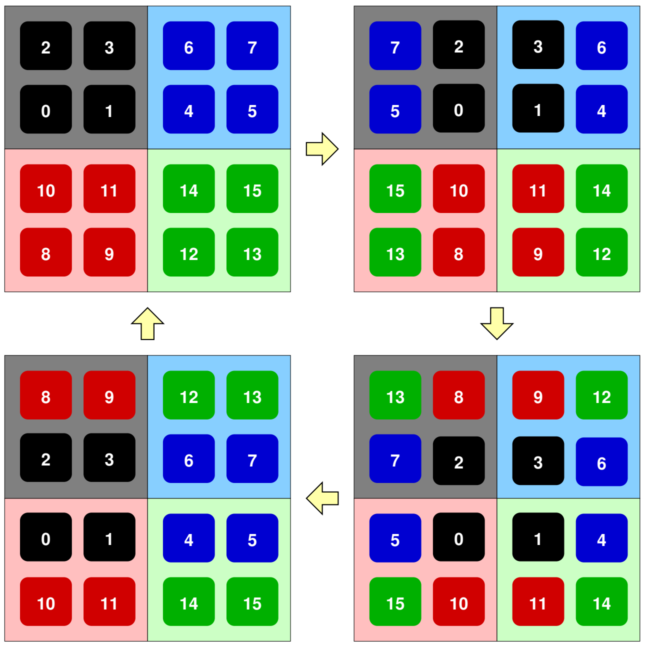


Figure 5. Parallelization of the Bower-Watson algorithm on four cores. Each core works on four sub-quadrants, which successive rotations to the right, down, left and up communicate among cores.

The Bowyer-Watson algorithm is difficult to parallelize, so it is an active research topic; Linardakis [25] wrote an extensive survey. State of the art distributed memory approaches combine algorithms of great complexity, such as graph partitioning and multiple passes that handle the borders of the decomposition. Understanding and modifying these algorithms to use regions effectively is beyond the scope of testing and evaluating the memory allocator, so we follow a simpler parallelization approach.

Our benchmark uses a grid decomposition to divide the 2D space statically into a number of regions equal to four times the number of worker cores. Each region holds the triangles with circumcenters within its bounds. All regions are at the lowest level of a hierarchy with a degree of four; *e.g.*, the top-level master region owns the whole space and its four children own one fourth of the space. Initially, after we create all regions, the space is empty except for placeholder triangles that form the borders. A single core begins the triangulation process by inserting a small number of points up to a limit. We dynamically allocate all triangles into the appropriate last-level regions of the hierarchy, according to the triangle centers. When we have inserted enough points to create an adequate number of triangles, the core delegates the four quadrants to three other cores and to itself and the algorithm recurses with four times more workers.

Apart from the first step, in which a single core owns the entire space, points near the borders of the space owned by a core may need to modify triangles that belong to other cores. Our algorithm postpones the processing of these points, when three *re-triangulation* phases occur. Figure 5 shows the concept used with four active cores. The *main triangulation* is in the top-left, where each core owns a quadrant of space comprised of four sub-quadrants, which are the regions one level below in the region hierarchy. The first re-triangulation uses communication with other cores and rotates the four sub-quadrants to the right. For example, a point that needs triangles from regions 3 and 6 would be postponed in the main triangulation, but handled successfully by the blue core in the first re-triangulation. The two next re-triangulations rotate the sub-quadrants down and left, while a fourth rotation brings the sub-quadrants upwards back to their original position, in order to be split to more workers[5].

Figure 4d shows the results for a triangulation with 5 million points. The dotted line represents the ideal scaling. We find that the scaling is superlinear due to the increased caching effects that the division of work has over the approximately 650 MB dataset. This memory locality effect is also apparent from the difference between the serial run and the one in which a single worker communicates with a single scheduler. In contrast to the Barnes-Hut runs, the two-process run is faster despite the MPI communication.

6.5 Comparison to Unified Parallel C

With our fourth benchmark, we compare the Myrmics memory allocator to Unified Parallel C (UPC) [10]. We use the Berkeley UPC 2.14.0 for these measurements. For the most faithful comparison possible, we instruct the UPC compiler to use its MPI backend for interprocess communication.

Each worker core (in Myrmics) or thread (in UPC) begins by dynamically building a linked list of objects. After all cores are done, an all-to-all communication pattern happens in multiple stages, separated by barriers. In each stage, a pair of workers exchange their lists, the receiving core modifies all objects and the lists are swapped back to their original owners.

In UPC, we allocate the list nodes in the shared address space of each thread using the `upc_alloc()` call. In the exchange phase, each thread fetches a list node locally with `upc_memget()` for the node, modifies it and returns it to its owner with `upc_memput()`. In Myrmics, each worker creates a region and then allocates all list nodes inside it, which supports a more efficient exchange. To fetch a remote list, a worker fetches the whole region. We traverse the list nodes by following the pointers locally. When the whole list is modified, we send the region back in one operation.

Figure 4g shows how the benchmark performs in UPC and Myrmics. For both implementations, we use 16 worker cores/threads; in Myrmics a 17^{th} core runs the scheduler. All list nodes are 256 B (including the shared pointer to the next node), as dynamic memory allocation requests for typical applications are on average less than 256 B [4]. The X axis shows the number of objects that are allocated for each worker list. Myrmics performs 3.7–3.9 times better. Sending or receiving the whole list in one call more than compensates for communication between the scheduler and worker, which happens for every memory allocation, and region packing, while we must communicate the list nodes one by one in UPC.

Figure 4h shows that the benchmark scales to more than 16 workers; the time scale on the Y axis is logarithmic. We keep the list size constant at 30,000 objects per core. We could not use larger problem sizes due to UPC's limits on total shared memory available. When using a single scheduler core, Myrmics outperforms UPC by a factor of 3.8–7.0×. The scheduler overhead can be further improved when using hierarchical scheduling, which makes Myrmics 4.6–10.7× faster than UPC.

7. Related Work

Serial region-based memory management. Tofte and Talpin introduced managing memory in regions for serial programs in 1997 [30] as a programming discipline to facilitate mass deallocation of dead objects in languages, replacing the garbage collector. Memory is managed as a stack of regions and static compiler analysis determines when regions can be scrapped in their entirety, thus avoiding the expensive operations of freeing or garbage-collecting dead objects individually. Gay and Aiken implement RC [13], a compiler of an enhanced version of C for dynamic region-based memory management that supports regions and sub-regions. RC focuses on safety: reference counts are kept to warn about unsafe region deletions or to disable them. The authors claim up to 58% improvement over traditional garbage collection-based programs. Berger et al. [3] verify that region-based allocation offers significant performance benefits, but the inability to free individual pointers can lead to high memory consumption.

Parallel region-based memory management. To our knowledge, our work is the first to introduce parallel region-based allocation. Titanium [17] uses "private" regions for efficient garbage collection, in the same way as serial region-based allocators do. There is some tentative support for "shared" regions, which are implemented inefficiently with global, barrier-like synchronization of all cores. Gay's thesis [14] provides some details on the Titanium shared regions and briefly mentions a sketch of a truly parallel implementation as future work. Parallel regions in Myrmics must not be confused with the X10 language regions [8], which are defined as array subsets and not as arbitrary collection of objects.

Partitioned Global Address Spaces. PGAS languages are specifically designed to offer parallel semantics by differentiating between local and global memory accesses. Unified Parallel C (UPC) [10] is a popular example: it extends C by providing two kinds of pointers: private pointers, which must point to objects local to a thread, and shared pointers, which point to objects that all threads can access but may have affinity to specific cores. The Berkeley UPC compiler [20], which is a reference implementation, translates UPC source code to plain C code with hooks to the UPC runtime system, which manages shared memory aspects. Other well-known PGAS languages are X10 [8, 15], which defines lightweight tasks (activities) that run on specific address spaces (places), Co-Array Fortran [27], which extends Fortran 95 to include remote objects accessible through communication, Titanium [17], which extends Java to support local and global references and Chapel [7], which is a language written from scratch that aims to increase high-end user productivity by supporting multiple levels of abstractions.

Our programming model resembles PGAS since we base communication on data structures. Myrmics, however, does not pin object and region locality to cores; a task can specify any accesses and the runtime attempts to schedule the task close to the data.

Shared memory parallel allocators. For thread-based, shared-memory architectures, Hoard [4] is considered one of the best parallel memory allocators. Hoard implements a small number of per-processor local heaps, which are backed by a global heap when they run out of memory, which is backed in turn by the operating system virtual memory system. While Hoard focuses on increased throughput, Michael [26] improves on multi-threaded, lock-based allocators by presenting a scalable lock-free allocator that guarantees progress even when threads are delayed, killed or deprioritized by the scheduler. MAMA [22] is a recent high-end parallel allocator that introduces client-thread cooperation to aggregate requests on their way to the allocator. McRT-malloc [19] follows a different approach, by implementing a software transactional memory layer to support concurrent requests; threads maintain a small local array of bins for specific, small-sized slots and they revert to accessing a public free list to get more blocks; larger slot sizes than 8 KB are directly referred to the Linux kernel.

Our work resembles, in many respects, parallel memory allocators that use heap replication. Our schedulers trade address ranges hierarchically and serve requests from these ranges. In the MAMA

[5] Our algorithm assumes each point can be triangulated within two adjacent sub-quadrants and requires the number of workers to be a power of four.

paper, the authors describe a three-way tradeoff for memory allocators: they can only feature two of the benefits of space efficiency, low latency or high throughput. The Myrmics memory system sacrifices space efficiency (memory is hoarded by multiple schedulers and preallocated for region usage) but offers high throughput, low latency and also compactness for memory objects inside regions.

8. Conclusion

This work presents the design, implementation and evaluation of the hierarchical memory allocator of the Myrmics runtime system. To our knowledge, our implementation is the first distributed region-based memory allocator. As the number of available cores continually scales, we must evolve programming models towards easier, or more automated parallelization. Scalable memory allocation is a basic prerequisite for this transformation. Hierarchically organized region-based memory allocation is an interesting approach with many benefits for parallel programmers. It offers the programmer better control over memory management, abstracts tedious communication primitives, and allows the programmer to expose locality constraints naturally to the scheduling subsystems.

Acknowledgments

The research leading to these results has received funding from the European Union 7^{th} Framework Programme [FP7/2007-2013], under the ENCORE (grant agreement n^{o} 248647) and TEXT (grant agreement n^{o} 261580) Projects.

This article (LLNL-CONF-545875) has been authored in part by Lawrence Livermore National Security, LLC under Contract DE-AC52-07NA27344 with the U.S. Department of Energy. Accordingly, the U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for U.S. Government purposes.

References

[1] C. Arens. The Bowyer-Watson Algorithm; An efficient Implementation in a Database Environment. Technical report, Delft University of Technology, January 2002.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3): 404–418, 2009.

[3] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering Custom Memory Allocation. In *OOPSLA '02: Proc. 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–12.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGPLAN Not.*, 35:117–128, November 2000.

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP '95: Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216.

[6] J. Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USTC '94: Proc. 1994 USENIX Summer Technical Conference*, pages 87–98.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *IJHPCA*, 21(3):291–312, 2007.

[8] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538.

[9] J. Dubinski. A Parallel Tree Code. *New Astronomy*, 1(2):133–147, 1996.

[10] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Language Specifications v1.1.1. October 2003.

[11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *SC '06: Proc. 2006 ACM/IEEE Conference on High Performance Networking and Computing*.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI '98: Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223.

[13] D. Gay and A. Aiken. Language Support for Regions. In *PLDI '01: Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80.

[14] D. E. Gay. *Memory Management with Explicit Regions*. PhD thesis, UC Berkeley, Berkeley, CA, USA, 2001.

[15] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A Performance Model for X10 Applications. In *X10 '11: Proc. ACM SIGPLAN 2011 X10 Workshop*.

[16] D. R. Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software Practice and Experience*, 20:5–12, January 1990.

[17] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick. Titanium Language Reference Manual, Version 2.19. Technical Report UCB/EECS-2005-15, EECS Berkeley, November 2005.

[18] J. Howard, S. Dighe, Y. Hoskote, S. R. Vangal, and D. Finan. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC '10: Proc. 2010 IEEE International Solid-State Circuits Conference*, pages 108–109.

[19] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: A Scalable Transactional Memory Allocator. In *ISMM '06: Proc. 2006 International Symposium on Memory Management*, pages 74–83.

[20] P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *ICS '03: Proc. 17th International Conference on Supercomputing*, pages 63–73.

[21] M. S. Johnstone and P. R. Wilson. The Memory Fragmentation Problem: Solved? *SIGPLAN Notices*, 34:26–36, October 1998.

[22] S. Kahan and P. Konecny. "MAMA!": A Memory Allocator for Multithreaded Architectures. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 178–186.

[23] A. Kukanov and M. Voss. The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4), Nov. 2007.

[24] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[25] L. Linardakis. *Decoupling Method for Parallel Delaunay Two-Dimensional Mesh Generation*. PhD thesis, College of William & Mary, Williamsburg, VA, USA, 2007.

[26] M. M. Michael. Scalable Lock-Free Dynamic Memory Allocation. *SIGPLAN Notices*, 39:35–46, June 2004.

[27] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[28] OpenMP ARB. OpenMP Application Program Interface, v. 3.1. `www.openmp.org`, July 2011.

[29] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. A Programming Model for Deterministic Task Parallelism. In *MSPC '11: Proc. 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness*, pages 7–12.

[30] M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109 – 176, 1997.