DRASync: Distributed Region-based memory Allocation and Synchronization

Christi Symeonidou
FORTH-ICS
chsymeon@ics.forth.gr

Polyvios Pratikakis
FORTH-ICS
polyvios@ics.forth.gr

Dimitrios S. Nikolopoulos
Queen's University of Belfast
d.nikolopoulos@qub.ac.uk

Angelos Bilas
FORTH-ICS
bilas@ics.forth.gr

ABSTRACT

We present DRASync, a region-based allocator that implements a global address space abstraction for MPI programs with pointer-based data structures. The main features of DRASync are: (a) it amortizes communication among nodes to allow efficient parallel allocation in a global address space; (b) it takes advantage of bulk deallocation and good locality with pointer-based data structures. (c) it supports ownership semantics of regions by nodes akin to reader-writer locks, which makes for a high-level, intuitive synchronization tool in MPI programs, without sacrificing message-passing performance. We evaluate DRASync against a state-of-the-art distributed allocator and find that it produces comparable performance while offering a higher level abstraction to programmers.

General Terms

Memory Management, Distributed, Synchronization, Locking

1. INTRODUCTION

MPI programs using dynamic data structures with pointers, such as graphs or trees, must either marshal and unmarshal data in order to transfer them among nodes, or use non-intuitive representations for dynamic data structures e.g. represent graphs with matrices and limit list sizes. Distributed memory models, such as the Partitioned Global Address Space (PGAS) languages [**? ? ?**], support global pointers, but they do not perform very well, because implicit communication often causes unnecessary and expensive messages to be exchanged.

Recent work uses regions to allocate dynamic data structures to avoid marshalling and unmarshalling of pointers for communication. Regions allow intuitive representation of pointer-based data structures that is easy to send and receive [**? ?**] in bulk, without e.g., traversing a list to pack it or transferring it one item at a time. Some of these systems still maintain a message-passing synchronization structure where transferring a region and its data still requires the programmer to code explicit cumbersome send and receive operations.

This paper takes the region abstraction further by extending regions with ownership semantics. Therefore, in addition to providing an intuitive way to manage dynamic pointer-based data structures, our regions can also be used in MPI programs to synchronize on data, instead of using barriers or explicit rendez-vous synchronization. This helps parallel programmers reason about ownership of data at a high level, without having to manually track which node owns the last copy of an object, or predict where an object will be required next. Each process can operate on data by acquiring the containing region and releasing it at the end of computation for other processes to acquire. Finally, our approach abstracts both the location of data and the communication for transporting the region.

Overall, this paper makes the following contributions:

- We propose a high-level synchronization abstraction for MPI programs with dynamic pointer-based data structures, in which dynamic regions are treated like reader-writer locks.
- We implement DRASync, a region-based distributed allocator for MPI programs.
- We evaluate our implementation on two benchmarks and compare with an existing state-of-the-art distributed allocator. Our allocator performs equally well on average, while offering an intuitive mutual exclusion synchronization mechanism for MPI programs and abstracting the location of data from the programmer.

2. MEMORY ALLOCATOR DESIGN

We implement DRASync as a runtime library for MPI programs. We provide an API for creating and deleting regions, allocating memory for objects inside these regions, and also for acquiring and releasing ownership of regions. Our approach abstracts the need to explicitly transfer these regions across nodes by the programmer.

2.1 Regions

A region is a collection of memory pages, within which the program can allocate objects or other subregions. Regions enable transferring large amounts of data between nodes without the overhead for packing and unpacking pointers [**?** **?**]. Our region allocator is based on an existing sequential implementation by Gay et al. [**?**]. We extend the sequential allocator with support for a global address space among MPI processes, without expensive remote memory accesses as in PGAS models; DRASync ensures that a region will occupy unique virtual memory addresses among all MPI processes. Then, upon transferring a region from one MPI node to another, it occupies the same virtual memory addresses. Therefore, all pointers internal to the region or to other regions that are also placed locally can be safely followed by the MPI process. In our implementation, a region is composed of:

- *Region Pages:* A linked list of contiguous memory blocks (pages), where we store the allocated data. The pages can have arbitrary size, with the default being 4KB.
- *Region Ownership:* The rank number of the MPI node that "owns" the region. The owner can be the creator process of the region or some other node where the region was transferred. The owner node has read/write permissions to the regions' data.
- *Region ID:* A unique identification number; we use its memory address in the global virtual memory space for convenience. Tying a region to its virtual memory address guarantees that the address will be available on other MPI processes and pointers will remain valid even upon transferring the region to the remote host.
- *Region Hierarchy:* Its subregions; we keep the hierarchical model of the original sequential allocator. We use a tree-based implementation and store the `parent`, `siblings`, and `children` regions.
- *Region lock:* The region's current ownership state. We mark the region as locked if a node has acquired the region's ownership and unlocked otherwise. When unlocked, the region is free to be transferred to whichever node requests it.

2.2 Allocator API

To preserve a global address space abstraction, we require all memory allocations to return a unique address. To do that, we assign the core with MPI rank 0, the *manager node*, to manage memory page allocation. The manager node serves allocation requests by the remaining nodes sequentially, serving as a monitor that preserves uniqueness of virtual addresses. The manager node allocates pages of memory to the nodes requesting them, in the order the requests are received. In all other ways, the manager node acts as any other node in the system.

New regions and subregions can be created by calling `new_region()` and `new_subregion()`, respectively. Function `new_region()` creates a region with no parent, at the root of a region hierarchy tree. Root regions have a dummy parent node to facilitate their management. Function `new_subregion()` adds a new leaf in the region hierarchy tree, as a subregion of its argument. The metadata that represent the newly created subregion are allocated within its parent's memory space.

A region is deleted by `delete_region()`. Deleting of a region results in the bulk deallocation of its data and the deallocation of all its children recursively. To delete a region, a node must own it before it calls `delete_region()`. The pages of the deleted regions are moved to free lists. We keep a free list for two groups of pages, one for the default size and another for page sizes larger than the default, which may have resulted from the allocation of large objects. The free lists are kept local and recycled within the node. As with used pages, every node in the system will have unique pages stored in its free lists.

Object memory allocation is done by `ralloc()`, which takes as an argument a region ID and the object's size. Function `ralloc()` checks whether the region contains any pages with enough space to allocate the object immediately. If not, it checks for free pages in the free list locally, recycles a free page into the region and allocates the object there. If both local searches do find space in memory, then `ralloc()` sends a message to the manager node to reserve a new memory page.

```
/* Region Management */
region new_region(void);
region new_subregion(region r);
void delete_region(region r);

/* Object Management */
void *ralloc(region r, size_t s);

/* Region Transfer */
void acquire_region(region r, int choice);
void release_region(region r);
```

Figure 1: The DRASync API.

When a system node needs data that are not local, it needs to contact the owner of the wanted region to obtain it. Using MPI, the programmer would have to pack the data structures in a portable representation, send/receive the data, and recreate the structure on the requesting node. Using the previous approach, one can avoid packing and unpacking the data, but still needs to use send/receive primitives to send regions. This means a node that requires a given region must wait until the previous owner sends it. We propose a higher level abstraction, using asynchronous acquire and release primitives. Specifically, when a node stops needing a region, and before any other node has need for the data, it calls `release_region()` to release ownership of the region. When another node requests data from that region, it acquires ownership by calling `acquire_region()` on the region ID, which will transfer the region seamlessly to the requesting node.

Locally, `release_region()` marks a region as unlocked by setting the lock field in the region descriptor as `UNLOCKED`. That marking is also passed to all the region's children. The function `acquire_region()`, is used to obtain a region. We distinguish two cases, similarly to reader-writer locks:

In the first case, a node requests a region for reading, by setting the second argument to `choice = READ`. If the owner of the region is also the requester, then the region and its

children are marked as locked by setting the lock field in the region descriptor as `LOCKED`. If the current owner is different than the requester, the requester sends a message to the owner to obtain a *copy* of the region's data. The owner sends the region and its data, recursively, to the requester.

In the second case, a node needs a region for writing, set by using the argument `choice = WRITE`. If the requester is the owner of the region then the region is locked as above. If the requester is not the owner, it sends an ownership request to the owner. Following lock semantics, if that region is marked as `LOCKED` by the owner, the requester must wait until the owner calls `release_region()`. When the region becomes available, the owner sends the region and its data, recursively, while also transferring the ownership of that region. For that purpose, the owner updates the "owner" field of the transferred region's local descriptor to point to the new owner and keeps that descriptor as a dummy, to forward any further requests to the new owner.[1]

If a node receives a request for a region it does not own, it simply forwards the message to the new owner. If the region has ever been stored locally, this will be available in the region descriptor, otherwise it forwards the request to the manager node.

3. IMPLEMENTATION

This section focuses on the communication between the system nodes. All communication is done with MPI messages, initiated by calling the library API. These MPI messages are handled by a special *server thread* used by the library, as mentioned in the previous section. Note that if the application also uses MPI to communicate outside the region library, the MPI implementation used must be thread-safe.

We categorize the dispatched request messages exchanged among server threads and the "main" application threads of different nodes into four types depending on functionality. Two of them correspond to requests and are sent only from the application thread to the server threads of remote nodes. The other two correspond to responses and are sent from the server thread to remote application threads. Every message is identified by an ID, depending on the exchange. Specifically, the types of the messages are:

Request New Page: This message type is always sent to the manager core. Along with the ID it contains the size of the new page. As the name states, this message is a request message for allocation of a new, unique page to the requesting node.

Request Region: This message type is sent from application thread of a node to the server thread of another node that owns a given region. The message has a field denoting the READ/WRITE ownership of the requester, the region ID the sender needs and the senders rank. We save the rank in the message in case of request forwarding, because there is a possibility that the receiver of the message is no longer the owner of the region. If we only use `MPI.Status` for retrieving information about the senders then the information about the first sender will be lost. Therefore, if the owner is found after several message forwards then the response goes back only to the original requester.

Transfer Page: This message type transfers a page from the server thread of a node to the application thread of a different node. It stores the total size of the page to be transferred and is immediately followed by a transfer of the page contents. This message only follows as a response after a *request region* message. Note that the message may not originate from the node that received the request, as one or more forwardings of the request may have happened due to the region moving to a different node.

Transfer Completed: This is a response message stating that all the pages of a region have been safely transferred. This message does not pass any other information, it behaves as an ACK message.

A message exchange between two *server threads* can happen only if a region request must be forwarded to another core due to the region being moved.

3.1 Synchronization Locks

We mention above that every region descriptor contains a `locked` field, which may be `LOCKED` or `UNLOCKED`. This functionality exists to implement ownership semantics over whole regions, and in this manner the region need not be transferred to another node before the owner has finished operating on the region's data. This introduces a small race window between the server thread trying to respond to a remote request and a local application thread trying to reacquire the region. We use a standard thread mutex mechanism to avoid this race; The server thread locks the region throughout its transfer to a remote node to prevent the application thread from locking it before the transaction is finished. Conversely, the application thread locks the region in order to use the data of the region, blocking the server thread from transferring it. Note that this locking is to resolve a race among the two local threads, and is different from the locking semantics we implement on regions among nodes. Simply using the same `locked` variable that implements remote locking is not possible, because it does not differentiate between the server and application threads as to who owns the region. Thus, we have split the `LOCKED` state into server thread locked `SLOCKED` and application thread locked `CLOCKED`.

3.2 Server Thread

A *server thread* of the manager node runs at all times in the background at every node, handling all incoming messages concerning the allocator. When a message requesting a new page is received, the server thread allocates a new page and transfers it to the requester. Pages can have an arbitrary size, with a minimum page size of 4KB.

When a message requesting a region arrives, the server thread checks, based on the region ID, whether this region descriptor exists in its address space. If the descriptor does not exist it forwards the message to the manager node for a global search. If the local node is not the owner but has a descriptor of the region, then the descriptor points to the next owner. Then, the server thread forwards the message to the corresponding node's server thread. Finally, if the

[1]Requests are forwarded synchronously by the server thread, so multiple requests will block until they are forwarded in order.

local node is the owner, the server thread marks the region as locked by the server, if the region was previously released, and starts to send the pages of the region to the requesting application thread. When this process is over, the server thread sends a message declaring the end of the transfer and also the new region descriptor to the receiving application thread. Finally, depending on whether the region was requested for READ or WRITE, the server thread will either take no further action, or change the owner of the dummy region descriptor that will continue to exist in its address space.

3.3 Client Thread

Each node runs an *application thread* that executes the main application. During execution of the application code, it calls the library functions for allocating, deleting and transferring functions. When the application thread calls function `acquire_region()`, it blocks at an `MPI_Recv()` function waiting for the region's pages to arrive. When the first message is received, the application thread receives the size of a page from that message and uses it to allocate and receive that page's data with a subsequent message. We use two different MPI messages for this operation because the application thread needs to know the size of a page before it receives its data so that the page can be properly placed at the correct virtual address. We do not use an `MPI_Probe()` to avoid having two messages, because the application thread at that time can either receive the message with the page size, or the message that denotes the end of the transaction and also to avoid copying. When the second message containing the data of the first page is received, the application thread receives the data directly into the corresponding memory address.

Because DRASync does not keep track of all the pages that a region has stored, we do not know when the transfer of the region and its pages will finish. Therefore, the owner sends a separate message denoting the end of the transfer. Together with this message the previous owner sends also the region descriptor to the application thread of the new owner node, to properly change ownership and lock the region.

As stated above, the manager node is responsible for allocating new pages for the other cores, to preserve global uniqueness of memory addresses. Note that to allocate memory pages for the application node on the manager core, the thread does not send a message. This occurs because the MPI implementation forbids sending messages from a node to oneself. Thus, all local allocations made by calling `ralloc()`, as far as the manager core is concerned, are not handled by the local server thread but only by the application thread. To avoid a race between the call of `ralloc()`, we use a POSIX mutex when the global memory pointer is augmented.

4. EVALUATION

We evaluate DRASync using two applications also used to evaluate the Myrmics allocator [**?**], namely Barnes-Hut N-body simulation and Delaunay triangulation. Both are MPI applications that use pointer-based data structures and have been already written to use a region-based allocator [**?**]. We modified these applications to adapt to DRASync's ownership semantics for regions, instead of using send/receive primitives and explicit communication. In short, the modification steps are:

1. Replace region creation and destruction calls with DRASync's equivalent API.
2. Replace memory allocation and deallocation calls with equivalent calls to the corresponding regions.
3. Remove `sys_send()` and `sys_recv()` and replace them with `region_release()` and `region_acquire()`. Note that `region_release()` can be used to unlock a region before the equivalent `sys_send()` in the original code, possibly making the program more intuitive, as region ownership code is together in the application with the code using the region data.

We compare the results of our MPI implementation with the Myrmics memory allocator [**?**]. We use a single system with four AMD Opteron 6272 multiprocessors, totaling 64 cores running at 2.10GHz, with a total of 256GB main memory. We use separate MPI processes, each with an application and a server thread as state above, and pin the threads to specific cores so that each process was running on two cores of a single chip. We enable `MPI_THREAD_MULTIPLE` in order to provide thread safety to POSIX threads that call MPI functions. This is a prerequisite for our library, because both application and server thread use MPI to communicate with other MPI processes. Every MPI process consists of two threads, as stated above.

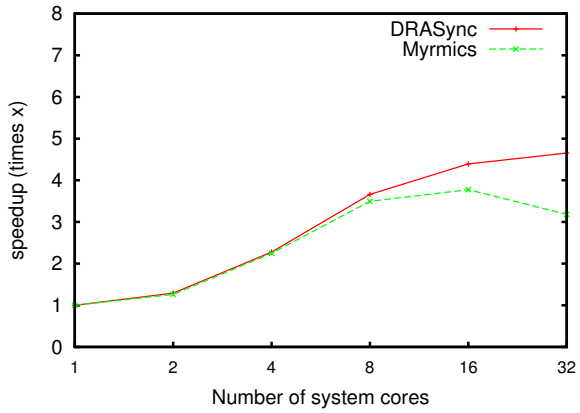
4.1 Barnes-Hut

Barnes-Hut is an N-body simulation that calculates the movement of a number of astronomical objects based on their gravitational effects by using the center of their masses. Figure 2 shows the results of running the Barnes-Hut simulation for 8KB and 82 KB of input data. Figure 2a shows the speedup attained for processing 8KB of data, for various numbers of MPI processes. Note that the number of threads is twice that of MPI processes, so the x-axis reaches 32 processes for 64 total cores. We compare with the Myrmics allocator for an equivalent number of worker processes, where the Myrmics specialized scheduler cores are not counted. We find that DRASync scales slightly better above 8 processes. Figure 2b shows a breakdown of the total running time for the manager process, showing that communication overhead does not increase as much with the number of cores as with the Myrmics allocator.

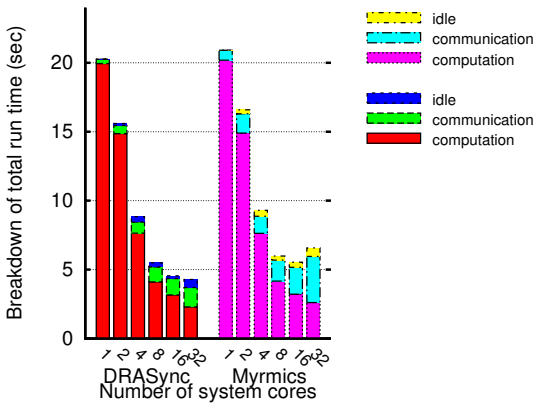
Figure 2c shows the speedup for an input set of 82KB, for various numbers of MPI processes. Again, DRASync scales better than the Myrmics allocator, mainly due to the communication overhead being less for large core counts, as shown in Figure 2d. We believe that this behavior is due to the existence of a server thread that effectively transforms the benchmark into using asynchronous communication, reducing the time waiting to receive an MPI message in the application critical path.

4.2 Delaunay

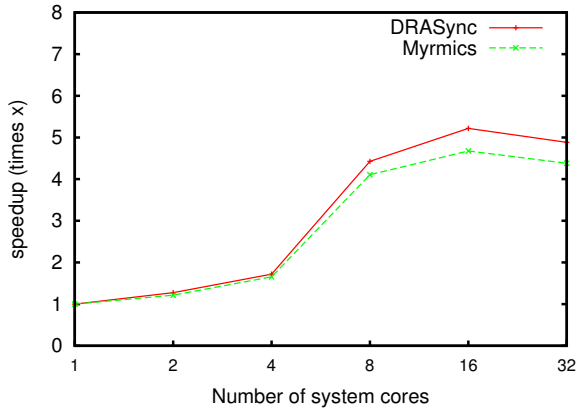
Delaunay triangulation is an algorithm that creates a set of triangles from a set of points in a 2D plane such, that every triangle connects three points. We port the implementation used in the Myrmics allocator, which is based on the serial Bowyer-Watson algorithm. Note that this implementation



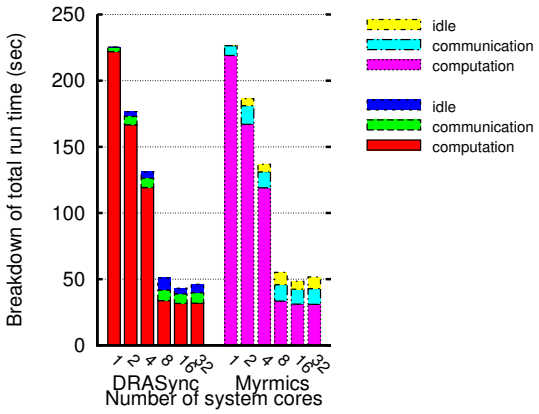
(a) Speedup graph of for 8KB data.



(b) Breakdown graph of total run time for 8KB data.



(c) Speedup graph for 82KB data.



(d) Breakdown graph of total run time for 82KB data.

Figure 2: Barnes-Hut N-body simulation with different workload sizes.

requires the number of processes to be a power of 4, thus, we measured performance up to 16 MPI processes that take 32 cores.

Figure 3 shows the results for a small dataset of 100KB and a large dataset of 1MB of data. Figure 3a shows speedup is close to linear for DRASync, while the Myrmics allocator scales to 10× for 16 cores, mainly due to idle time spent waiting for communication, as shown by the breakdown in Figure 3b.

For the large data set, Figure 3c shows that DRASync behaves very similarly to the Myrmics allocator, attaining a superlinear speedup for 16 MPI processes. Moreover, Figure 3d shows that the behavior of the two allocators is similar also for time spent in communication and idling.

5. RELATED WORK

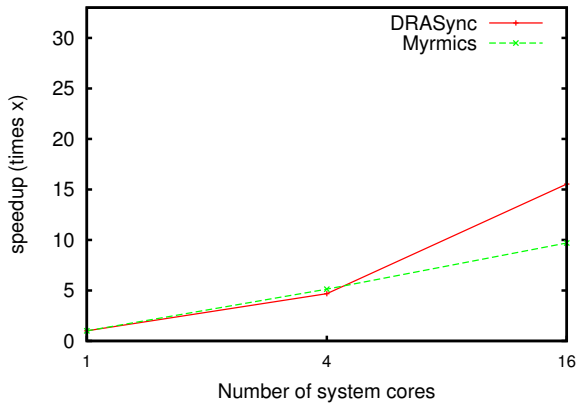
Region-based memory management [**?** **?** **?**] has been extensively studied in the past as a mechanism to control object lifetimes and fast, bulk object deallocation. Region-based memory management has been recently used in parallel programming to control the memory effects of parallel threads and processes [**?**], and restrict shared data [**?**].

The Myrmics memory allocator [**?**] is a region-based memory allocator designed for distributed or manycore machines with no cache coherence. In contrast to DRASync, Myrmics specializes several nodes into schedulers that only manage memory allocation and it uses send/receive communication to transfer regions of data, while our allocator implements an ownership semantics similar to reader-writer locks. Legion [**?**] uses logical regions to describe the organization of data. Legion uses logical regions to express locality and dynamic computations over pointer-based data structures, and transfers regions using GASNet for inter-node communication.

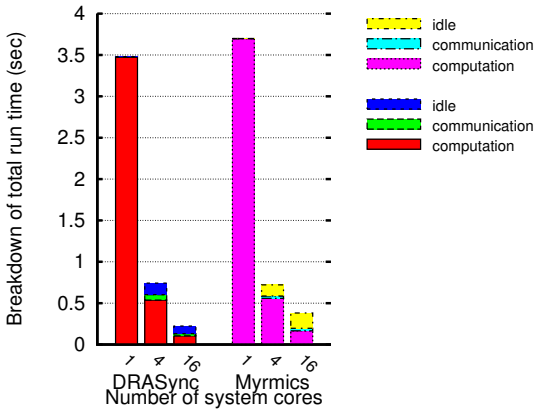
The problem of memory allocation for parallel programs has been studied extensively [**?** **?**], although most allocators focus on parallel systems with shared memory, software distributed shared memory, or statically Partitioned Global Address Space systems [**?** **?** **?**]. Distributed Transactional Memory systems [**?**] adapt software cache coherency to detect concurrent access to remote copies of data and abort or block conflicts.

6. CONCLUSIONS

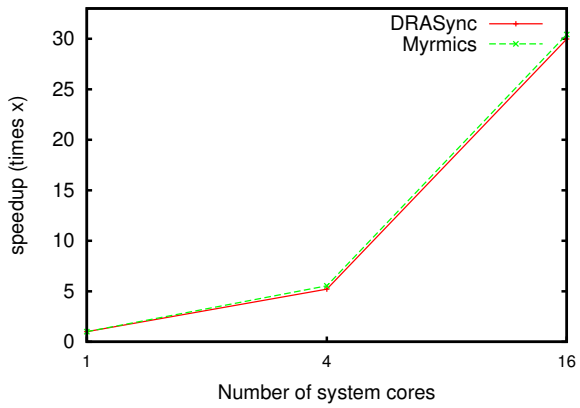
In this paper we present a region-based memory allocator for MPI programs that supports pointer-based data struc-



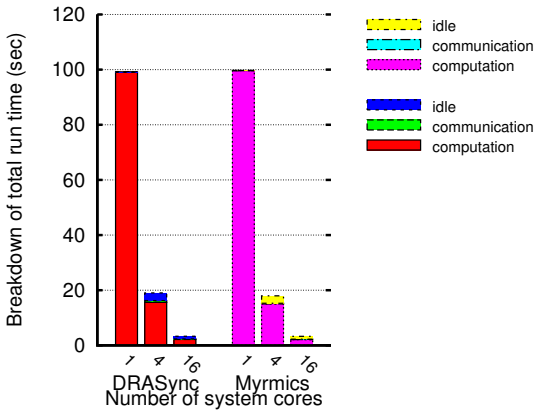
(a) Speedup graph of for 100KB data.



(b) Breakdown graph of total run time for 100KB data.



(c) Speedup graph for 1MB data.



(d) Breakdown graph of total run time for 1MB data.

Figure 3: Delaunay triangulation with different workload sizes.

tures. The allocator performs as well as the state of the art and provides an intuitive synchronization abstraction to programmers, similar to reader-writer locks.

References

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *SC*, 2012.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS*, 2000.

[3] R. Bocchino, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[4] Chapel Language Specification 0.796. `http://chapel.cray.com/spec/spec-0.796.pdf`, Oct. 2010.

[5] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI*, 1998.

[6] P. Gerakios, N. Papaspyrou, and K. Sagonas. Race-free and memory-safe multithreading: design and implementation in cyclone. In *TLDI*, 2010.

[7] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.

[8] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 20, January 1990.

[9] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *International Conference on Distributed Computing*, 2005.

[10] S. Lyberis, P. Pratikakis, D. S. Nikolopoulos, M. Schulz, T. Gamblin, and B. R. de Supinski. The myrmics memory allocator: hierarchical,message-passing allocation for global address spaces. In *ISMM*, 2012.

[11] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification v2.1. `http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf`, Feb. 2011.

[12] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM*, 2006.

[13] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2), 1997.

[14] UPC Language Specifications v1.2. `http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`, May 2005.