Distributed Region-based Allocation and Synchronization

Christi Symeonidou

Thesis submitted in partial fulfillment of the requirements for the

Masters’ of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

and

Institute of Computer Science
Foundation for Research and Technology Hellas

Thesis Advisors: Prof. Angelos Bilas, Dr. Polyvios Pratikakis

This work has been performed at the Foundation for Research and Technology-Hellas
The work is partially supported by the the European Union 7th Framework Programme
[FP7/2007-2013], under the ENCORE (grant agreement n° 248647).
Distributed Region-based Allocation and Synchronization

Thesis submitted by
Christi Symeonidou
in partial fulfillment of the requirements for the
Masters’ of Science degree in Computer Science

THESIS APPROVAL

Author:  
Christi Symeonidou

Committee approvals:

Angelos Bilas, Professor, Thesis Supervisor
Computer Science Department
University of Crete

Manolis GH Katevainis, Professor
Computer Science Department
University of Crete

Dimitrios S. Nikolopoulos, Professor
School of Electronics, Electrical Engineering and Computer Science
Queen’s University of Belfast

Departmental approval:

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, June 2013
Abstract

This thesis presents DRASync, a region-based allocator that implements a global address space abstraction for MPI programs with pointer-based data structures. The allocator amortizes communication among nodes to allow efficient parallel allocation in a global address space and takes advantage of other region-based allocation benefits, such as bulk deallocation and good locality even in pointer-based data structures. Moreover, we implement 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 our allocator on two application-like benchmarks and compare the results against a state-of-the-art distributed allocator. We find it is as efficient, while offering a higher level abstraction to programmers.
Περίληψη

Στην εργασία αυτή παρουσιάζουμε τον DRASync, έναν δεσμευτή μνήμης βασισμένο σε περιοχές, ο οποίος υλοποιεί ένα ‘αφηρημένο’, ενοποιημένο συστήματα διευθύνσεων για MPI προγράμματα τα οποία κάνουν εκτενή χρήση δομών δεδομένων με πολλούς δείκτες, όπως π.χ. γράφοι ή δέντρα. Ο δεσμευτής, αυτός, αποσβένει την επικοινωνία μεταξύ των κόμβων επιτρέποντας μια αποδοτική, παράλληλη δέσμευση μνήμης στον καθολικό χώρο διευθύνσεων. Ταυτόχρονα, αξιοποιεί τα οφέλη που προσφέρουν οι δεσμευτές που βασίζονται σε περιοχές, όπως τη μαζική απελευθέρωση μνήμης και τη χαλάτ οπτικότητα, ακόμα και σε δομές δεδομένων με πολλούς δείκτες. Επιπλέον, υλοποιούμε μια σημασιολογία ιδιοκτησίας των περιοχών από τους κόμβους του συστήματος, μετά τις τεχνικές reader-writer locks. Ο τρόπος αυτός δημιουργεί ένα διαστημικό εργαλείο συγχονισμού υψηλού επιπέδου στα MPI προγράμματα, χωρίς όμως να θυσιάζεται η απόδοση της διανομής μηνυμάτων. Αξιολογούμε τον DRASync με δύο εφαρμογές και συγκρίνουμε τα αποτελέσματα με έναν κατανεμημένο δεσμευτή μνήμης τελευταίας τεχνολογίας. Βλέπουμε ότι ο DRASync είναι εξίσου αποτελεσματικός, ενώ παράλληλα προσφέρει ένα υψηλό επιπέδο abstraction στους προγραμματιστές.
Acknowledgements

I would like to express my sincere gratitude to my supervisors and academic advisors, professor Angelos Bilas, professor Dimitris S. Nikolopoulos and Dr. Polyvios Pratikakis. Their assistance and guidance throughout my academic studies were truly valuable and helped me to grasp a better and deeper understanding of the field of Parallel and Distributed Systems.

I would also like to thank all my colleagues in CARV laboratory of ICS-FORTH that made studying and programming fun and interesting. I am grateful for them being able to help me when they were most needed. I feel also grateful towards ICS-FORTH, for providing me a student scholarship and an environment along with the necessary equipment to complete my research.

Finally, I would like to thank my family, all my friends and Panagiotis for their support and care during the time of my studies, with any way they could.
Contents

1 Introduction .............................................. 3
  1.1 Contributions ........................................ 3
    1.1.1 Distributed Shared Memory ....................... 4
    1.1.2 Region-Based Memory Management .............. 6
  1.2 Background and Related Work ..................... 7
    1.2.1 Virtual Memory .................................... 7
    1.2.2 General Purpose Memory Allocators ........... 8
    1.2.3 Region-Based Memory Management .............. 9
    1.2.4 Memory Management Strategies .................. 11

2 Design ................................................. 15
  2.1 Regions ............................................... 15
    2.1.1 Region Description ............................... 16
  2.2 Allocator API ....................................... 18
    2.2.1 Region Creation and Deletion .................... 18
    2.2.2 Object Creation and Deletion ................... 19
    2.2.3 Region Transfer ................................... 20

3 Implementation ......................................... 23
  3.1 Messages ............................................. 23
  3.2 Threads .............................................. 24
    3.2.1 Locking the Region ............................... 24
    3.2.2 Server Thread .................................... 25
    3.2.3 Client Thread .................................... 25

4 Evaluation .............................................. 29
  4.1 Delaunay ............................................. 30
    4.1.1 Measurement Methodology ....................... 31
    4.1.2 Results ........................................... 31
  4.2 Barnes-Hut .......................................... 33
    4.2.1 Measurement Methodology ....................... 33
    4.2.2 Results ........................................... 34
5 Conclusions and Future Work

5.1 Future Work ................................................................. 39
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Distributed Shared Memory View</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Region Hierarchy</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Allocator API</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Simplified acquire_region()</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Example of a region transfer</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>State machine for creating a new region</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>State machine for acquiring a region for reading purposes</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>State machine for acquiring a region for writing purposes</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Graphs for the Delaunay Triangulation with different sizes of data</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Graphs for the Barnes-Hut N-body simulation with different work-load sizes</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Graphs for the Barnes-Hut N-body simulation with different work-load sizes</td>
<td>37</td>
</tr>
</tbody>
</table>
List of Tables

4.1  Speedup of the Delaunay triangulation. . . . . . . . . . . . . . . . 31
4.2  Speedup of the Barnes-Hut N-body simulation. . . . . . . . . . . . 35
Chapter 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 [1, 2, 3, 4], 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 marshaling and unmarshaling of pointers for communication. Regions allow intuitive representation of pointer-based data structures that is easy to send and receive [5, 6] 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.

Our work 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 rendezvous synchronization. This helps parallel programmers reason about ownership of data at a high level, 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.

1.1 Contributions

We make the following contributions:

- We implement a region-based distributed allocation library for MPI programs by providing DSM-like functionality. Our aim is to provide a portable library
that can operate on top of multiple machines. MPI gives an essential virtual
topology, synchronization, and communication functionality between a set of
processes that have been mapped to nodes, servers or computer instances.
Also, the MPI is designed in a language-independent way. These features
enhance the portability of our allocator in the different nodes participating
in a cluster.

- 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. The data structures’ pointers are now meaningful to
all nodes of the system, since each pointer refers to an address of the global
space and every node can refer to that address without it having to perform
any memory address translation

- 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.

1.1.1 Distributed Shared Memory

Distributed Shared Memory (DSM) is a memory abstraction where a set of inde-
pendent nodes, connected by a general interconnection network and communicating
through a message passing interface, seemingly share a global address space. This
global address space is actually a virtual memory logically partitioned into the
separate physical memory address spaces of each node participating in the sys-
tem. DSM hides the underlying communication functionality from the application
programmer, since it provides only simple load/store calls simplifying the program-
ing of applications and giving the programmer the ability to manage the data
distribution [7].

In our study, we implement a software DSM-like memory model as a runtime
library on top of a message passing interface, which is linked with an application
program that uses a shared virtual address space. We base our virtual address
space design on the implementation proposed by [5]. This virtual address space
is a part of the virtual memory of the system. This virtual part starts at the
same memory address in each system node, but every virtual address maps to the
separate physical address of the node. We view the shared memory as a linear array
of bytes which is virtually shared by every node participating in the cluster. We
define in our implementation the size of memory space that is exchanged between
nodes to be the same as the most common memory page size of 4KB.

We design our DSM model with a central manager node responsible only for
allocating space in the shared address space and then transferring it to the cor-
responding node. The central allocator removes the need for any synchronization
1.1. CONTRIBUTIONS

Figure 1.1: The DSM model is similar to the shared memory paradigm. The left box represents the virtual memory address space all the nodes have access to. Any thread/node in the cluster may reference the virtual shared space, for example to the address of the variable x. This virtual address space though, is partitioned into the actual physical memory of the nodes. Only local threads may reference the private physical memory and every memory has affinity to its corresponding thread. Even though the variable x may be referenced from all nodes, it is actually stored in the first physical memory.

points by executing all the incoming requests for space allocation sequentially. Otherwise, the model is dynamically distributed since all the nodes communicate with each other in order to transfer data already allocated in separate memory spaces. Every memory space is called a region or arena or zone \cite{8, 9, 10}. This distribution is achieved with the use of node ownership over the regions. Every node is responsible for the memory space that it owns, making it responsible for transferring it to whichever system node requests reading or writing permissions over data residing on those memory spaces.

The system provides a relaxed memory consistency model with a single writer and multiple readers. The writer node is the region’s owner node. All the other nodes are possible readers. If a node needs to read the data residing in a region owned by another node then it sends a message to the owner requesting the replica-
CHAPTER 1. INTRODUCTION

tion of the region’s data. After the replication that region’s owner is not responsible for updating the data unless it receives another request message. The owner and hence the writer of a region can change. If a system node wants to obtain a region and become its owner in order to have writing permissions over it, it has to send a message to the owner requesting the ownership. If the owner node stops using the region, it transfers it to the requesting node along with the ownership. This approach minimizes the overhead of keeping up-to-date replicas of the original data and also the overhead of replicating all the data to all the nodes participating in the system. Since the read operation is the most used pattern in parallel applications this approach reduces the message exchange between system nodes.

We have implemented an acquire/release model that divides synchronization accesses to acquire and release functions. Acquire is responsible for requesting a region from an owner node and after its transmission, is responsible for locking that region until a release call for the same region. Release is responsible for unlocking the region, making it available for acquire from other nodes. All reads and writes can occur on a region only after the acquire call regarding that region has completed on the same node and before the release of that region. The acquire and release synchronization accesses must hold node consistency and the programmer must use acquire and release as synchronization primitives along with barrier synchronization to ensure the consistency of the data.

1.1.2 Region-Based Memory Management

Region-Based Memory Management is a type of memory management where all objects are allocated inside regions. A region [10], is a collection of blocks of memory, inside which reside allocated objects that can be massively deallocated at once. Regions are a good abstraction for describing locality of accesses since all objects in a region are allocated in a page, namely a contiguous memory space.

As we have previously stated, we have introduced regions in our DSM model. Instead of transferring single objects from one node to another when necessary, we transfer a region that contains several objects. In the traditional DSM model for every object we would have a single read request and a single answer. For a collection of such objects there would be as many read requests and answers as the number of the objects needed to be transferred. With the use of regions, and thus the gathering of the objects in contiguous memory spaces, read requests and answers are reduced to the number of the regions.

For example, let us consider a graph, a data structure that needs to store to the traditional DSM numerous different objects, and the number of transfers required for each of these different objects. We expect to have as many transfers as the objects the graph contains. If that graph was to be stored in a single region, the transfers would be diminished to the number of the contiguous memory spaces the region contains. These transfers are considerably less than the number of the transfers for each object. And if we consider that each transfer consists of a request message and a response message, the total number of messages is much larger in
the traditional DSM than the region-based DSM.

1.2 Background and Related Work

Many different memory management techniques have been introduced and used in order to provide dynamically memory space. Each architecture specification has different requirements that need to be covered by the memory management, in order to increase its effectiveness. The initial designs of memory management were simple in order to satisfy the needs of a single application. But as system architecture complexity increases, the more the memory management techniques have become more specific. Some of the techniques of dynamic memory management that have been proposed and used effectively are:

- **Single contiguous allocation.** It is the simplest memory management technique where a single application has access to most of the computer’s memory. In order to support multitasking, the system must swap the memory contents of a process in order to make the whole memory available for another process. This technique is also practical for embedded systems that run a single application. A system that used this approach is MS-DOS.

- **Partitioned allocation.** The primary memory is divided into multiple memory partitions, contiguous areas of memory. Each partition might contain all the information for a specific task. The allocation of a partition is performed when a task starts and the deallocation when the task ends. Partitions may be either static or dynamic.

- **Virtual Memory or Paged memory management.** This management divides the primary memory into fixed-size units called pages and the application’s address space into pages of the same size. The physical memory can be allocated on a page basis while the address space appears contiguous.

- **Segmented memory management.** This memory management technique divides the memory into segments that correspond to a group of information. A segment is created for programs or for different classes of memory usage. Some segments can be shared between programs. Addresses in such a segmented system consist of the segment id and an offset relative to the segment base address, defined to be offset zero.

Since our work is based on memory virtualization, we focus our background research on this memory management technique and we will describe it in the following sections.

1.2.1 Virtual Memory

The concept of virtual memory was the inspiration for many dynamic memory management implementations. One of them is Memory Pool [11] that uses free lists of fixed-size blocks of memory, called memory pools. The compiler is responsible to allocate these pools during compile, so that an application during runtime to able to allocate and deallocate memory blocks from those pools. Regions are very
common with memory pools, but their difference is that pooling does not provide bulk deallocation of the memory spaces.

Another implementation of dynamic memory management is Buddy Blocks [12]. In Buddy Blocks, the memory is divided into blocks with size of a power of two. All memory blocks are preserved in a data structure, such as a tree or a sorted linked list, based on their size. When an allocation occurs, a new block is created and is added to its respective data structure. When an object asks for memory space, a block is selected and is halved until the size of the block becomes the smallest size available to hold the object. These halves are called buddies as the name of the implementation states.

Virtual memory also was the basis for the more generalized technology of memory virtualization. Memory virtualization provides a more flexible and abstract relationship between many, different physical resources. While virtual memory provides an abstract relationship between an application and the physical memory of the system that runs the application, the memory virtualization technology enables the abstraction of the physical memories of a whole cluster.

1.2.2 General Purpose Memory Allocators

A very common approach of memory management is the dynamic memory allocation performed by the C standard library. As its name states, the memory management is written in the C programming language and the system using this memory management calls functions `malloc`, `realloc`, `calloc` and `free` residing in the C standard library. The C dynamic memory allocator binds memory space dynamically from the heap and returns a pointer to the application in order for it to use that memory block. When the block is no longer needed, the application calls the `free` function, with that pointer as a parameter, in order to deallocate the space. That block can be later used for other purposes. An issue from which general-purpose memory allocators suffer is the fragmentation of the free memory areas created in the heap.

The implementations of the C memory management must focus on two factors; the performance of the allocator and fragmentation. The implementation must provide fast allocation and deallocation functions in order to not add excessive overhead to the run time of the application since memory allocation is one of the most important and common functionalities of an application. Memory fragmentation is the result of the multiple allocations of objects with different sizes.

The allocators focus on providing solutions of this issue in order to provide a better division of memory spaces. Fast allocation and defragmentation affect one another. For example an implementation might give weight to defragmentation and thus slowing down deallocation.
1.2. BACKGROUND AND RELATED WORK

Doug Lea’s Malloc

Doug Lea’s Malloc is a general-purpose allocator developed in 1987 and is the basis of the allocator used by GNU C library. The memory on the heap is allocated as "chunks". Each chunk is bounded by tags that preserve information about the chunk, such as size and flags. A chunk can be either free or in-use. Unallocated chunks that have the same border, can be coalesced into one larger chunk, which minimizes the number of unusable small chunks [13].

The available unallocated chunks are grouped into bins based on their size. There are 128 fixed-width bins, approximately logarithmically spaced in size. Bins for object sizes below 512 bytes are placed in one bin spaced 8 bytes apart, and are allocated with a two power best fit algorithm. If there are no free blocks left in a bin, a block from the next highest bin is split in two. The chunks inside a bin are sorted by size, with ties broken by an oldest-first rule [13].

If a request cannot be satisfied, the malloc function tries to increase the size of the heap by calling sysbrk. At the end of the allocated heap resides a "wilderness" chunk that denoted the end of the expiration of the available memory [13].

Hoard: A Scalable Memory Allocator for Multithreaded Applications

The Hoard memory allocator is an allocator that aims for scalable performance by avoiding synchronization costs and false sharing. Hoard uses mmap for allocating memory from the heap. The heap is logically divided into a global heap and local heaps, one for each thread. Each thread can only access its heap and the global heap [14].

Hoard manages the memory in chunks, named superblocks, all of which have the same size. A superblock contains an array of a number of allocated objects, small blocks, and a free list of available blocks. If a block to be stored has a size larger than half the size of a superblock, then it is allocated via mmap directly. Each superblock maintains blocks of a specific size class. If a superblock becomes empty, the system moves this superblock to the global heap so it can be reused for any size class. Using these methods, Hoard allocator keeps both internal and external fragmentation at a low level [14].

1.2.3 Region-Based Memory Management

Region-based memory allocation is an alternative approach to memory management. The basic concept of regions was first introduced by Douglas T. Ross [8] in 1967. The system’s memory was partitioned into zones structured in a hierarchy. Each zone had its own allocator and all the zones could be freed at once.

In 1990, Hanson presented in [9] a region (he called them arenas) implementation in ANSI C. He demonstrated an efficient algorithm based on object lifetimes, that is very efficient if there are a few object lifetimes. The space for all objects with the same lifetime is allocated from a list of large arenas, and the entire list
Hanson demonstrated that his arenas could achieve almost half the cost of quick fit and less than twice the cost of stack allocation.

Later work [15, 10, 16] investigated the use of regions for safe memory allocation by inserting the concept of regions in compile time, so that the allocation and deallocation of regions together with a statical analysis of regions can be used in averting the creation of dangling pointers and memory leaks.

The first region-based implementations were programmed using the C programming language [8, 9], but since they did not provide safe memory management, many C related implementations have been proposed, such as Cyclone [15] and an extension to C called RC [10].

Cyclone is a dialect of the C programming language that provides the C level of abstraction needed for low-level systems and is also safe, thus avoiding drawbacks such as incorrect type casts, dangling-point dereferences, space leaks and buffer overflows. In order to provide type safety, Cyclone uses a region-based framework where each object resides in a region and a region can be deallocated at once. Region analysis provides a way to avoid dangling pointers by assigning a segment of code, a block, to a region. All the local variables of a scope are allocated inside a region and at the end of the scope the region is deallocated. Cyclone also maintains a global region heap, which is not deallocated, where static data objects are created via malloc, which is garbage collected [15].

RC is an extension to C, implemented not only to use explicitly-managed regions, but also to guarantee safe memory operations by keeping a reference counter to each regions in order to ensure that no region is freed prematurely. Regions decrease the overhead of reference counting, since references internal to regions don’t require counts to be updated when they’re modified. RC includes an explicit static type system for regions that allows some reference count updates to be eliminated [10].

Regions have been generalized to other language environments as well, such as Real time Java [16], or web based applications written in PHP [17]. In more detail: Regions were introduced in Real time Java as a component of memory management which combines them with ownership types in order to enforce object encapsulation and to ensure program correctness while eliminating runtime checks for dangling pointers on region deallocation. A framework combining a compile-time static analysis, a runtime region allocation policy, and programmer hints are used in order to ensure that a region will not create dangling references. Also prevents threads in multi-threaded programs to not access references to objects allocated in the garbage-collected heap and allows them to share objects without using the heap and cause memory leaks. Regions are a good fit for real-time computing because their overhead can be predicted statically [16].

In Web-based workloads, the use of a region-based memory management contrives to their performance since most of the memory objects are used only during one transaction. Thus all objects with the same lifetime are stored in a region and can be easily deallocated at the end of the scope with a free all. The use of regions also reduces the cost of defragmentation activities and removes the need of
maintaining a per-object header for every object, two operations that add overhead to the system [17].

Apart from providing program correctness by reducing dangling pointers, regions can also be used for task dependence analysis in order for the system to achieve good performance by exploiting the locality and the independent execution of tasks. **Legion**, a parallel programming system, uses logical regions to describe the organization of data [6]. A logical region holds a set of objects, and can be dynamically allocated, deleted and stored in data structures. In Legion the logical regions are used to provide locality by forming a relationship between a task and the data of the region. Also logical regions can be partitioned into subregions to determine independent tasks.

The **Myrmics** memory allocator is a region-based memory allocator designed for global address spaces. This allocator is composed of multiple allocator instances that implement the global address space with region support on distributed machines. The Myrmics allocator is implemented as a library and uses regions since they have proven to increase locality and to accelerate bulk allocation and deallocation. This parallel design improves the productivity with the use of dynamic regions in a global address space, which provide a convenient shared memory abstraction for dynamic and irregular data structures. Also the performance is improved through scaling on many-core systems without system-wide cache coherency [5].

### 1.2.4 Memory Management Strategies

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. Another memory management system is Distributed Transactional Memory that adapt software cache coherency to detect concurrent access to remote copies of data and abort or block conflicts.

**Shared Memory Management Strategies**

**Streamflow** [18] is a multithreaded memory manager able to provide simultaneous allocation in shared memory systems, while preserving low overhead and thread safety, and transparently favoring locality at several levels of the memory hierarchy. Streamflow decouples remote and local free lists and uses a non-blocking remote object deallocation mechanism in order to eliminate allocation synchronization and minimize deallocation synchronization. Streamflow also improves cache, TLB, and page-level locality via careful layout of heaps in memory, careful reuse of freed objects and the exploitation of superpages, make it an attractive unified framework for sequential and parallel memory allocation [18].
CHAPTER 1. INTRODUCTION

PGAS languages

Partitioned Global Address Space (PGAS) is a parallel programming model that assumes a global memory address space logically partitioned with a portion of it being local to each process or thread, while the other portion being shared to all processes or threads. Each thread, thus, has a private space and affinity with a portion of the shared address space. The most popular PGAS languages are UPC, X10, Chapel, and Co-array Fortran.

**Unified Parallel C (UPC)** [1] is a parallel extension of the C programming language designed for multiprocessors with a global address space or distributed memory. UPC provides high-performance by minimizing the overhead created during thread communication. UPC provides the programmer with a direct and easy mapping from the language to low-level machine instructions. At the same time, UPC’s parallel features can be mapped onto existing message-passing software or onto physically shared memory to make its programs portable from one parallel architecture to another.

**Chapel** [2] is an opensource parallel programming language, designed around a multithreaded execution model in which parallelism is described in terms of independent computations implemented using threads. Chapel provides higher-level abstractions for parallelism using anonymous threads that are implemented by the compiler and runtime system. Chapel supports abstractions for data parallelism, task parallelism, and nested parallelism. It also allows the programmer to control where data is stored and where computation takes place on the machine, and to allow this information to be added to a program incrementally. Moreover, it allows for code reuse and generality through object-oriented concepts and generic programming features.

**X10** is a parallel object-oriented programming language developed by IBM [3]. It is designed for high performance parallel programming and high productivity in the presence of non-uniform data accesses. X10 balances four major goals; safety, analyzability, scalability, and flexibility. X10 provides safety by ruling out by design errors common in HPC applications. This language is also designed to be conceptually simple in order to be both attractive as a language and also to be analyzable by programs such as profilers, compilers and static analysis tools. X10 highlights the explicit reification of locality in the form of places; the use of lock-free synchronization (atomic blocks); and the manipulation of cluster-wide global data structures.

**Co-Array Fortran** is an extension of Fortran 95 for parallel processing [4]. A Co-Array Fortran program is interpreted as if it were replicated a number of times and all copies were executed asynchronously. Each copy has its own set of data objects and is termed an image. The array syntax of Fortran 95 is extended with additional trailing subscripts in square brackets to give a clear and straightforward representation of any access to data that is spread across images. References without square brackets are to local data, so code that can run independently is uncluttered. Only where there are square brackets, or where there is a procedure
call and the procedure contains square brackets, is communication between images involved. There are intrinsic procedures to synchronize images, return the number of images, and return the index of the current image [4].

**Transactional Memories**

Transactional Memory is a concurrent programming API in which concurrent threads synchronize via transactions (instead of locks). A transaction is an explicitly delimited sequence of steps to be executed atomically by a single thread. A transaction can either commit or abort. If a transaction aborts, it is typically retried until it commits. Support for the transactional memory model on multiprocessors has recently been the focus of several research efforts, both in hardware and in software [19].
Chapter 2

Design

We have implemented a distributed region-based allocator as a runtime library with MPI as a back-end for the communication between nodes. We provide an API for creating and deleting regions, allocating memory for objects that reside in each of these regions and also for transferring these regions across nodes in the system. The message exchange for the communication is handled by MPI function calls since our system is designed for clusters. By using our API, all the MPI function calls are hidden from the user in order to provide a simpler and more user friendly functionality for the programmer.

Our system is designed as a runtime library where the allocation of memory and the transfer of regions are scheduled and executed at run time. In order for the MPI messages not to be entangled with the application code that runs at the same time, every node in the system uses a POSIX thread to serve as a message handler and as a scheduler.

2.1 Regions

Regions are a collection of large memory chunks that contain a list of memory pages within which a number of objects or other subregions can be stored. The regions can be easily allocated and deallocated and are ideal for data that share the same lifetime. In our distributed system the use of regions enables us to transfer large amounts of data from one node to another without the overhead for packing and unpacking pointers.

From the user’s point of view a region is a memory space where objects can be allocated and be massively deallocated. In a distributed system regions also provide a global address-space abstraction to the user, since the actual location of a region is not revealed, mainly because the regions shift from one machine to another, a transition the user does not have to be informed about.
2.1.1 Region Description

The region structure is composed of a region ID, an owner, a list of memory pages with the data, some pointers for constructing a hierarchical tree used for maintaining subregions, and finally a locking variable used for protecting the region from untimely transfers. The attributes of the regions are described in more detail in the following subsections.

In a distributed system where the regions’ data must be transferred across nodes, all the information concerning the regions are preserved in a region descriptor which is also transferred, stored locally in every node and updated after specific events, in order to safekeep the system’s data.

Region Pages

Every region in our implementation contains a set of pages. A page is a contiguous block of memory where the data that are allocated within the region are stored. Each page may have variable size. The minimum default page size is set to 4KB but when a node requests for a bigger amount of data than the default page size, the page size changes to the requested size. The pages are stored as a linked list, where the region always points to the latest obtained page.

Region Ownership

The regions in our implementation retain a variable in their descriptor that denotes its owner. Every node in the system initially obtains from the MPI implementation a rank number that is unique in the system. With this number, every system node marks the regions that it owns by storing it in the ownership field. Every region owner is able to modify the data of its region and always keeps the most updated copy of these data. Whenever another node needs a region that is not in its possession in order to modify the data (read and write), has to obtain the region from the current owner by requesting a region transfer. The owner then sends the region with all the corresponding data and updates the descriptor with the new owner. A node needs a region for the sole purpose of reading the data, then the procedure is the same apart from the change of ownership.

Region ID

Every region has a unique identification number. For this number we use the memory address where the region is stored in the virtual memory. This modification provides us many benefits. First of all, every node in the system behaves as a separate process that runs the same application, so the addresses of the virtual memory will be the same for every process but the data will be always different. Because of a centralized method of memory allocation, a region is created in a block of virtual memory and is handed to a single node, which owns it. This procedure provides the certainty that the address of the region posing as an ID is unique.
2.1. REGIONS

Figure 2.1: A simple example of the region hierarchy. Every node has as the root of the region hierarchy tree the permanent region. In the figure, the blue boxes are regions that are created by calling the new_region() and the yellow boxes are created by calling the function new_subregion(). Every region has a pointer to the first child and then every child points to the next child in the children list.

in the set of nodes. In addition, this choice simplifies the copy or transfer of the region and its data since the source memory address is the same as the destination memory address, which also makes transferring of pointers meaningful.

Region Hierarchy

We preserve regions in a hierarchy, which provides us the possibility to create both regions and subregions of every region. Every region can be a subregion of another region and at the same time it can have multiple other subregions. This demarcation is implemented by keeping pointers in the region descriptor which hold a specific property. We use a tree based implementation for preserving this hierarchy, the pointers we use are parent, sibling and children. The hierarchical model of the regions was first introduced by D. Gay [10] and we chose to use it since the management of the regions becomes simpler.

The parent field of the region descriptor denotes that a region was created as a subregion of the parent region and thus in that field is saved the region ID of the parent region. The children field of the descriptor, denotes that the referred region has several subregions stored. A region may have only one parent region and several children regions. The children regions are saved as a linked list, where the children field in the region descriptor stores the most recently created region. The sibling field maintains the list of the referred region’s parent’s children and always points to the next region. A simple example of the regions’ hierarchy is displayed in the Figure 2.1.
Region locked

The region descriptor has one other field named `locked`, that is needed for locking purposes. We mark the region as locked if a node still has not finished using the data stored in that region. When the node ceases to need the region, he marks it as unlock and then the region is free to be transferred to whichever node requests it.

2.2 Allocator API

The memory space where the regions are allocated is a virtual address space shared between the nodes of the system. This address space is partitioned at runtime, where every partition is a region. This memory statically allocated during the initialization and behaves as a very large array.

The partition of the address is logical and is maintained by the node that obtained the first number from the MPI. This node has been given the rank No 0 and may be referred to as manager node throughout the rest of the study. In this case we have a centralized design with blocking sends and receives in order for the manager node to serve the incoming requests for memory allocation in a serialized fashion. The serialization of the incoming requests provides us with the assurance that every request will be served in the order it was received, and that every memory chunk will be transferred to no other than the requester. It is important to note that this functionality is the only addition that the manager node has over the other nodes of the system.

Figure 2.2 illustrates the functions that compose API of the allocator library.

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

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

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

Figure 2.2: The allocator API.

2.2.1 Region Creation and Deletion

New regions and subregions can be created by calling the functions `new_region()` and `new_subregion()` respectively. Function `new_region()` adds a region that
poses as the root of the region hierarchy tree, and logically has no parent. In reality though, we point as parent of these regions a region called permanent, mainly for safekeeping and managing the region descriptors that have already been created. Function new_subregion() adds a new leaf in the region hierarchy tree. We give as an argument the parent of the new region, within which the new region is to be created. After the creation, the new region will exist as part of the data of its parent region.

Function ralloc() allocates internally all the newly created regions. This function returns a memory address from a page of the newly created region’s parents. Thus, the new regions descriptors are stored in their parents address space. In case of new_region() where the regions are created without a parent, the permanent region undertakes the role of maintaining the descriptors of these regions on one of its pages.

The permanent region exists only for the purpose of preserving region descriptors. Any other use of the permanent region such as transferring it toward any other node, the allocation of objects on the permanent’s region pages and its deletion is prohibited.

A region is deleted by calling the function delete_region(). The deleting of a region results in the bulk deallocation of its data and the deallocation of all its children recursively. For a region to be deleted though, some criteria must be met. The region marked for deletion must be owned by the node which calls delete_region(). If not, the result has no utter meaning, since all the data to be freed reside in another node’s local memory. Since the permanent region keeps all the region descriptors of the runtime it simplifies the mass deallocation of all the regions that were allocated.

When a region is deleted, all of its data stored in the pages are discarded and the page is moved to a free list. We have categorized the free lists based on the size of the pages. As we mentioned before, the pages are categorized into two groups, thus we maintain a free list for the pages with the default size and a free list for pages with size larger than the default. The free lists are local, which means that every node in the system has different pages stored in the free lists.

2.2.2 Object Creation and Deletion

For object memory allocation into a region we call ralloc(). This function takes as an argument a region ID, which is the region that will preserve the object and the object’s size. Then, ralloc() will check if the region contains any pages. If not it will check for free pages in the free list locally. If any of these cases provides ralloc() with free space, then it will return a memory space for the object. If the above local searches do not result in finding space in memory, then ralloc() sends a message to the manager node in order to request a new page. When the manager responds with a page, ralloc() returns a new memory address pointer to the caller.
2.2.3 Region Transfer

There are occasions where a system node during the execution of an application needs data that are contained in regions that are not in his possession. During these cases he needs to contact the owner of the region in order to obtain it. In most implementations when such a need arises, the programmer has to add a receive to the application running on the requester node and a send to the owner. These have to be at correct order and at specific places at the application code. The difference in our implementation is that we do not have pairs of send and receive, but acquire and release. When a node has no need of a region anymore he simply calls release_region() to release it and when an other node needs data of that region he calls acquire_region(). Figure 2.3 presents a simplified version of the body of acquire_region().

```c
1 function acquire_region(region r, int choice) {
2    if (r.owner != core_id) {
3        msg = request_region_msg(r.id, core_id, choice);
4        MPI_Send(msg, r.owner, ALLOCATOR_TAG);
5
6        MPI_Recv(msg, MPI_ANY_SOURCE, TRANSFER_TAG);
7        while (msg.type != TRANSFER_ACK) {
8            MPI_Recv(data, msg.size, status.MPI_SOURCE, TRANSFER_TAG);
9            unbundle_data(cur_page, data, size);
10            MPI_Recv(msg, MPI_ANY_SOURCE, TRANSFER_TAG);
11        }
12
13        MPI_Recv(r, msg.sender, TRANSFER_TAG);
14
15        if (choice == WRITE) {
16            r.owner = core_id;
17            link_region(r.id, permanent);
18            clientlock(r);
19        }
20    }
21 }
```

Figure 2.3: Simplified code for the function acquire_region().

The node that ceases to need a region, calls release_region() and gives the region ID as an argument. That function simply marks that 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(), as states its name, is used to obtain a region. We distinguish two cases. The first case is when a node requests a region exclusively for reading purposes, thus setting the second argument 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 owner is different than the requester, the requester sends a message to the owner
2.2. **ALLOCATOR API**

in order to receive the region’s data. The owner sends the region, its data and its children but only for a specific scope.

The second case occurs when a node needs a region for writing purposes. The requester calls `acquire_region()` with the argument `choice = WRITE`. If the requester is also the owner of the region then he has the previous behavior. If the requester is not the owner then again he sends a request to the owner. 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, its data and the subtree of children with the region as root, including their data. All these regions change ownership. The owner updates the owner field of the transferred region’s descriptor with the new owner and keeps that descriptor as a dummy in case he receives a false region request.

If a node receives a request for region transfer without being the owner, he simply forwards the message to the new owner, if he was a previous owner or he forwards the message to the manager node, if that region has never appeared in his memory space. In Figure 2.3 we present a simplified version of `acquire_region()` where in the MPI functions `MPI_Send()` and `MPI_Recv()` we have pointed only the three more important attributes, the buffer, the receiver and the tag.

A simple example of region transfers can be seen in Figure 2.4. Initially, as shown in Figure 2.4a, the node with rank No 2 has created two regions by calling the function `new_region()`, region 1 and region 2, and also three subregions of region 1 by calling the function `new_subregion()`, child 1, child 2 and child 3. Another node, ranked with No 4, has created one region, region 3 which also has a child, child 4.

Assume that the node with rank No 4 requests for the region 1 from the node with rank No 2. After the request message is received from node No 2 and if the region is released, its sends back all the data of the region. Figure 2.4b shows the regions of both nodes during the region transfer. As stated above in the study, both the region and its children are sent to node No 4. Also the regions to be transferred
are illustrated with a red rounding box to show that the region is locked and no further operation can be performed from the node No 2. After the transfer, and as shown in Figure 2.4c, the regions that were owned previously from node No 2 and have been sent to node No 4, are being kept as dummy nodes, thus in Figure 2.4c are painted gray.
Chapter 3

Implementation

In this chapter we are going to describe the communication between the separate entities of our system. The system can either be a cluster of machines or simple cores in one machine that do not have a shared memory. This communication is achieved by MPI messages that are masked under the functions of our library interface. These MPI messages are being handled by a POSIX thread, as we have already mentioned in previous sections. We named this POSIX thread server thread to distinguish it from the thread that runs the application, which we named it client thread.

3.1 Messages

The communication between the threads of two separate nodes is by sending messages. We have created our own structs in order to categorize the dispatched requests. Every message is identified by an ID depending on the exchange. Based on the ID, each message transfers various information concerning that exchange.

There are four types of messages; two of them are sent specifically from the client thread to the server thread and are categorized as request messages, because the client thread is requesting the creation of a new page or a region transfer. The other two messages are sent specifically from the server thread to the client thread and are categorized as response messages since the server thread sends back the requested information. The types of the messages are the following:

- **Request Messages:**
  - **Request New Page:** This message contains information about the region ID the page is to be stored and the size of the requested object. This message type has as the receiver the manager node.
  - **Request Region:** The recipient of this message type is the owner of the region. This message type contains information about the READ-/WRITE choice of the sender, the region ID the sender needs and the
sender's rank. We save the rank in the message because there is a possibility the receiver of the message is no longer the owner of the region. If we only used MPI.Status for retrieving information about the senders then the information about the first sender will be lost. If the owner is found after several message forwards then the response goes back only to the original requester. This is the second request message sent from the client thread to the server thread.

- Response Messages:
  - Transfer Page: This message type declares that a page is to be transferred. It contains information about the total size of the page. This message follows after a request for a region transfer. None of the nodes can send this type of message at any given moment since this is a response message that follows a transfer request.
  - Transfer Completed: This is the last type of the messages, a response message that states that all the pages of the requested region have been safely transferred. This message does not pass any information and it behaves as an ACK message.

The message exchange can be either with the server thread of one node and the client thread of another or between the two server threads, depending on the type of the exchange. A message exchange between two server threads can happen only if a region request has to be forwarded to another node due to the region being moved. Figures 3.1, 3.2 and 3.3 show all the possible message transfers between the threads.

### 3.2 Threads

#### 3.2.1 Locking the Region

In the previous section we mentioned that the region descriptor has a field locked and may be either LOCKED or UNLOCKED. This functionality exists in order for the region not to be transferred to another node before the owner has finished operating on the region's data. Whenever, though, the region is unlocked, there is a possibility that both threads would want to lock it. The server thread locks the region to prevent the client thread from locking it before the transaction is finished. Respectively, the client thread locks the region in order to use the data of the region for its operations, without the server thread being able to transfer it in the middle of an operation.

Because of these circumstances, if we had only a pair of LOCKED/UNLOCKED, in case of LOCKED, none of the two threads would be able to distinguish who has the lock, resulting both threads to think they have previously locked the region. Thus, we have split the LOCKED into server lock SERVER_LOCKED and client lock CLIENT_LOCKED.
3.2. THREADS

3.2.2 Server Thread

The server thread is the basis of our implementation. It runs at the background handling all the incoming messages and depending on the received message, it calls the corresponding functions. It receives the messages using the blocking receive function of the MPI, MPI_Recv(). We take advantage of the blocking feature and use it as a means of synchronizing the messages that are received until the application finishes its run.

When a new message requesting a new page is received, the server thread checks the message for the page’s size to identify the appropriate size group to place the page. If the size is less than the default 4KB, the size of the allocated page change to 4KB. If the size is larger than the default, it remains unchanged. Then, the server thread moves the pointer that marks the position up to which the virtual memory has been allocated. Finally, it transfers the new page back to the client thread.

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 or if the node is not the owner, the server thread forwards the message to the corresponding server thread. If it is the owner, it marks the region as locked by setting the lock as SERVER_LOCKED, if the region was previously released, and starts to send the pages of the region, one at a time, back to the requesting client thread. When the procedure is over, the server thread sends a message declaring the end of the transfer and also the new region descriptor to the receiving client thread. Finally, depending on whether the region was requested for READ or WRITE, the server thread will either make no further action, or change the owner to the dummy descriptor that will continue to exist in its address space.

3.2.3 Client Thread

The client thread runs the application. It calls the initialization function and the finalizing function. During the execution of the application code, it calls the library functions for allocating, deleting and transferring functions.

When the client thread calls the acquire_region() function, it blocks at an MPI_Recv() function waiting for the pages to start arriving. When the first message is received, the client thread learns the size of the page from that message and uses it to receive the second message that contains the data of the page. We have two different messages for this operation because the client thread has to know first the size of the page to be received and then receive its data, since the size is not a value known beforehand. We do not use the function MPI_Probe() provided by the MPI implementation because the client thread at that time can either receive the message with the page size, or the message that denotes the end of the transaction. When the second message with the data is received, the client thread saves the data to the corresponding memory address.
CHAPTER 3. IMPLEMENTATION

Because we do not keep track of all the pages that a region contains, we do not know when the transfer of the region and its pages will be finished. Thus, the owner sends a message denoting the end of the transfer. When such a message is received, the owner sends also the region descriptor for the client thread to change the ownership and lock it, if that is necessary.

Apart from those operations, there is a small alteration regarding the allocation of new pages from the manager node. The manager node is the only node responsible for allocating new pages and transferring them to the other nodes, but concerning the allocation of memory pages for itself, it does not send a message. This occurs because the MPI implementation forbids sending messages to oneself since it has no meaning. Thus, all local allocations made by calling ralloc(), as far as the manager node is concerned, are not handled by the server thread but by the client thread. To avoid a race between the call of ralloc(), we use a POSIX mutex to provide atomic reduction of the memory pointer.

Following, we have three figures that illustrate all the message transfers and the main operations each of the two threads are responsible of. In the first figure, Figure 3.1, we can see the operations and message exchanges regarding a new region request. The client thread of the node with rank No 1, that executes the application code, calls the new_region() function in order to create a new region. This function internally calls ralloc(), which in turn composes a "request new page" message and sends it to the master node.

The server thread of the master node receives the message and calls the function get_new_page(), that is responsible for allocating space in the global virtual memory. After the get_new_page() returns the address of the page that will hold the data of the newly created region, the master node’s server thread sends that page back to the requester, the node No 1. Then, the node No 1’s client thread can initialize the region and continue with the execution of the rest of the application.
3.2. THREADS

The next figure, Figure 3.2, shows the sequence of the messages that are transferred during the acquiring of a region from. Continuing from the previous figure, node No 1 has created a region r, which is now requested from node No 2 for reading purposes. The client thread of node No 2, calls acquire_region(), that creates and sends the message "request region" to the owner, which happens to be node No 1. The server thread of node No 1 receives the messages and calls the function transfer_pages() that transfers the pages one-by-one back to node No 1’s client thread as we have previously explained in section 3.2.2.

The client thread, after each page is received, calls immediately the function unbundle_data() that simply copies the data from the message to the appropriate memory address. At the end of the transaction the server thread sends the message "transfer completed" to inform the client thread that the transfer of the pages is completed, and then sends the region descriptor. The client thread on the other end, first receives the "transfer completed" message and then receives the region descriptor. Since it requested the region with read permissions, no further action is needed to be performed.

![Figure 3.2: State machine for acquiring a region for reading purposes.](image)

In figure 3.3 below, we demonstrate the region transfer for writing purposes. The differences between the figures 3.2 and 3.3 are only the call of the acquire_region() and the actions performed after the termination of the pages’ transfer. From the server thread’s side, after sending the "transfer completed" message, it calls the unlock_region(r) function that frees the region from the server lock that was
Figure 3.3: State machine for acquiring a region for writing purposes.

put on the region previously, during the transferring of the pages. Then, the server thread sends the region descriptor to node No 2 and since the node No 1 does not own the region anymore, it calls the unlink_region(r), which unlinks the region from the siblings and changes the ownership to the new owner.

The client thread of node No 2, after receiving both the "transfer completed" message and the region descriptor, changes the owner in the descriptor and then locks the region by calling the function clientlock(r). Then it can continue the execution of the rest of the application.
Chapter 4

Evaluation

In this chapter we evaluate our allocator using applications also used to evaluate the Myrmics allocator [5]. These benchmarks consist of a microbenchmark and two application quality benchmarks, which we transformed in order to run with our library calls. The applications are (i) a parallel, region-based, Delaunay triangulation application and (ii) a parallel, region-based, Barnes-Hut N-body simulation application. The microbenchmark consists of several library calls that aim to verify the proper operation of the allocator. We chose to use the benchmarks because both are MPI applications that use pointer-based data structures and have been already written to use a region-based allocator. The steps we took for the modification of the benchmarks in order to run with our allocator library were:

1. Replace region creation and deletion functions with our library’s equivalent API.
2. Replace memory allocation and free functions with equivalent calls to the corresponding regions.
3. Remove `sys_send()` and `sys_recv()` function calls 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.

The environment in which we evaluated our runtime was provided by ICS-FORTH and consists of 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.

The implementation we used for the MPI standard is the OpenMPI version 1.6, an MPI-2 implementation [20]. We chose this implementation due to the thread safety it provides since we do an excessive use of the POSIX threads as they are the basis of our allocator. We configured the OpenMPI with the flags `--enable-mpi-thread-multiple` and `--enable-opal-multi-threads` in order to
enable thread safety and give the ability to threads to call MPI functions, with no restrictions. This configuration is a prerequisite for the applications to run error-free with our allocator library. Unfortunately if \texttt{MPI\_THREAD\_MULTIPLE} is not enabled we cannot provide guarantee that the runtime will run without errors.

4.1 Delaunay

The Delaunay triangulation creates a set of triangles from a set of points in a 2D plane such that every triangle connects a number of these points. We port the implementation used in the Myrmics allocator [5], which is based on the serial Bowyer-Watson algorithm implemented by Arens [21]. 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 [5].

The 2D plane in the Delaunay triangulation benchmark is statically divided into a number of regions equal to four times the number of system nodes. All these regions are stored in a hierarchical order such that the top-level region owns the whole plane, its four children own one fourth of that space and so on. Each of these regions store the triangles within their bounds depending on the points that the triangles connect [5].

At the beginning, after the region creation, the 2D plane consists only of the triangles that form the border. Then the initial step of the triangulation follows. The primary node that owns the 2D plane begins the triangulation process by inserting a small, limited, number of points. The triangles are dynamically allocated to the appropriate lowest-level regions of the hierarchy, based on their centers. When the regions have a sufficient number of points in order to create an adequate number of triangles, the primary node transfers its four quadrants to three other nodes and to itself and the algorithm recurses with four times more nodes that repeat the triangulation [5].

After the initial step, the other nodes begin to participate in the triangulation process since the primary node transfers the divided plane with the appropriate points to them. The algorithm processes these points, with a four-phase triangulation, the first phase is called the main triangulation, while the following three phases are called re-triangulations. This four-phase process is needed because the points near the borders of the space owned by a node may need to modify triangles that belong to other nodes. In each triangulation, every node rotates the regions existing in the immediately following level in the region hierarchy and transfers them to the appropriate node. The first re-triangulation rotates the four sub-quadrants to the right, the two next re-triangulations rotate the sub-quadrants down and left, and last the fourth rotation sends the sub-quadrants back to their original position, in order to be split to more nodes [5].
4.1. DELAUNAY

4.1.1 Measurement Methodology

The Delaunay triangulation benchmark requires the number of processes to be a power of 4, thus, we measured performance up to 16 MPI processes that take 32 cores. The total lines of the benchmark, after our alterations described before, are 2200. We have evaluated the delaunay triangulation with 10K, 100K, 1M and 5M points, with 260KB, 1.6MB, 15.8MB and 78.9MB sizes of data respectively.

4.1.2 Results

We measure each step of the algorithm based on the operation performed. The algorithm consists of the following steps: "main triangulation", "retriangulations", "shift regions" and "split points", as we described previously. The phases of main triangulation, retriangulation and shift regions consist only of computational operations while the phase of shift regions consists of region transfers and thus we declare it as a communication phase. We also measure the time waiting at each barrier as idle time.

Table 4.1 below shows the run times and speedups of each dataset. We present the total run times with one, four and sixteen nodes and compare the speedups of our allocator with the state of the art Myrmics allocator.

<table>
<thead>
<tr>
<th>Workload</th>
<th>No of Nodes</th>
<th>Allocator</th>
<th>Myrmics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total time (s)</td>
<td>Speedup</td>
</tr>
<tr>
<td>10 K</td>
<td>1</td>
<td>0.148669</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.056841</td>
<td>2.616</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.039543</td>
<td>3.759</td>
</tr>
<tr>
<td>100 K</td>
<td>1</td>
<td>3.476154</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.742983</td>
<td>4.679</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>0.223806</td>
<td>15.532</td>
</tr>
<tr>
<td>1 M</td>
<td>1</td>
<td>99.123881</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>19.030056</td>
<td>5.209</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>3.3068781</td>
<td>29.975</td>
</tr>
<tr>
<td>5 M</td>
<td>1</td>
<td>933.81514</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>173.76662</td>
<td>5.374</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>31.627913</td>
<td>29.525</td>
</tr>
</tbody>
</table>

Table 4.1: Speedup of the Delaunay triangulation.
Figure 4.1: Graphs for the Delaunay Triangulation with different sizes of data.
4.2. **BARNES-HUT**

Figure 4.1 shows the results for two small datasets of 10K and 100K and two large datasets of 1M and 5M points. For the dataset of 10K points, figure 4.1a shows a somehow steady $2 \times$ speedup for our allocator for 4 and 16 cores, while the Myrmics allocator scales for $3 \times$ for 4 cores but for 16 cores the speedup drops due to idle time spend waiting for communication, as shown by the breakdown in Figure 4.1b.

For the dataset of 100K points, Figure 4.1c shows speedup is close to linear for our allocator, while the Myrmics allocator scales to $10 \times$ for 16 cores, mainly due to higher idle time spent waiting for communication, as shown again by the breakdown in Figure 4.1d.

For the two large data sets, Figure 4.1e and Figure 4.1g show that our allocator behaves very similarly to the Myrmics allocator, attaining a superlinear speedup for 16 MPI processes. Moreover, Figure 4.1f and Figure 4.1h show that the behavior of the two allocators is similar also for time spent in communication and idling.

## 4.2 Barnes-Hut

The third benchmark is an application that performs the Barnes-Hut algorithm, an N-body simulation, to calculate the movement of a number of astronomical objects based on their gravitational effects by using the center of their masses. We port the implementation used in the Myrmics allocator [5], which is based on the MPI-based Dubinski 1996 approach [22].

In this Barnes-Hut benchmark, each node builds a local oct-tree in each simulation step for each body that it owns. For each level of the oct-tree, we create a different memory region. Inside of each region are saved the local bodies and their bounding boxes. The bounding boxes are communicated in pairs with all the nodes. The nodes then calculate what part of the oct-tree they need to receive from the communicating peer. Each node sends the region and its subregions that contain the bodies [5].

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 [5].

### 4.2.1 Measurement Methodology

The transformations we did to this implementation in order to support our library were as described in the previous subsections. The Barnes-Hut benchmark requires the number of nodes to be a power of two due to the recursive bisection, thus, we measured performance up to 32 MPI processes that take 64 cores. The total lines
CHAPTER 4. EVALUATION

after the transformations are 3700. We have evaluated the barnes-hut algorithm with multiple workloads; 8KB, 32KB, 42KB, 64KB and 82KB of bodies.

4.2.2 Results

For each run of the Barnes-Hut application, the bars in the breakdown figures in Figure 4.3 and Figure 4.2 show the average time of each node. The time is then divided in the time is spent for the communication between the nodes, the local computations and the idle time spent waiting for message responses.

Table 4.2 below shows the run times and speedups of each dataset. We present the total run times with one, two, four, eight, sixteen and thirty-two nodes and compare the speedups of our allocator with the state of the art Myrmics allocator.

Figures 4.3 and 4.2 show the results of running the Barnes-Hut simulation for 8 KB, 32 KB, 42 KB, 64 KB and 82 KB of input data. Figure 4.2a and Figure 4.3a shows the speedup attained for processing 8KB of data and 82KB, 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 our allocator scales slightly better above 8 processes for both datasets. Figure 4.2a and Figure 4.3b show 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 4.2c and Figure 4.2g show the speedup for an input set of 32KB and 64KB respectively, for various numbers of MPI processes. In both datasets the scale of the two allocators drops after eight nodes. This behavior occurs because after eight nodes, the bodies are not divided equally for each node. Our allocator scales better than the Myrmics allocator for eight nodes and above, mainly due to the communication overhead being bigger for large core counts, as shown in both Figure 4.2d and Figure 4.2h. 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.

The dataset with 42KB number of bodies presents the most uncommon behavior regarding the other results, as shown in Figure 4.2e for various numbers of MPI processes. In this dataset the bodies are not distributed evenly to all nodes, apart from the use of 4 or 32 system nodes. It is important to note that the bodies are distributed among cores based on their bounding boxes and are never divided based on the cores working on the benchmark. In Figure 4.2f we can see that the idle time is larger than in the other breakdown figures, which proves our point.
### Table 4.2: Speedup of the Barnes-Hut N-body simulation.

<table>
<thead>
<tr>
<th>Workload</th>
<th>No of Nodes</th>
<th>Allocator</th>
<th>Total time (s)</th>
<th>Speedup</th>
<th>Myrmics</th>
<th>Total time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 KB</td>
<td>1</td>
<td>20.26382</td>
<td>1</td>
<td>20.91199</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15.69925</td>
<td>1.2907</td>
<td>16.58895</td>
<td>1.2606</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8.898296</td>
<td>2.2772</td>
<td>9.29796</td>
<td>2.24909</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5.535218</td>
<td>3.6608</td>
<td>5.98216</td>
<td>3.49572</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4.613131</td>
<td>4.3926</td>
<td>5.53708</td>
<td>3.77672</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4.354879</td>
<td>4.6531</td>
<td>6.57228</td>
<td>3.18184</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32 KB</td>
<td>1</td>
<td>117.448035</td>
<td>1</td>
<td>118.22207</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>67.2023195</td>
<td>1.7476</td>
<td>69.5094425</td>
<td>1.70081</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>41.732007</td>
<td>2.8143</td>
<td>42.864975</td>
<td>2.75801</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>22.2567967</td>
<td>5.2769</td>
<td>23.955417</td>
<td>4.93509</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>26.5054589</td>
<td>4.4310</td>
<td>28.3566381</td>
<td>4.16912</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>26.3454590</td>
<td>4.4579</td>
<td>30.5176958</td>
<td>3.87389</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42 KB</td>
<td>1</td>
<td>159.478714</td>
<td>1</td>
<td>160.993414</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>169.822740</td>
<td>0.9390</td>
<td>173.433439</td>
<td>0.92827</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>45.9728695</td>
<td>3.4689</td>
<td>48.9051127</td>
<td>3.29195</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>82.2785891</td>
<td>1.9382</td>
<td>86.5548043</td>
<td>2.61545</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>55.2009551</td>
<td>2.8890</td>
<td>45.6781341</td>
<td>3.52452</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>52.3202269</td>
<td>3.0481</td>
<td>40.7869467</td>
<td>3.94718</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64 KB</td>
<td>1</td>
<td>207.560003</td>
<td>1</td>
<td>210.297247</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>179.833417</td>
<td>1.1541</td>
<td>185.676712</td>
<td>1.13259</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>116.135181</td>
<td>1.7872</td>
<td>120.486716</td>
<td>1.74539</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>40.3224401</td>
<td>5.1475</td>
<td>44.1239681</td>
<td>4.76605</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>60.893504</td>
<td>3.4085</td>
<td>65.9373945</td>
<td>3.18935</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>58.2442941</td>
<td>3.5636</td>
<td>69.0151485</td>
<td>3.04712</td>
<td></td>
<td></td>
</tr>
<tr>
<td>82 KB</td>
<td>1</td>
<td>225.372156</td>
<td>1</td>
<td>226.470918</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>176.569418</td>
<td>1.2763</td>
<td>186.388345</td>
<td>1.21504</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>131.090008</td>
<td>1.7192</td>
<td>136.804398</td>
<td>1.65543</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>50.9004103</td>
<td>4.4277</td>
<td>55.1626463</td>
<td>4.10551</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>43.1823634</td>
<td>5.2190</td>
<td>48.4388842</td>
<td>4.67539</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>46.1586648</td>
<td>4.8825</td>
<td>51.7041035</td>
<td>4.38013</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.2: Graphs for the Barnes-Hut N-body simulation with different workload sizes.
4.2. BARNES-HUT

Figure 4.3: Graphs for the Barnes-Hut N-body simulation with different workload sizes. 

(a) Speedup graph for 82KB data.  

(b) Breakdown graph for 82KB data.
Chapter 5

Conclusions and Future Work

We presented DRASync, a region-based memory allocator for MPI programs that supports pointer-based data structures and provides an intuitive synchronization abstraction to programmers, similar to reader-writer locks, on top of a DSM-like system. We provided a portable library that creates a distributed shared memory on top of multiple nodes. We used MPI for communication between nodes, because apart from providing communication functionality, it also gives an essential virtual topology and synchronization between a set of processes that have been mapped to nodes, servers or computer instances. Also, MPI is language-independent and thus enhance the portability of our library.

We proposed a region allocator API, with a centralized allocator and deallocator, named master node, and with a decentralized system for communicating regions between nodes. Along with the region allocator, we proposed a high-level synchronization abstraction in which dynamic regions are treated like reader-writer locks. The combination of the DSM features and thus, the sharing of the regions, the data structures' pointers are now meaningful to all nodes of the system, since each pointer refers to an address of the global space and every node can refer to that address without it having to perform any memory address translation.

We evaluated our implementation on two application-level benchmarks, the Barnes-Hut N-body simulation and the Delaunay Triangulation, and compare the results with an existing state-of-the-art distributed allocator. Our allocator proved to perform equally well on average, while offering an intuitive mutual exclusion synchronization mechanism for MPI programs and abstracting the location of data from the programmer.

5.1 Future Work

Task-based programming models provide a fine-grained and structured way of expressing parallelism than threads because they emphasize the parallel nature of the processing. Thus, in addition to the current functionality, we intend to support task parallelism. DRASync can use also a task scheduler to manage the task
footpring and schedule parallel tasks on nodes.
Bibliography


