Sparsely Faceted Arrays: A Mechanism Supporting Parallel Allocation, Communication, and Garbage Collection

by

Jeremy Hanford Brown

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by . . . .
Thomas F. Knight, Jr.
Senior Research Scientist
Thesis Supervisor

Accepted by. . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Sparsely Faceted Arrays: A Mechanism Supporting Parallel Allocation, Communication, and Garbage Collection

by

Jeremy Hanford Brown

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Conventional parallel computer architectures do not provide support for non-uniformly distributed objects. In this thesis, I introduce sparsely faceted arrays (SFAs), a new low-level mechanism for naming regions of memory, or facets, on different processors in a distributed, shared memory parallel processing system. Sparsely faceted arrays address the disconnect between the global distributed arrays provided by conventional architectures (e.g. the Cray T3 series), and the requirements of high-level parallel programming methods that wish to use objects that are distributed over only a subset of processing elements. A sparsely faceted array names a virtual globally-distributed array, but actual facets are lazily allocated. By providing simple semantics and making efficient use of memory, SFAs enable efficient implementation of a variety of non-uniformly distributed data structures and related algorithms. I present example applications which use SFAs, and describe and evaluate simple hardware mechanisms for implementing SFAs.

Keeping track of which nodes have allocated facets for a particular SFA is an important task that suggests the need for automatic memory management, including garbage collection. To address this need, I first argue that conventional tracing techniques such as mark/sweep and copying GC are inherently unscalable in parallel systems. I then present a parallel memory-management strategy, based on reference-counting, that is capable of garbage collecting sparsely faceted arrays. I also discuss opportunities for hardware support of this garbage collection strategy.

I have implemented a high-level hardware/OS simulator featuring hardware support for sparsely faceted arrays and automatic garbage collection. I describe the simulator and outline a few of the numerous details associated with a “real” implementation of SFAs and SFA-aware garbage collection. Simulation results are used throughout this thesis in the evaluation of hardware support mechanisms.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist

Acknowledgments

During my time at MIT I have received innumerable kinds of help from uncountably many people. It pains me that I have neither the space, nor the accuracy of memory, to thank them all personally here; I beg forgiveness from those whom I fail to acknowledge.

First and foremost among those who have helped me produce this thesis is Katie Hayes, to whom I am immensely grateful. Without her unflagging moral and emotional support, I would have been locked in a small rubber room weeks ago. Without her technical assistance, I would still be preparing slides for my defense talk. Katie, if I'm graduating, it's because of you. You are the best.

I want to thank Tom Knight, my eternally patient advisor, for providing an environment genuinely open to unorthodox thinking. I also want to thank André DeHon, who first introduced me to Tom and the AI lab.

I am indebted to Anthony Zolnik and Marilyn Pierce, extraordinary administrators who have gone out of their way to help me time and time again. I also want to thank Anthony for his invaluable suggestions on my writing.

I want to thank all of the friends who have helped me through the years at MIT — there are too many of you to list here, but you know who you are. I do, however, want to single out three people for special attention.

Michelle Goldberg has been an extraordinary friend to me for over ten years. My friendship with her has been one of the foundations of my experience at MIT.

Nick Papadakis is another long-time friend. He has helped me time and time again on matters ranging from auto repair to grad school recommendations; he has also hired me repeatedly.

Marc Horowitz, past and present roommate, in the last year has taken on the additional roles of landlord, sysadmin, consultant, and source of encouragement. In addition to providing me with a place to live and work, he has been an incredible source of help whenever I have had to wrestle with recalcitrant computers and C code.

I want to thank my family for their encouragement and their patience. When I came to MIT as a freshman in 1990, I'm sure that nobody expected I would still be a student in 2002; I am greatly appreciative of my family's continuing support in the face of repeatedly-slipping graduation dates.

This thesis has benefited immensely from conversations with Kálmán Réti, Jan Maessen, Tom Knight, my thesis readers Gill Pratt and Gerry Sussman, and all of the members, past and present, of the Aries research group, including Andrew "bunnie" Huang, J.P. Grossman, Ben Vandiver, Josie Ammer, and Dom Rizzo. (Bunnie also deserves a special mention for helping with some of the technical details of thesis production.)

Of course, any errors, omissions, etc. are entirely my responsibility!

This thesis describes work performed under DARPA/AFOSR Contract Number F306029810172.

In memory of Laurie Wile.

Contents

List of Figures

Chapter 1

Introduction

Although microprocessor performance continues to improve at an astonishing rate, there remain problems which cannot be practically solved using only one processor. To solve such a problem with adequate speed, one must use multiple processors to work on subproblems in parallel.

Computer programming is an inherently complex task. Programming parallel computers requires the coordination of a number of processors to accomplish a single task, and is therefore even more complex.

This is a thesis about reducing the complexity of parallel programming by raising the level of abstraction available to the parallel programmer. In particular, it is about making it possible to efficiently implement an important class of data structures on an important class of parallel computer architectures.

In the following section I describe the challenges addressed by this thesis, illustrating the inadequacy of conventional parallel architectures. Next, I outline the major research contributions this thesis makes in addressing these challenges. Finally, I outline the structure of the rest of this document.

1.1 Challenges

1.1.1 A Class of Parallel Architectures

The class of parallel computer architectures addressed in this thesis is represented by machines such as the Cray T3D [59] and T3E [62], and the NEC SX series of vector supercomputers. This class has several defining characteristics.

- Massively Parallel Processors (MPP): These machines are massively parallel; they are scalable to hundreds or thousands of processors.
- Distributed, Shared Memory (DSM): Each processor is bundled into a node along with some memory. There is hardware support for each processor to access memory in other nodes.
- Non-Uniform Memory Access (NUMA) times: Access to memory within a node is low-latency; access to memory in other nodes is higher-latency, often varying depending on the relative locations of the two nodes within the machine.
- No inter-node data caching: Unlike cache-coherent NUMA (ccNUMA) architectures such as DASH [37], FLASH [31], and the SGI Origin [67], these explicitly NUMA architectures do not attempt to conceal inter-node memory latency with a complex data caching strategy.

I shall refer to this class of architectures as the NUMA-DSM class; NUMA-DSM machines are the most common type of shared-memory parallel computer available today.

NUMA-DSMs have several appealing characteristics.

Scalability is straightforward: each new node brings with it both processing and memory resources.

Hardware-supported shared memory provides an extremely low-latency mechanism for inter-node communication.

By opting not to attempt cache-coherent data sharing between nodes, the design is kept relatively simple. This design also works to the benefit of many irregular computations which tend to miss in caches; for such applications, the absence of overhead from a cache hierarchy is beneficial.

1.1.2 A Class of Parallel Data Structure

In a parallel computer, a distributed object[24] is an object that has been allocated memory on multiple nodes. The *name* of a distributed object somehow identifies all of these constituent pieces of memory; using the name and an index, it is possible to access each piece of memory in turn.

It is desirable to be able to create a distributed object that has memory only on a subset of the nodes of a parallel machine. Such non-uniformly distributed objects are useful in implementing hierarchical data structures with replicated components, and in implementing divide-and-conquer algorithms such as quicksort.

Such "partitioning" data structures and algorithms use large numbers of non-uniformly distributed objects, each potentially distributed over a different set of processors. To support such fine-grain usage, allocating and using non-uniformly distributed objects must have low overhead.

1.1.3 An Architectural Disconnect

Unfortunately, conventional NUMA-DSM architectures do not provide support for non-uniformly distributed objects. On these architectures, objects are allocated on a single node or distributed uniformly over every node. Since the overhead of implementing non-uniformly distributed objects in software is enormous, conventional parallel programs are unable to use them in fine-grain fashion.

This is the first of two major problems addressed by this thesis. To solve it, I describe *Sparsely Faceted Arrays*, a hardware-supported mechanism that enable the implementation of non-uniformly distributed objects.

1.1.4 The need for automatic memory management

Automatic memory management is the second major problem addressed by this thesis.

Even in a single-processor environment, memory management is one of the most error-prone aspects of programming. Explicit memory management makes the programmer responsible for requesting memory when needed, and “freeing” it when it is no longer needed. Programs written using explicit memory management often suffer bugs by freeing memory that is still being used, or by failing to free memory after it is no longer used.

On the other hand, automatic memory management places the burden of detecting unused memory on an underlying system of “garbage collection.” Programs using automatic memory management still request memory when they need it. However, an automatic garbage collector is made responsible for detecting objects that are no longer in use, and reclaiming their memory for future re-use.

In general, programs written using automatic memory management suffer dramatically fewer bugs than those using explicit memory management.

Parallel architectures and distributed objects add to the complexity of the memory management task; non-uniformly distributed objects add further complication.

Current, conventional parallel programming systems do not provide garbage collection. Those few parallel programming systems that have provided parallel garbage collection have not, in general, addressed the issue of garbage collecting non-uniformly distributed objects.

1.2 Research Contributions

This thesis offers three major research contributions.

The most important contribution is Sparsely Faceted Arrays (SFAs). Sparsely faceted arrays are a new, parallel data structure that enable the efficient, straightforward implementation of non-uniformly distributed objects. A Sparsely Faceted Array is virtually a global, uniformly distributed array; in actuality it is sparse, as regions of memory on individual nodes — facets — are allocated lazily. In order to illustrate their usefulness, I discuss the use of SFAs in several parallel applications. Sparsely faceted arrays require simple hardware support; I describe the implementation requirements and evaluate the effectiveness of additional support mechanisms in simulation.

Although the high-level semantics of SFAs are quite simple, the underlying lazily-allocated data structure is inherently complex; automatic management, including garbage collection, is essential to preserving the simplicity of the mechanism presented to the parallel programmer. This leads to the other, garbage-collection-related contributions of this thesis, discussed below.

The traditional approach to parallel GC has been based on parallelizing precise, tracing garbage collectors, e.g. mark/sweep. As my second major contribution, I prove that in the worst case, "precise" tracing garbage collectors can eliminate the entire performance benefit of parallel processing.

In a minor related contribution, I also present a novel garbage collection algorithm which, given an oracle, is precise; I leave open the question of whether or not a heuristic approximation to an oracle exists which could make the algorithm effective in practice. Due to its ineffectiveness, I do not use this algorithm in the remainder of the thesis.

As my third major contribution, I present a novel scalable, parallel garbage collection algorithm capable of managing sparsely faceted arrays. This strategy meets two key requirements for managing sparsely faceted arrays: correctness, which requires that no facets be garbage collected until there are no live references to the SFA anywhere in the system; and efficiency, which requires that freeing an SFA does not require communication with nodes that never received pointers to the SFA. This GC scheme requires simple hardware support which in large part parallels that required for SFAs. I describe the implementation requirements, and evaluate additional supporting mechanisms in simulation.

1.3 Thesis Road Map

The remainder of this document is organized in the following fashion.

First, in Chapter 2 I review prior work on parallel programming models, garbage collection, and various related topics.

Next, in Chapter 3, I introduce Sparsely Faceted Arrays. I present examples of their use; describe their implementation; and present the results of simulating a hardware support mechanism.

In Chapter 4, I prove that precise tracing garbage collectors can eliminate the entire performance benefit of parallel processing. I also present a novel, impractical garbage collection algorithm, leaving open the question of whether it could be made useful in practice; I make no further mention or use of this algorithm.

In Chapter 5 I move from theoretical matters to practical matters by introducing a novel parallel garbage collection algorithm for garbage collecting sparsely faceted arrays. I discuss the algorithm's implementation, and present the results of simulating a hardware support mechanism.

In Chapter 6, I describe Mesarthim, the high-level simulation system used to generate the simulation results used in Chapters 3 and 5. Whereas previous chapters maintain generality as much as possible, in this chapter I describe many of the details associated with a specific implementation of sparsely faceted arrays and parallel garbage collection. I also discuss the node-local garbage collection strategy that cooperates with the inter-node parallel GC strategy of the previous chapter.

Finally, in Chapter 7, I summarize the key points and results of the work described in this thesis.

Chapter 2

Related Work

In this section, I will briefly discuss previous work related to this thesis. In particular, I will discuss mechanisms for naming distributed objects in the NUMA-DSM class of parallel architectures, and I will discuss work on parallel, distributed, and large-heap garbage collection.

2.1 Naming distributed resources

In this section I discuss mechanisms for naming distributed objects in NUMA-DSMs. Such mechanisms have been provided at the hardware, library, and programming language level, and I discuss each in turn.

2.1.1 Hardware support

In conventional NUMA-DSM architectures such as the Cray T3D [59] and T3E [62], the NEC SX series, etc. node manages its own local memory.

Processes running on multiple nodes can conspire to allocate the same local memory locations to a distributed object; the name of the common location thus becomes the name of the distributed object. This approach requires that all of the conspiring nodes have similar allocation situations, and thus typically all nodes are required to participate in all allocations — in other words, every distributed object is distributed over every node in the system. The Cray/SGI shmem [12] libraries provide a specific example, requiring that any distributed memory object exist over all nodes; such an object is called a “symmetric data object” in the shmem documentation. In fact, shared memory operations may only be carried out on symmetric data objects — i.e. there are no provisions at the library level for allocating referencing scalar objects on independent nodes.

One notable departure from the conventional approach is the J-machine [51] parallel computer. In the J-machine, all references to objects, distributed or otherwise, are indirected through a segment table on each node; this style of addressing is similar to that used by early *capability* [14] architectures [39].

Using indirection tables allows the J-machine to provide distributed objects with arbitrary, globally unique names. The COSMOS operating system [25] uses the J-machine’s translation tables to name “aggregate objects”. Aggregate objects are composed of representative objects distributed over some subset of the system’s nodes; each node holding a “representative” of an aggregate object has a translation entry mapping from the aggregate object’s name to the local representative’s capability. To keep track of which nodes actually have allocated space, COSMOS encodes information in a distributed object’s name specifying the placement of its constituent objects. Under this encoding, if there are fewer constituent objects than there are nodes in the machine, the constituents are placed in such a manner as to provide an even distribution over the entire machine.

The J-machine suffers from the problem, common to early capability systems, that indirecting every memory access through a segment table is inefficient; [51] reports that in practice, an unacceptably large percentage of program time is spent engaged in translation.

The M-machine [15] multicomputer, a successor to the J-machine, provides direct addressing. It supports a coarse-grained mechanism for distributing resources over variable regions of the architecture. In particular, the M-machine’s page-translation mechanism provides partitions over multiple adjacent nodes; partitions are at the page granularity, and have dimensions measured in powers of 2. Each distinct partition requires a separate TLB entry; objects distributed over distinct partitions must therefore be stored on different pages.

2.1.2 Library support

PVM [19] and MPI [17, 18] are popular, oft-ported libraries for parallel programming based on message-passing. Both PVM and MPI provides the ability to name distributed resources in a coarse-grained fashion by assigning processes to *groups*; the name of a group thus denotes a set of distributed resources, in this case processes.

The recent MPI 2.0 standard [18] goes one step further. A group of processes may simultaneously allocate local "windows" of memory which are then directly accessible by other processes in the group. Using the put and get operations on windows is known in MPI as "one-sided communication", since, on a shared-memory architecture, only one processor needs to be involved in the operation.

Since each window may have different address, length, and other properties, in practice every process is compelled to record the attributes of every window in every other process (e.g. [69] which discusses implementing one-sided communications on the NEC SX-5); in a group of N processes, this means that there is an $\mathrm{O}(N)$ storage requirement per process, per window.

2.1.3 High Level Parallel Programming Models and Languages

SIMD languages such as HP Fortran [30], *Lisp [42], and APL [20] provide data-parallel primitives which are a thin veneer on standard vector-parallel operations.

"Idealized" SIMD languages such as Paralation Lisp [60] and NESL[8] express parallelism with parallel-apply operations performed over data collections (typically vectors); nested parallelism is allowed, but is compiled into non-nested, vector-parallel operations.

None of these languages is well-suited to distributing objects non-uniformly over a multiprocessor.

Concurrent Aggregates [10] and Concurrent Smalltalk [24, 23] are languages specifically intended for the J-machine. Each is based on the notion of essentially replicating important data structures spatially in order to provide means for parallel access. The languages do not explicitly reveal distributed object placement to the programmer; the programmer merely specifies at allocation-time the number of representative objects that an aggregate object should contain.

2.2 Parallel, Distributed, and Area-Based Garbage Collection

There are an incredible number of schemes for parallel, distributed and segmented garbage collection. In this section I will focus on garbage-collection schemes with particularly compelling relationships or contrasts to the work presented in this thesis.

It is not my intent in this section to duplicate the efforts of many fine survey works, but rather to present the salient details of those schemes which are directly relevant, either by similarity or by significant contrast, to the approach I am taking.

2.2.1 GC survey works

Classical GC

Rather than attempt a survey of approaches to uniprocessor garbage collection, I shall refer the reader to Jones' and Lins' *Garbage Collection* [27], an excellent survey of traditional garbage collection techniques including, but certainly not limited to, copying garbage collection, mark-sweep GC, and reference-counting. Its coverage of distributed and parallel garbage collection is somewhat sparse.

Distributed GC

Although I shall discuss most of the major approaches to the distributed GC problem below, additional approaches and references are presented in the survey paper [56]. Another presentation and detailed analysis of several distributed GC schemes is presented in chapter 2 of the thesis [43]. Additional references may be found in the bibliography paper [61], which includes not only *avant-garde* topics such as parallel and distributed GC, but also a great many references on classical GC techniques.

2.2.2 Copying GC

System-wide copying GC doesn't actually use areas to any great effect; each processor is responsible for GCing its own region of memory, but GC is system-wide and requires cooperation between all the processors. When a GC is started, each processor begins a copying collection from its local roots. When it encounters a local pointer, it copies as normal; when it encounters a remote pointer, it sends a request to the processor owning the target object which causes that processor to copy from that pointer. This approach is used by the Scheme81 processor [4] in multiprocessor configurations, and by the distributed garbage collector for the parallel language Id described in [16]. Termination detection is an important issue in copying GC. The Id garbage collector detects termination by a messaging protocol,very similar to the scheme described in [58], based on arranging all the processors in a logical ring.

In general, since copying GC requires GCing the entire heap at once, it is inappropriate as the only means of GC in a large parallel system, and is always inappropriate when the size of the heap exceeds that of main memory.

2.2.3 Reference listing

Schemes which use reference listing keep track of which remote areas have pointers to an object in a particular area; the IN list serves as a set of roots for performing independent, area-local GCs which do not traverse references that point out of the area.

Note that reference listing requires that whenever a pointer into area A is copied from an area B to an area C, a message must also be sent to area A to create the appropriate IN entry.

Some reference listing schemes have an entry for every instance of a pointer in a given area; others have one entry for an area regardless how many copies of the pointer there are in that area.

ORSLA

The first area-based garbage collection system in the literature is the reference-listing scheme described in [7] as part of the custom-hardware capability system ORSLA. ORSLA has an IN and OUT entry (actually the same object, an Inter-Area-Link (IAL), threaded onto one area's IN list and the other area's OUT list) for every single inter-area pointer. Whenever an inter-area pointer is created, ORSLA's hardware detects the event and updates the appropriate IN and OUT lists. Area-local GC is performed with a copying collector which copies data objects and IALs; at the end of an area-local GC, the old set of data and IALs is freed.

Inter-area pointers are actually indirected through their IAL under normal circumstances; an area may be "cabled" to another, however, in which case pointers from the first to the second may be "snapped" to point at their targets directly. Note that this means that while the first area may be GCed independently, the second area may only be GCed simultaneously with the first, since its IN list does not record incoming pointers from that area.

ORSLA eliminates garbage cycles by migrating objects which aren't reachable from area-local roots into areas that reach them; in theory, this eventually causes a garbage cycle to collapse to a single area, at which point area-local GC destroys it. [7] doesn't work out details insuring that migration terminates. For many architectures, the overhead of an IAL for every inter-area pointer would be unacceptable due to the consumption of per-node physical memory; additionally, having one IAL object serve as both an IN and an OUT entry requires that area-local GC operations require inter-area update messages.

Thor

A reference-listing scheme was designed for Thor [41], a persistent, distributed, object-based database for which garbage collection has been the subject of much research. Thor runs on a heterogeneous collection of conventional hardware; object references are opaque (indirect), thus allowing intra-area object relocation without requiring inter-area communications.

The garbage collection scheme described in [43] uses conservative reference lists between operating regions (ORs) of the database itself, and between ORs and front-end (FE) clients. The scheme is designed to be fault-tolerant in the face of lost or delayed messages.

Each OR maintains a table of objects reachable from other ORs, listing for each object which other ORs might have references. Each OR maintains a similar table for objects reachable from FEs. When an FE or an OR performs a local GC, it sends a “trim” message to each OR listing only those objects still reachable from the OR/FE, thus allowing the OR to clear some entries in the appropriate table. If a trim message is lost, no harm is done, since the table is conservative; the next trim message will fix the problem. Timestamping prevents delayed trim messages from removing entries created after the message was initially sent. Since FE clients are relatively unreliable, FE entries are “leased” — if not renewed periodically, they expire and can be cleared. Note that inter-area garbage cycles aren’t reclaimed at all by this scheme.

The problem of inter-area garbage cycles in Thor is taken up in [44], which adds a migration scheme on top of the reference listing scheme. Each object is marked with its distance from its nearest root at local GC-time; objects which are only rooted via IN entries start with the distance at that entry and increase it as they proceed. Distances are exchanged with “trim” messages. Thus, while rooted (i.e. reachable) data distances remain finite, the distances associated with garbage cycles will always increase with each round of local GC and trim-message exchanges. A cutoff threshold on distance dictates when migration should occur; an estimated target-node for migration is propagated with distance values, thus causing most objects in a garbage cycle to immediately migrate to the same area.

This strategy has several drawbacks. The overhead of translating opaque references is high. Repeatedly broadcasting trim lists means that bandwidth is *continually* consumed in proportion to the number of inter-area references. Finally, migrating garbage to discover cycles consumes bandwidth in proportion to the amount of garbage, but garbage is the last thing to which we wish to dedicate precious communications bandwidth.

2.2.4 Reference flagging

Reference flagging systems are extremely conservative: each area maintains an IN list which records all objects for which remote references might exist. An entry is created for an object when a reference to that object is first stored into a remote area. The primary advantages of reference flagging are first, that no inter-area messages are needed when a pointer into area A is copied from area B to area C; and second, that IN lists may be extremely compact. The disadvantage is that some form of global knowledge is needed to remove conservative IN entries and inter-area garbage cycles.

Hughes [26] attacks the problem of global garbage collection using a timestamp-based scheme in which inter-area messages are assumed to be reliable, and nodes are assumed to have synchronized clocks. Remote pointers are indirect. Area roots (including IN entries) and remote pointers carry timestamps. Area-local GC propagates the most recent timestamp from roots to remote pointers;- active process objects, the true roots of the system, are timestamped with the time at which the local GC begins.

At the end of the local GC, the timestamps on remote pointers are propagated to the IN entries in their target areas. Each area keeps track of the oldest timestamp that it may not have propagated yet, called the “redo” time. The globally oldest redo time is called “minredo”; at any time, any IN entry which is timestamped with a time older than minredo is garbage and may be deleted. Minredo is calculated using the ring-based termination-detection protocol described in [58].

The reference flagging scheme described in [32] uses Hughes’ algorithm adapted to cope with unreliable hardware, network, and messages, and also loosely synchronized clocks. Heap segmentation is at the node granularity; intra-node pointers are direct, but inter-node pointers are indirect in order to allow node-local relocation of objects. The scheme relies on a high-reliability central service to preserve certain information across individual node-crashes. Areas occasionally send their OUT lists to the central service; the service uses the collected OUT lists of all areas to compute less-conservative IN lists for each area. Additionally, the Hughes algorithm variant runs entirely on the central service. A variety of additional complexity underlies the correct operation of this system in the face of various message and node failures.

The major drawback of reference flagging is that each pass at eliminating conservatively-created IN entries involves communications and work overhead proportionate to the number of live entries.

2.2.5 Distributed reference counting

In distributed reference counting (DRC) [38], for each remotely reachable object in an area, the area maintains a count of the number of remote areas which can reach that object. When an object's reference count goes to zero, it is no longer remotely reachable. DRC has roughly the same messaging requirements as reference listing, but obviously has much smaller memory overhead – one counter per remotely reachable object, regardless of how many areas it is reachable from. Distributed reference counting is much more vulnerable to message loss, duplication, and reordering than reference listing; significant care must be taken to avoid race-conditions. DRC does not collect inter-area garbage cycles.

The DRC-based scheme proposed in [50] for use with the Thor [41] distributed database system is particularly interesting because it uses logging to defer updating reference counts immediately, and thus avoids the need to page in IN entries for an area every time the number of outstanding references changes.

A more complex use of logging for GC Thor is described in [45]. In this scheme, each area maintains IN and OUT lists, where an IN list contains precise counts of external references to objects in the partition, while an OUT list precisely identifies all outgoing pointers. Partition-local garbage collection uses objects in the IN list as (part of) the partition's root set. Rather than maintain IN and OUT lists eagerly, this scheme records all object modifications in a globally shared log. The log is scanned to generate an up-to-date IN list prior to garbage collecting a partition. IN and OUT lists are broken into several parts, however, in order to avoid having to go to disk on every update. A number of synthetic benchmarks demonstrate the effects of tuning various parameters related to the sizes of the "basic", "potential", and "delta" lists.

Inter-area cyclic garbage is collected using an incremental marking scheme piggybacked on partition-local garbage collection. Marking begins at global roots; marks are propagated intra-area by local GC, which also pushes marks across OUT pointers. A mark phase terminates when every area has performed a local GC without marking any new data; at that point, any unmarked data may be discarded.

2.2.6 Weighted reference counting

Weighted reference counting was independently described at the same time in both [6] and [70]. In weighted reference counting each object has a weight assigned to it at allocation time, and each pointer to that object also has a weight. Pointer weights are powers of two.

When a pointer is duplicated, its weight is reduced by half, and the duplicate receives the other half of the weight; thus, duplicating a pointer requires no communication with the area holding the target object. When a pointer is deleted, a message is sent which causes the weight of the pointer's target object to be decremented by the weight of the pointer. When the object's weight reaches zero, there are no outstanding pointers to the object and it may be reclaimed. There are no synchronization issues with weighted reference counting. WRC does not collect garbage cycles.

One problem with WRC as described is that when a pointer's weight hits one, evenly splitting it becomes impossible. [6] suggests that at this point a new indirection object is created with a large weight; the indirection object contains the weight-one pointer to the actual target object, but the results of the pointer duplication are pointers to the indirection object. This avoids any need to send messages to the original object upon pointer creation, at the cost of memory and indirection overhead.

Note that this scheme as described does not really make use of areas, but it is a a straightforward extension to move the weights to IN and OUT list entries on a per-area basis, rather than maintaining them in every single object and pointer. Such a scheme would reduce the space overhead of normal objects and pointers, and enable arbitrary per-area garbage-collection that would recover intra-area cycles.

[11] proposes a slightly different approach to duplicating a pointer whose weight has dropped to one; the weight in a pointer is tagged as being either original or borrowed

weight. When a pointer with weight one (original or borrowed) is duplicated, its weight is replaced by a borrowed-weight of MAX, and MAX is added to an entry for the pointer in a *reference weight table*. When a pointer with borrowed weight is destroyed, the borrowed weight is subtracted from the table entry, whereas when a pointer with original weight is destroyed, the weight is subtracted from the original object's weight. The advantage of this scheme is that it avoids loading pointers with indirections; the space overhead is claimed to be roughly equivalent to that of maintaining indirection objects. [11] does not address details such as maintaining separate reference weight tables tables on different nodes.

A key disadvantage of WRC is that if multiple pointers are destroyed at once, the original object's home node may become swamped with decrement messages.

Generational reference counting, proposed in [21], is very similar to weighted reference counting; instead of maintaining an individual weight, however, an object keeps a ledger counting outstanding pointers of each of several generations, and a pointer keeps track of its own generation, and the number of children that have been copied from it. When a pointer is destroyed, a message is sent to its target object which decrements the ledger entry for its generation, but increments the entry for the next generation by the size of its children-count.

2.2.7 Indirect reference counting

Two problems with weighted reference counting are first, how to handle the case when a pointer runs out of weight to divide amongst further copies; and second, when multiple pointer copies are destroyed simultaneously, the resulting decrement operations can create a message backlog at the node containing the target object.

Indirect reference counting [52] (IRC) avoids the problem by counting up, instead of down. Specifically, when a pointer P is copied from area A to area B, area A increments a reference count associated with P, and area B records the fact that the pointer came from area A. When all copies of P are finally eliminated from B, if B's reference count for P is zero, B sends a message to A decrementing A's reference count for P. Thus, the total reference count of P is represented in a tree whose structure mimics the inter-area diffusion pattern of P; the root of the tree is the area containing P's target, and children point to their parents, rather than the other way around. Note that as long as B still has a copy of P, even if A eliminates all of its copies, it must preserve the reference count entry for P as long as it is nonzero. Area-local GC operates using any local object with a nonzero reference count as a root.

The communications overhead of IRC is one extra message per inter-area pointer copy – the decrement message sent when an area discovers it no longer has copies of a pointer it has received. The space overhead is the record at each intermediate node in the diffusion tree which notes the local outstanding reference count, and from which area the pointer was originally received.

Object migration is relatively easy with IRC – the object is migrated to its new location, and the root of the IRC tree is made an internal node, with the new root as its parent.

In the worst case, a single remote area holding a copy of P may lock in place an arbitrarily long chain of records through other remote areas. [49] presents a scheme similar to IRC, but which eliminates these chains via diffusion-tree reorganization. In this scheme, when an area B is to send a pointer P to area C, where P points into A, the following sequence happens: B increments its local reference count for P; B sends the pointer to C; C sends an "increment-decrement" message to A; A increments its local reference count for P; A sends a decrement message to B; B decrements its local reference count for P. Thus, at the end of the flurry of messages, all diffusion tree leaves have area A as their parent – and since area A is where P's target object lives, they need not explicitly record their parent area.

This scheme thus has two advantages: first, it avoids long chains of records through areas that otherwise have no copies of a pointer; second, records may be smaller since they don't need to record a parent area distinct from the pointer's target area. The disadvantage, however, is that the communications overhead for a pointer copy is now three messages instead of one: the "increment-decrement" message to the home area, the "decrement" message to the sending area, and then eventually the "decrement" message to the home area. Only the last of these is needed in straight IRC.

The garbage collection algorithm presented in Chapter 5 of this thesis builds on IRC; a more thorough review of IRC is included in Appendix A.

2.2.8 Grouped garbage collection

Weighted or indirect reference counting alone does not collect of inter-area garbage cycles. One approach to solving this problem suggested in [7], [33] and [54] is to dynamically form groups of areas, and garbage collect them together. [33] describes how to do such grouping with a mark-sweep algorithm on top of distributed reference counting; [54] extends the grouping scheme to work with indirect reference counting. Groups may be dynamically formed, and can exclude failed nodes in order to reclaim garbage cycles which do not pass through failed areas.

2.2.9 Other indirect GC schemes

As pointed out in [54], nearly any GC algorithm can be modified to be an indirect algorithm by altering it to traverse parent pointers in the IRC tree instead of directly following remote pointers to their targets.

For instance, indirect mark and sweep [53] builds on top of the IRC diffusion tree by propagating marks normally within an area, but when marking from a remote pointer, following the parent link in the remote pointer's copy-tree entry. When the marking phase is completed, any object or copy-tree entry which hasn't been marked may be eliminated in the sweep phase (note that sweeping does need to keep reference counts consistent in surviving portions of IRC trees.)

Another indirect scheme is indirect reference listing (IRL) [55], in which counters in an IRC tree are replaced by explicit entries, one for each area to which a pointer has been copied. IRL is somewhat more resistant to individual node failures than IRC, although it is still not resistant to message loss or duplication.

The SSP chains scheme [63] is similar to indirect reference listing, but includes a timestamping protocol which adds tolerance for duplicated, lost, delayed, and reordered messages, as well as individual node crashes. [35] extends SSP chains with a timestamp propagation scheme inspired by [26] that eventually eliminates inter-area cycles.

Chapter 3

Sparsely Faceted Arrays

3.1 Introduction

In this chapter I introduce Sparsely Faceted Arrays (SFAs), which are in many ways the centerpiece of this thesis. Sparsely faceted arrays are a novel shared-memory mechanism for implementing distributed objects on NUMA-DSMs. Implemented with simple hardware support, they are extremely low overhead, as they enable shared memory access at full hardware speeds.

In section I will discuss the gap, filled by SFAs, that lies between existing hardware support for distributed objects and the type of distributed objects that are desirable for certain algorithms and data structures. In Section 3.2, I introduce sparsely faceted arrays. In Section 3.3, I discuss several application of SFAs, with detailed explanation of their use in quicksort and Kd trees. In Section 3.4, I address the details of implementing SFAs at the hardware level, supporting the discussion with simulation results. Finally, in Section 3.5, I conclude by summarizing the important characteristics of SFAs.

Although sparsely faceted arrays are amenable to garbage collection, I reserve discussion of SFA memory management for Chapter 5, which is dedicated to the topic.

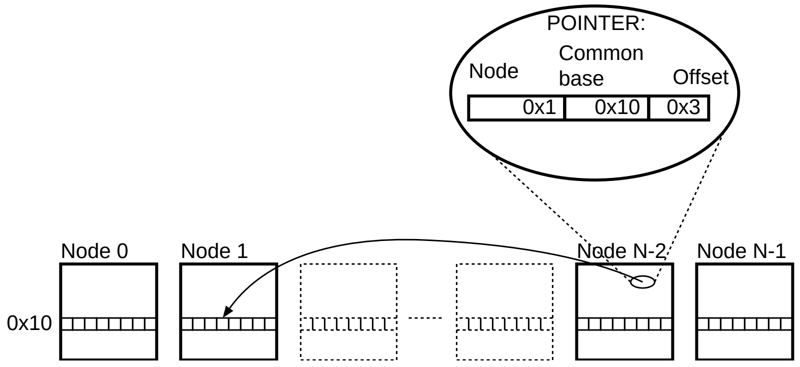


Figure 3-1: A global, uniformly-distributed array with a facet on every node, and a pointer to element 3 on node 1. Note that the array has been assigned the same base address on every node.

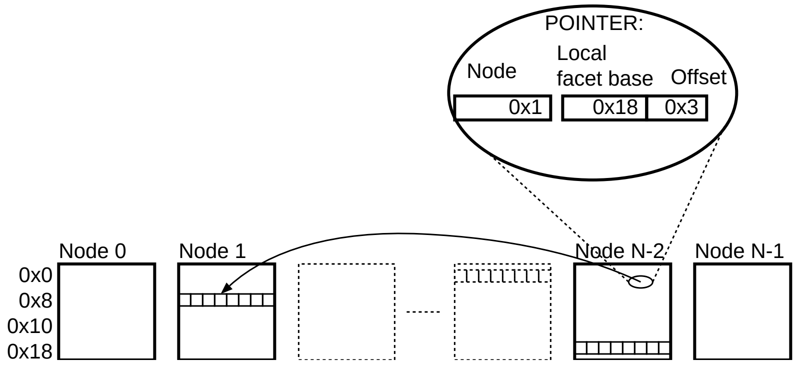


Figure 3-2: A sparsely faceted array with facets on a subset of nodes, and a pointer to element 3 on node 1. Note that within a node, pointers to the SFA on other nodes are represented in terms of the local facet address.

3.2 An introduction to Sparsely Faceted Arrays

A conventional globally-distributed array has a region of storage allocated on every node in a system. I will refer to the per-node regions of a distributed array as facets, in order to differentiate them from regions of storage allocated to scalar (non-distributed) objects. A typical distributed array is shown in Figure 3-1; the per-node blocks of memory for such an array are generally located at identical locations on every node.

A sparsely faceted array is essentially a virtual, globally-distributed array. Although it has virtual facets on every node, actual facets are allocated lazily; in other words, an SFA has a sparse set of actual facets.

Because an SFA may be used over arbitrary subsets of nodes, the abstraction presented is explicitly two-dimensional: the first dimension is the node; the second is the offset within a facet on the node. For example, if S is an SFA, `S[1][3]` refers to the third word of the facet on node 5. See Figure 3-2.

Since an SFA's facets are allocated lazily, there is no guarantee that they will be allocated at identical locations on different nodes. Within a node, a pointer to an SFA is always represented in terms of the node's local facet address; a pointer to an SFA may be dereferenced just like a pointer to a scalar object, and therefore a thread which accesses a local facet suffers no indirection overhead. Again, see see Figure 3-2.

A translation table at the network interface converts from local facet address to an SFA's globally unique identifier (GUID) and back. The details of implementing the translation mechanism are discussed in Section 3.4.

3.3 Example applications

3.3.1 Data structures over all nodes

There are a wide variety of data structures that evenly distribute information over the entire machine. SFAs can be used to implement any data structure that could be implemented with a more traditional globally-distributed array mechanism; of course, such implementations do not make take full advantage of SFAs.

Vectors

One common class of parallel data structures is the class of vectors and arrays distributed over all nodes; these primitives are found in data-parallel SIMD-style languages such as *Lisp [42], NESL [8], and HPFortran [30]. A vector with K words per node over N nodes is represented with an SFA with C-word facets. Translating a simple vector reference to an SFA reference is straightforward: a reference `V[i]`, where V is a vector backed by SFA S, translates to `S[i/C][i%C]`. (HPFortran allows multidimensional arrays with varying alignment to be distributed evenly over all nodes; the reference translations from such an array to an SFA will be correspondingly more sophisticated, but still quite straightforward.) References can be made in parallel by any and all nodes with a reference to the vector.

Hashtables

An SFA can serve as the basis for a distributed hashtable. Each slot of the SFA stores a (possibly empty) linked-list of key/value pairs. To perform a lookup key K in a hashtable H using hash function hash, and which is backed by SFA S with facet size C, we find the appropriate slot in the hashtable much as we found a vector index above: `S[hash(K)/C][hash(K)%C]`. Again, references can be made in parallel by any and all nodes with a reference to the hashtable.

Tables

Database-style tables are another important type of data structures. By distributing each table over the entire set of nodes, it becomes possible to search all of the entries in the table with maximal parallelism. The mapping between SFA and table is sufficiently straightforward as to merit no further discussion here.

Array-of-queues

A distributed array of queues, with one queue on every node, is an important data structure. It can serve as the basis for a message-passing layer, or for a work-stealing mechanism.

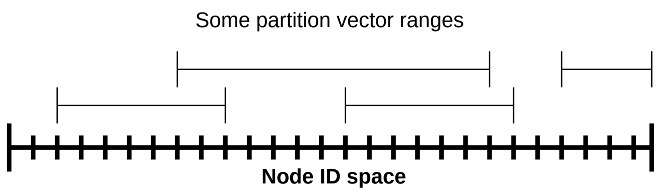


Figure 3-3: Some example partition-vector ranges over the range of node IDs.

Such a structure is quite easy to implement with an SFA: each facet consists of two words containing pointers to the head and tail of the queue on that node.

3.3.2 Message-passing between arbitrary sets of processors

The Message Passing Interface standard [17, 18] provides an interface allowing processes to form themselves into groups which may be arbitrary subsets of the complete set of processes. Groups may generate “communicator” objects in order to perform intra-group communications.

Sparsely faceted arrays provide a natural mechanism for implementing communicator objects. In essence, a communicator can be represented by an SFA-based array-of-queues as discussed above. A processor sends a message to another processor by inserting the message into its target’s local queue using synchronized shared-memory operations.

This is a more interesting application than the global array of queues because the communicator for a group which does not include all nodes can be sparsely faceted. Since MPI does not allow communicators to be shared with processes outside of a group, the name of an SFA underlying a communicator will never be passed to nodes outside of the group, and thus facets will only be allocated on those nodes that are in the group.

3.3.3 Partition-vectors and Quicksort

A partition-vector, or pvector, is much like the distributed vectors described above. However, rather than being striped over the entire machine, a pvector may store its data over a contiguous subset of nodes; in other words, if we treat the node identifiers as representing as a one-dimensional space, a partition-vector stores data over a line segment in that space. See Figure 3-3.

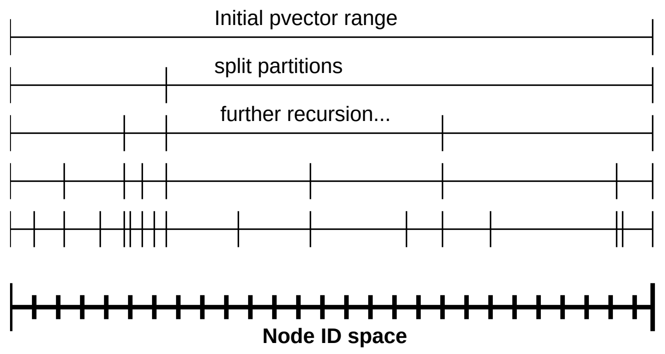


Figure 3-4: Linear partitioning due to quicksort; each node must store $\mathrm{O}(\log(N))$ pvectors for a length N array.

A partition vector is represented by an immutable structure containing (at least) three pieces of information: its base node, the number of nodes over which it is striped, and the SFA providing its backing store.

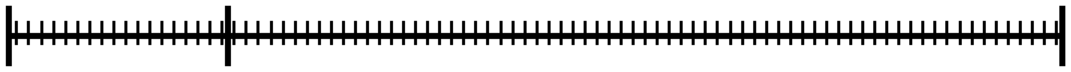
Note that while a pvector only stores data on nodes in its node span, if a pvector-structure is stored on a node outside of the span, a facet for the pvector's SFA will be allocated on that node anyhow. Thus, it behooves the programmer to avoid widespread distribution of pointers to "small" pvectors.

Conventional data-parallel operations including reduce, scan, map, etc., can all be implemented straightforwardly for pvectors. Using these operations, divide-and-conquer algorithms such as quicksort can be implemented using pvectors.

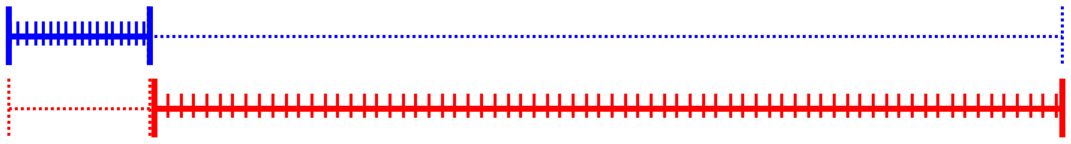
Quicksort

Quicksort benefits from the use of pvectors and careful partitioning of recursive quicksort invocations. The initial invocation of quicksort partitions the input into three sets: less than, equal to, and greater than an arbitrarily-selected pivot element; the less-than and greater-than sets are recursively sorted with quicksort, and these partial results concatenated with the equal-to set to form the final, sorted result.

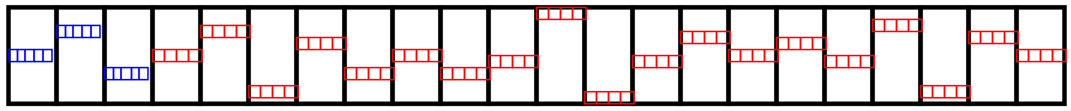
To exploit locality of reference, the recursive quicksort invocations may performed over independent partitions of the machine, rather than spreading each sub-problem more thinly over the whole machine. Thus, with each sub-problem, the less-than set is allocated to a pvector over a subpartition of current problem's partition, and the greater-than to a pvector



(a) Initial partition sizes



(b) Sub-partitions in SFAs — virtually everywhere...



(c) ...but actually allocated only where used.

Figure 3-5: Non-linear Quicksort partitioning using SFAs. Note that the left SFA is denser than the right SFA, i.e. its facets are larger.

over the remainder of the current partition. See Figure 3-4.

Note that since a linear partitioning strategy results in an even density of elements distributed over the processor array, it is possible to express this type of quicksort in terms of operations on global, uniformly distributed vectors, albeit at the cost of unnecessary synchronization between independent subproblems. This strategy is used by NESL [8], which converts recursively data parallel algorithms into operations on global vectors.

Unnecessary synchronization is not the only cost of such a strategy, however; partitioning the processor array linearly with respect to the size of the quicksort subproblems turns out to be suboptimal because it does not balance the expected work.

Expected work of Quicksort

The expected work of quicksorting N numbers is O$(N \log N)$. Consider an initial split which produces subproblems with a ratio of sizes of 4:1. (For this analysis, let us assume that the equal-to set is small enough to be negligible.) The expected work due to the larger subproblem will be

$$\frac{4N}{5}\log\frac{4N}{5}$$

while the expected work due to the smaller subproblem will be

$$\frac{N}{5}\log\frac{N}{5}$$

.

The expected work ratio R of the two subproblems will therefore be

$$R = \frac{\frac{4N}{5}\log\frac{4N}{5}}{\frac{N}{5}\log\frac{N}{5}} = 4 + \frac{8}{\log\frac{N}{5}}$$

For $N = 1280$, $R = 5$; if we increase N significantly to $N = 330,000$, $R = 4.5$. In other words, even for large N, there is a significant difference between the subproblem size ratio and the subproblem work ratio. This difference is magnified by decreasing the overall problem size, but even very large problems can exhibit a significant difference.

This analysis suggests that the processing resources should be partitioned according to the expected work ratio of the subproblems, rather than than according to the subproblem sizes. This will have the effect of spreading the elements of the larger subproblem more thinly, while placing those of the smaller subproblem more densely. See Figure 3-5. Note that in order to pack each subproblem efficiently, it is necessary to use different SFAs, with different facet sizes.

Simulation results

The benefits of partitioning processors by work-ratio rather than by size-ratio are borne out by simulations of an idealized architecture using the Mesarthim simulator described in Chapter 6.

Figure 3-6 shows the ratio of total instructions executed in performing quicksort with work-ratio partitioning vs. the total instructions executed in performing quicksort with size-ratio partitioning. The total number of instructions executed in solving the problem are a rough indication of the total amount of work involved in solving the problem; as

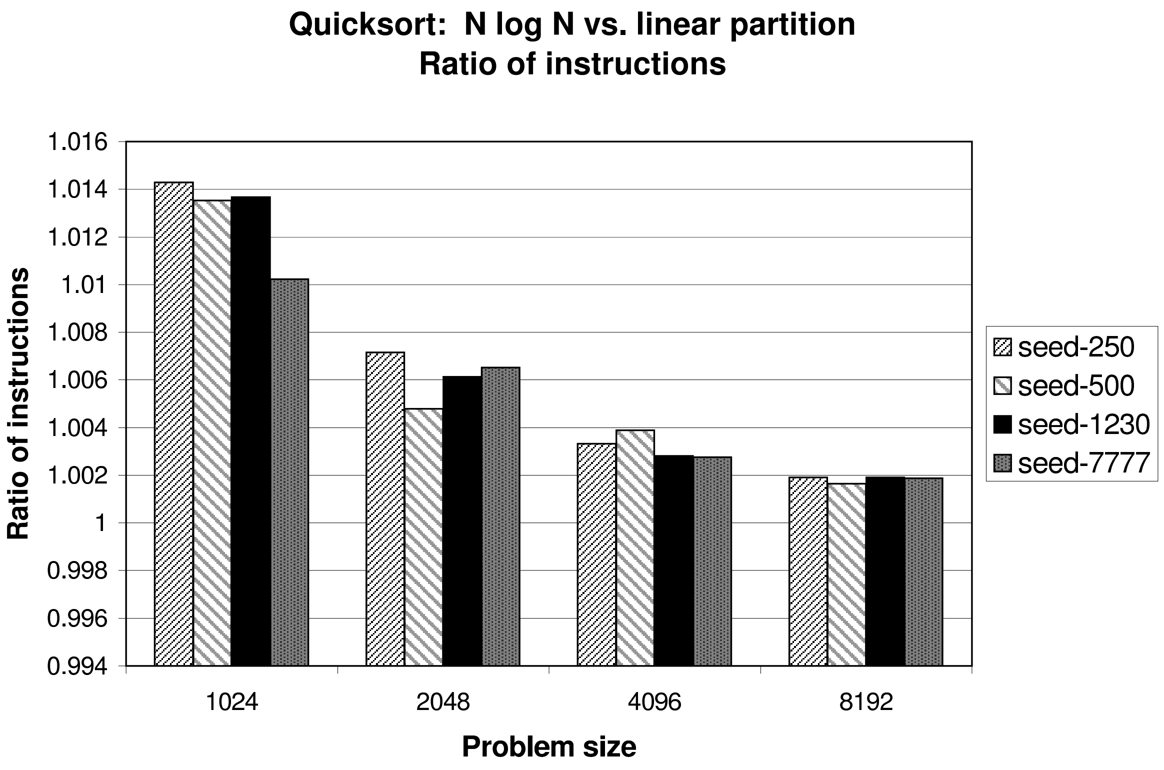


Figure 3-6: Quicksort: Ratio of instructions executed by quicksort with work-ratio partitioning to instructions executed by quicksort with size-ratio partitioning, on 256 nodes; ratios for four different problem sizes, and four different random seeds, are shown.

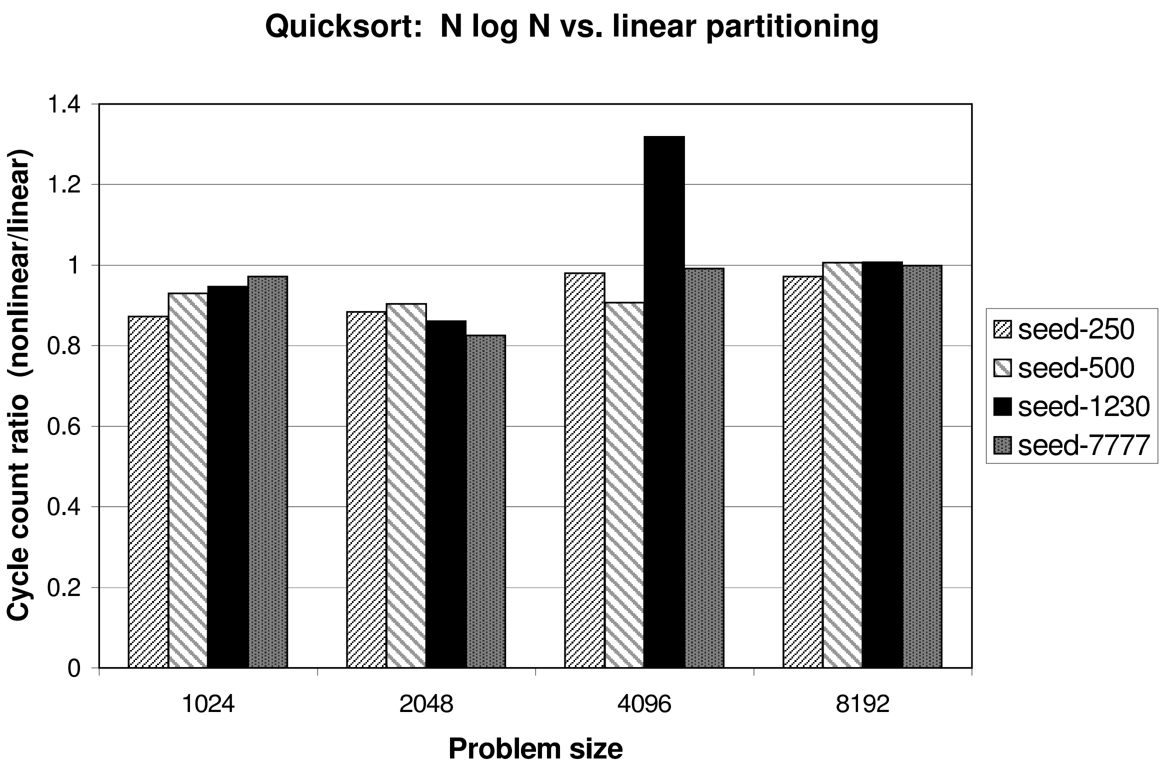


Figure 3-7: Quicksort: Ratio of cycles needed to complete quicksort with work-ratio partitioning to cycles needed to complete quicksort with size-ratio partitioning, on 256 nodes; ratios for four different problem sizes, and four different random seeds, are shown.

the figure makes clear, the total work performed by both versions of quicksort is about the same, with the work-ratio partitioning strategy performing a little extra work, primarily due to the more complex partitioning calculations.

Figure 3-7 shows the ratios of the total number of cycles needed to complete quicksort with work-ratio partitioning vs. the total number needed to complete quicksort with size-ratio partitioning. For the smaller problem sizes, speedups of up to 17.4% are observed; for larger problems, the impact is not generally as great. An outlier at 4096 elements is a reminder that quicksort's performance is dependent upon the selection of splitting values, and that a bad series of choices can have a significant impact. Since we can see from Figure 3-7 that, for the problem in question, work-ratio partitioning quicksort does not perform significantly more work than size-ratio partitioning quicksort, we may conclude that the trouble in this case is a series of bad splits which conspire to distribute the actual workload unevenly in the case of work-ratio partitioning.

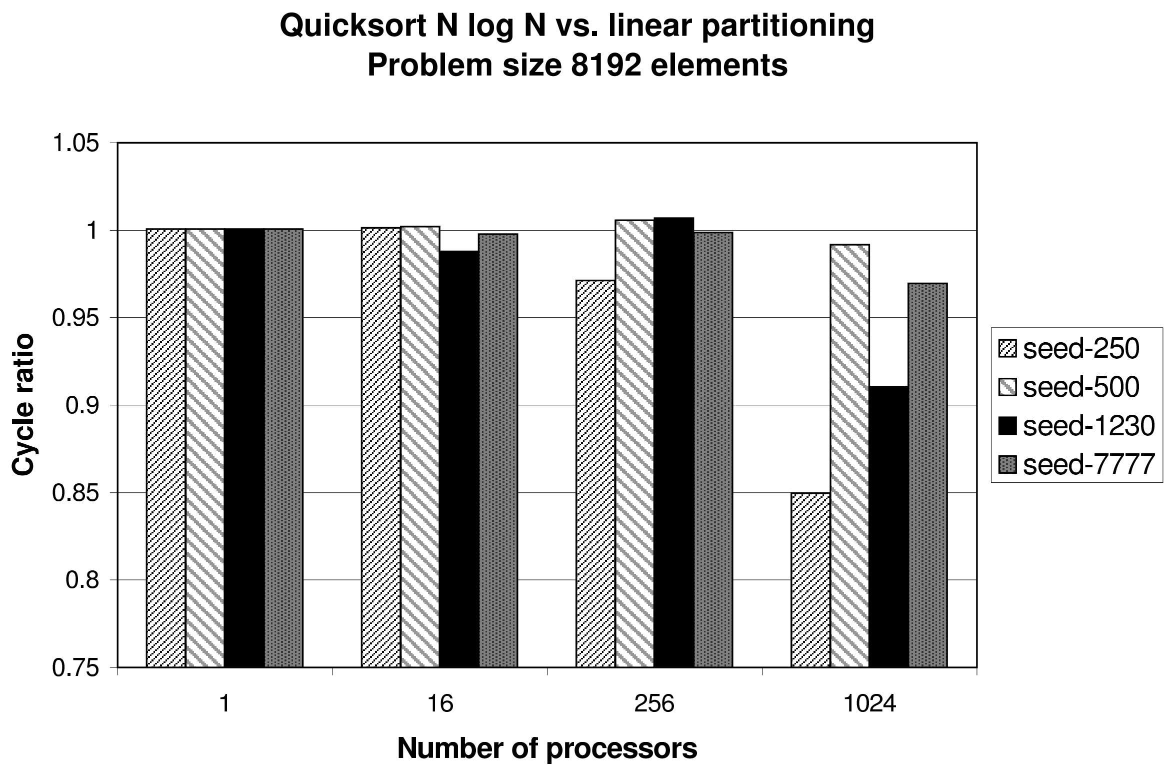


Figure 3-8: Quicksort: Ratio of cycles needed to complete quicksort with work-ratio partitioning to cycles needed to complete quicksort with size-ratio partitioning; ratios are shown for sorting 8192 numbers, generated with four different random seeds, on varying numbers of processors.

Figure 3-7 shows that the importance of work-ratio splitting goes down as problem size grows. By contrast, Figure 3-8 shows that the importance of work-ratio splitting goes up as the number of processors grows, relative to a fixed problem size. As more parallelism becomes available for use, the impact of bad work balancing increases.

3.3.4 Kd Trees

For many problems, the one-dimensional model of pvectors is a natural representation, but there are problems for which an awareness of higher dimensionality is advantageous. One example is the Kd tree [13]. A Kd tree partitions a set of points or objects in N-dimensional space (where N is typically two or three) such that each leaf of the tree contains only a few items.

An application of Kd trees

One potential application of Kd trees is in model registration in medical imaging [34]. In this application, sensors provide real measurements of a feature of interest, e.g. a skull. The goal of model registration is to determine the geometric relationship between the sensor data and a pre-surgical model; discovering this relationship enables, for instance, the real-time overlay of pre-surgical planning images on top of images of the actual surgery.

The approach taken in [34] is to build an octtree [13] of a model surface, and to then use an iterative closest points algorithm [5] refine an initial alignment estimate to within a very small error. A kd tree is a close relative of an octtree, but is more amenable to spatially-aware, recursive, parallel, non-uniform partitioning; thus, the remainder of this discussion will assume the use of a kd tree in place of an octtree.

In the iterative closest points algorithm, each sensor-provided sample data point is translated and rotated according to the current alignment estimate, then the Kd tree is used to find the closest point on the model-surface to the transformed sample point. The per-point distances are used to refine the alignment guess, and the process is repeated until the alignment error is sufficiently low.

The role of the Kd tree in this algorithm is in finding the closest surface-point to a

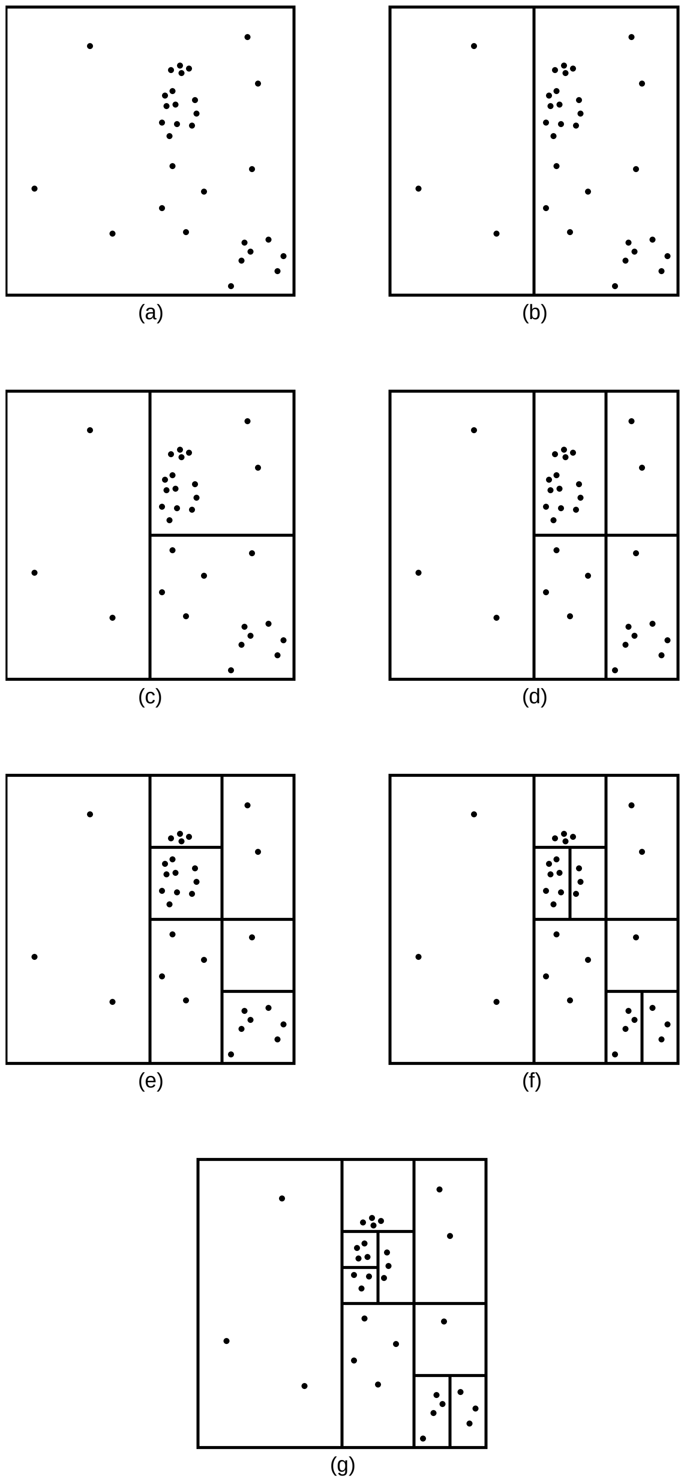


Figure 3-9: Constructing a Kd tree partition of points in 2-D. The initial point-set is shown in (a). Recursive partitioning in steps (b)-(f) lead to the final partition shown in (g).

sample point. This may involve searching multiple Kd tree leaves. The initial traversal of the tree simply finds the leaf whose volume contains sample point's coordinates; the minimum distance between the sample point and any of the model points in the leaf volume is computed by brute force. If, however, the distance from the sample point to the nearest model point is greater than the distance from the sample point to one or more boundary of the leaf's volume, it is possible that there might be a point in one of the leaf's neighbor's volumes containing a point closer to the sample. Thus, a given sample point may need to be compared against the model points in a number of neighboring leaves.

Basic Kd tree generation

An algorithm for generating a Kd tree over a volume V and a point set P such that each leaf has fewer than K points is as follows:

KD-build(P):

1. If $|P| < K$ return a leaf node containing P.

2. Otherwise, pick a dimension D. V has a certain length on D; partition the data around the midpoint M of that dimension into two sub-volumes V_L and V_R, where V_L gets all points whose position along the dimension is less than or equal to M, and V_R the rest.

3. Recursively apply KD-build to V_L and V_R, rotating through splitting dimensions with each layer of recursion.

4. Return an interior node containing D, M, and the two sub-trees returned in the recursive builds of the previous step.

An example partitioning is shown in Figure 3-9.

Distributed, partitioned Kd trees

The purpose of building a distributed Kd tree is to be able to compute the closest-points for an entire set of sample-points in parallel; the strategy is to replicate the upper nodes of the tree to enable parallel access. I will assume that the processor node ID encodes the same

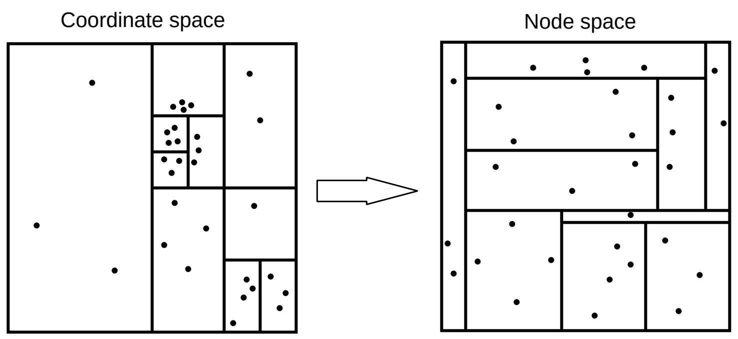


Figure 3-10: A Kd tree in 2 dimensions. Regions containing points are leaves; T-intersections are inner nodes. Note the relative uniformity of points-distribution in nodespace in spite of their asymmetric distribution in coordinate space.

number of physical dimensions of the machine as the number of dimensions of the volume the Kd tree is partitioning.

The scheme I am about to describe for representing a distributed Kd tree is based on the following assumptions:

1. The sample points are roughly evenly distributed over the model.

2. The initial alignment estimate is pretty close. (Typically a human provides the initial alignment estimate for medical image alignment.)

If these assumptions hold, roughly equal numbers of sample-points will be compared against model points in each leaf. Thus, we would like each leaf-node to be given an approximately equal portion of the processor array in order to load-balance the final comparisons. Furthermore, since the sampling process for a single sample-point may involve comparing it against several neighboring leaves, we would like leaves representing adjacent volumes to be spatially adjacent in the processor array in order to exploit the locality.

Thus, each time we split a coordinate-space dimension in half, we split the processor array along the same dimension – but not in half. Rather, we split it according to the ratio of the number of points that falls on either side of the partition. This strategy will generate a fairly even distribution of points over the processor array, while also roughly preserving the spatial relationships between adjacent nodes. An example in two dimensions is shown in Figure 3-10.

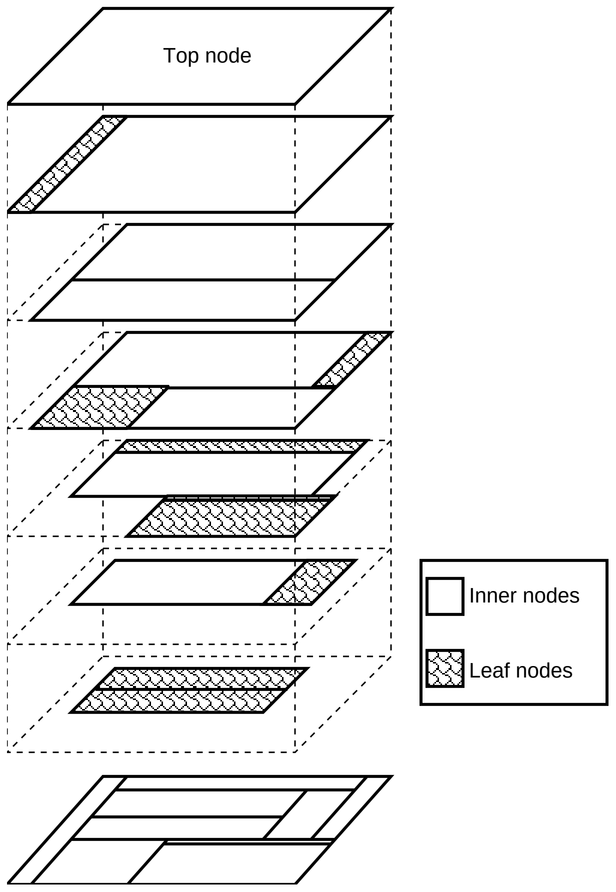


Figure 3-11: The replication of the nodes of the Kd tree from Figure 3-10. Figure is in nodespace rather than coordinate space.

Of course, we distribute each leaf over its N-dimensional partition simply by allocating an SFA, and using it as an N-dimensional array over the partition storing the leaf's points.

In addition to spatially distributing the leaves of the Kd tree, we must replicate the inner nodes of the tree over their partitions to enable parallel traversal, i.e. the topmost-node is replicated over the entire machine, its children are each replicated over their partitions, and so forth. See Figure 3-11 for a visualization. Once again, we resort to SFAs to implement the replicated nodes, with the replicated data contained in the facets within each node's partition.

Thus, each leaf node and each inner node is represented with an immutable structure defining its partition/volume and identifying its SFA. Absent degenerate model-point distributions, each node ends up storing data associated with O$(\log N)$ Kd tree nodes.

There is a point of inefficiency here that deserves a brief mention: since each inner node V contains references to the SFAs of both of its children, each processor in V's partition will allocate a facet for both children's SFAs, although usually only one child will actually store replicated data on the parent node. This will cause excess allocation by roughly a factor of two, although the bound remains the same at O$(\log N)$.

SFA-based, replicated distributed objects are similar to the distributed objects of Concurrent Smalltalk [24, 23]. There is a key distinction, however: the SFA-based placement respects the spatial properties of the problem being represented; by contrast, distributed objects in CST as provided by the COSMOS [25] operating system will be evenly distributed over the entire machine, thus losing the advantages of locality of subproblems.

The importance of being sparse

The importance of the sparsity of the SFA representation can be seen by considering the case in which the point distribution is moderately even. Assume that on an N-processor machine, each SFA leaf is distributed over an average number of processors M where M is fairly small, and that SFA leaves are non-overlapping.[1] Then there are N/M leaf nodes,

[1] In practice, since we have integral numbers of nodes in each dimension, SFA leaves will sometimes wind up overlapping at the edges. This only increases the amount of per-processor data storage by a constant factor in the worst case.

and nearly that many inner nodes; since I specified a small M, this is essentially O(N) Kd tree nodes.

If sparse arrays were not available, and storage for each node was simply allocated across every processor (even if not used on most), the per-node storage requirement for the Kd tree would be increased by O$(\frac{N}{\log N})$.

Alternatively, in the absence of distributed arrays of any sort, we could construct fan-out data structures to represent each replicated node.... but then the root of each fan-out tree itself becomes an un-replicated bottleneck!

Of course it is ultimately possible to build any structure with enough software, but SFAs make the task simple and the implementation efficient.

Computing closest-points

The process for computing closest-points in parallel thus becomes the following, for each sample point, starting from the root node:

kd-closest-point(V, P):

1. If V is a leaf node, compare P against each of its points and find the closest point. Return the point and the distance to it.

2. Otherwise, compare P against N's split-coordinate, and choose the appropriate sub-node V_L or V_R.

3. If current processor is within the sub-node's partition, recursively call kd-closest-point on the current processor. Otherwise, pick a random processor within the sub-node's partition, and make a remote, recursive call on that processor.

4. If the returned distance is greater than the distance from P to the split-coordinate (i.e. if the returned distance is greater than the magnitude of the difference of P's coordinate in the split dimension and the split coordinate value), then recurse on the other sub-node, and return whichever result had the smaller distance.

5. Otherwise, simply return the result from the recursive call.

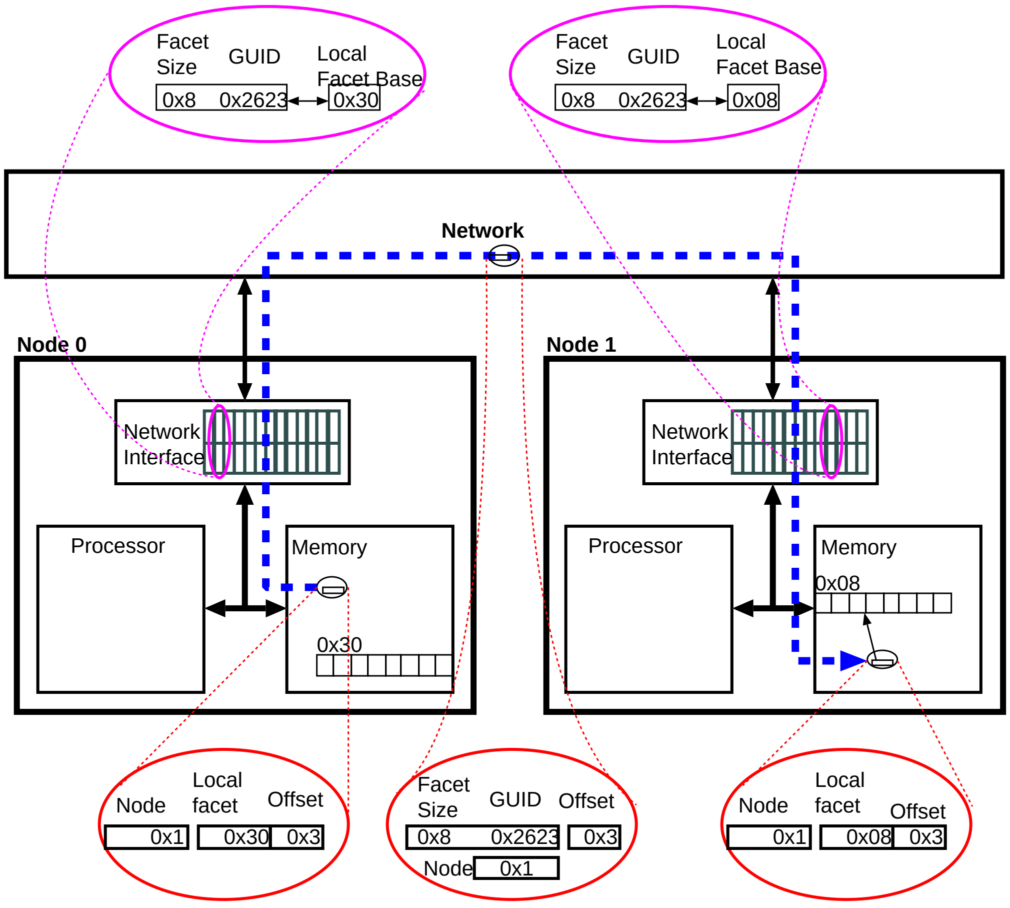


Figure 3-12: A pointer to an SFA is sent from node X to node Y. (a) shows the pointer representation on node X; (b) shows the same pointer in the network; (c) shows the pointer on Node Y. (d) and (e) show the network interface translation entries for the SFA in nodes X and Y, respectively.

The reasons for executing recursive calls on processors in the child-nodes' partitions, rather than simply reading their data structures remotely, are twofold. First, each reference to a remote SFA read from a foreign partition may cause another facet allocation on the local node; although it is possible for garbage collection to later reclaim this unused facet (see Section 5.4) this is inefficient. More importantly, however, is simply that it is advantageous to move computation closer to the data it is working on, as this reduces latency and network load.

3.4 Implementation

Although an SFA has an virtual presence on every node in a system, when an SFA is first created it only requires memory on the creating node. Thus, the initial creation of an SFA is a purely local operation, requiring no inter-node communication or synchronization. The node which first creates an SFA is known as the SFA's home node; other nodes are remote nodes with respect to the SFA.

An SFA pointer is represented differently on different nodes, and in the network. Figure 3-12, which shows the different representations and the translation points between them, provides a reference for the following discussion.

An SFA's home node must create a globally unique identifier (GUID) for the SFA when a pointer to the SFA is first sent to a remote node. A remote node allocates a facet to an SFA when it first encounters that SFA's GUID; to enable this allocation, GUIDs must include facet size information.

Within a node, an SFA is named by its local facet's address; thus, pointers to SFAs may be dereferenced just like pointers to scalar objects, and a thread which accesses a local facet suffers no indirection overhead.

Since a node allocates new storage to each SFA encountered, there is always a one-to-one mapping between a facet address and the corresponding SFA GUID. Every node maintains a translation table with such a mapping for each local facet. As shown in Figure 3-12, an SFA pointer sent between nodes is translated from its local name (facet address) on the source node to its global name (GUID); the global name is transmitted over the network; and upon receipt, the global name is translated to its local name (facet address) on the destination node.

Obviously, the heart of an SFA implementation lies in the translation tables.

3.4.1 Translation table implementation

The signature property of a shared memory architecture is the ability for one node to reference another's memory without interrupting its processor. In such architectures, the node network interface is capable of directly accessing node memory in order to respond to remote requests.

To maintain this signature property in the presence of sparsely faceted arrays, the network interface (NI) must be able to perform SFA translations independently; this implies low-level support for SFAs — although whether or not it implies “hardware” support depends on the programmability of the NI in question.

In principle, a node’s translation table is stored in the network interface; this enables the NI to translate incoming and outgoing references without interrupting the processor. In practice, the table may grow too large to fit in an NI-specific hardware table. Since the network interface in a shared-memory machine has direct access to a node’s memory, one approach to implementing a GUID/facet translation table is to store it as a hashtable in the node’s physical memory. In general, this approach is quite straightforward, but there are a few details which merit specific discussion.

Incoming translations

When an SFA’s GUID is first received by a node, the node will have to allocate a facet for the SFA, and create a GUID/facet translation entry in the translation table.

One perfectly reasonable strategy for handling this case is to interrupt the processor and handle it in software. However, if the process of allocation is made sufficiently simple, it is reasonable to consider letting the network interface perform the allocation of facet and translation entry.

In a system featuring compacting garbage collection, allocation is as simple as advancing a pointer: each new object is allocated immediately after the previous one. In such a system, the network interface could be assigned its own allocation region; the processor would only need to be interrupted when the region ran out of space, or when the hashtable requires resizing.

Hashtable resizing will disable SFA translation services for the duration of the operation; its frequency should therefore be minimized. In practice, it would be to a system’s advantage to synchronize hashtable resizing between all nodes so that individual nodes did not randomly stall the entire computation by becoming unresponsive; [68] demonstrates that independent per-node garbage collection operations should be performed synchronously for the same reason.

Outgoing translations

A translation entry from facet-address to GUID is keyed on the base address of the facet. If pointer math is allowed, a pointer being sent out of a node could refer to an interior word of a facet, rather than the base. In order to make translation work in the presence of pointer math, the NI must be able to determine the base address of a facet from a pointer to an interior word.

My recommended strategy, employed in the Mesarthim simulator, is to use a guarded pointer [9] format from which it is simple to determine the base of the segment of memory the pointer denotes. See Section 6.2.2 for details of Mesarthim's guarded pointer format.

An SFA must be assigned a GUID when a pointer to its first facet is sent out of its home node. To enable other nodes to allocate facets for the SFA, the GUID must embed the facet size. If the network interface is to be responsible for generating a GUID when it discovers that no translation entry exists for a particular facet address, it needs to be able to determine facet size, given a pointer to the facet.

Once again, my recommended strategy is to use a guarded pointer format; the data which makes it possible to determine the base of a memory segment from a guarded pointer into the middle of the segment also encodes the segment's size. An alternate strategy, in the absence of pointer math, would be to store an object's size in its first or last word; the network interface could then look up the object's size on demand, at the cost of additional latency and memory bandwidth consumption.

3.4.2 Relationship to hashtables

The translation tables underlying sparsely faceted arrays are much like system-maintained, communal hashtables. However, they have several unique properties which differentiate them from regular hashtables.

- The keys – local addresses and GUIDs – are system-generated, unique values.

- Because keys are unique, the system can detect when no copies of a key remain.

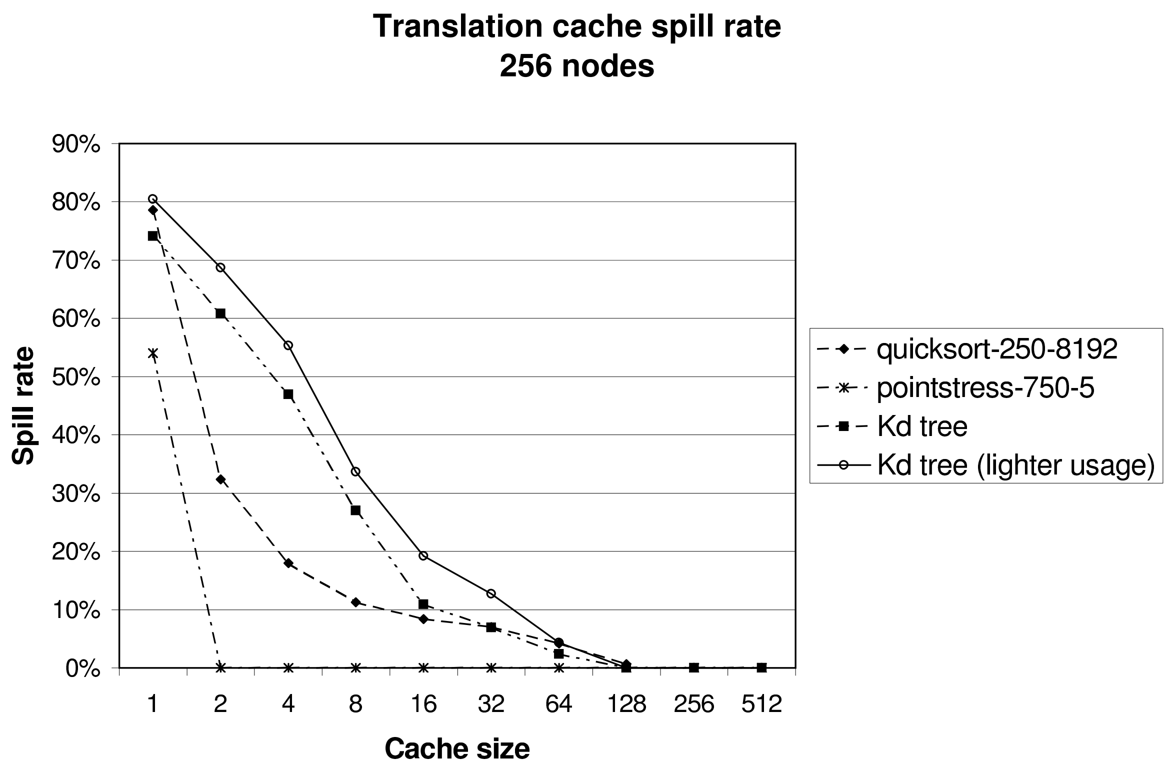


Figure 3-13: Translation cache spill rate for several applications and cache sizes.

- Each table is directly accessed by the network interface hardware.

The first two properties enable SFAs to be garbage collected; a translation-table implementation built on top of normal hashtables, without system support, would be incapable of detecting when an SFA was no longer in use.

The third property is simply related to efficiency: to provide true shared-memory performance, the network interface must be able to perform translations without interrupting the main processor.

3.4.3 Additional hardware support: a translation cache

There are two drawbacks to maintaining the facet/GUID translation table in a node's main memory. The first is that the act of translation will consume valuable memory bandwidth. The second is that the latency of accessing information in main memory may be several cycles; latency in performing translations inherently delays the delivery of messages.

Both of these drawbacks may be addressed by a translation-entry cache at the network

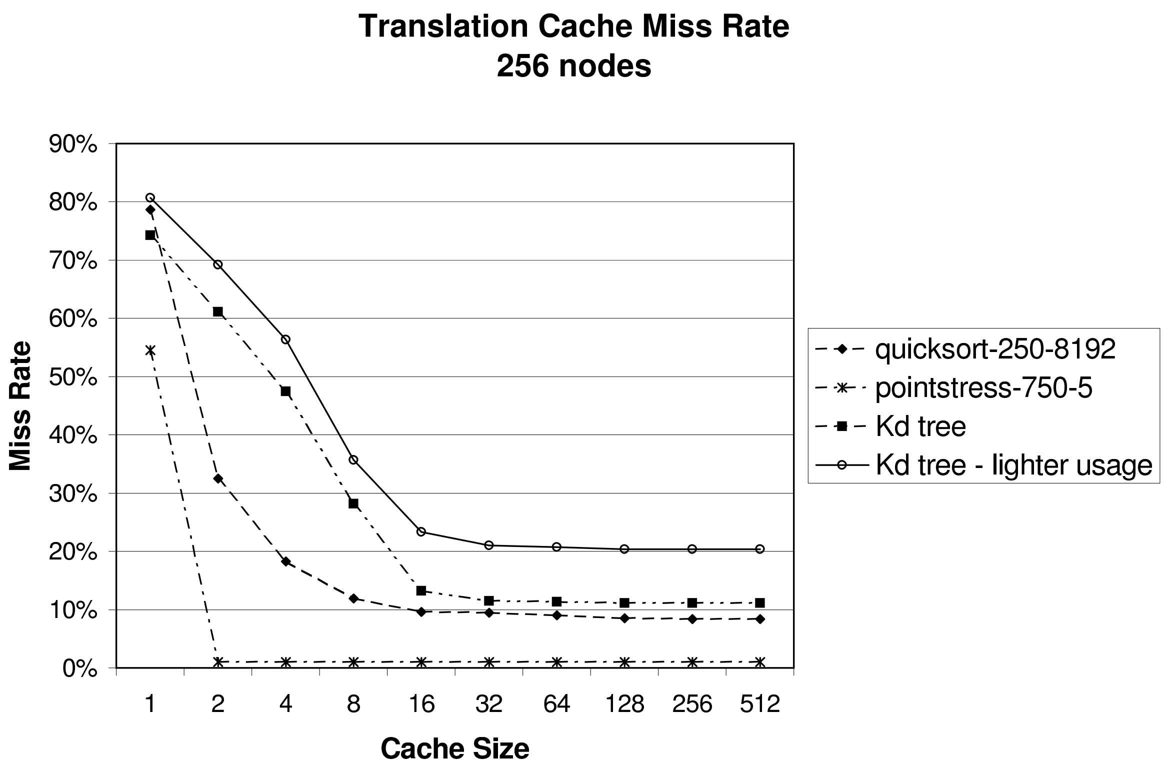


Figure 3-14: Translation cache miss rate for several applications and cache sizes.

interface. To study the efficacy of such caches, I ran a handful of benchmark programs under the Mesarthim simulator described in Chapter 6. The results should be taken not as definitive — the benchmarks are tiny, the simulator idealized — but rather as indicative of the likelihood that a reasonably-sized cache could significantly ameliorate these drawbacks.

In these studies, a single cache is used to perform both incoming and outgoing translations. Each cache is fully associative. The simulated machine has 256 nodes.

I present results for three applications, representing three categories of SFA usage.

Pointer-stressmark is a benchmark from the Data Intensive Systems "stressmark" suite [2]. Pointer-stressmark allocates a single SFA, fills it with random indices[2], and then starts a number of threads; each thread accesses a small region of values in the SFA centered around some index, computes a new index from the values, and repeats the process until a specific value is found, or until it has repeated the process a certain number of times. In terms of SFA usage, pointer-stressmark represents a minimalist extreme: it allocates a

[2]My implementation of pointer-stressmark is not entirely faithful to the formal specification, particularly with respect to the means of generating the initial random numbers.

single SFA, then pounds on it for the rest of its run.

Quicksort stands at the other extreme. This implementation is the recursively partitioning, work-ratio quicksort described in Section 3.3.3; it allocates new SFAs for every subproblem, uses them just long enough to solve the subproblem, then discards them.

The Kd tree benchmark fits in the middle, and this is represented by the two different runs. The benchmark first constructs a distributed Kd tree, as described in Section 3.3.4, based on a 3-D volume filled with random “model” points. Once the tree is constructed, an additional set of random “sample” points is generated. The tree is then used to find the closest model point for each sample point; the operation is performed in parallel for all sample points. The two runs construct the same Kd tree over the same set of model points, but differ in the number of sample points that are subsequently searched for in the tree – the “lighter usage” run looks up fewer sample points.

Figure 3-13 shows the translation cache spill rates for these benchmarks and various cache sizes; Figure 3-14 shows the corresponding miss rates. The spill rate is the frequency with which a cache access forces an existing cache entry to be written back to memory; the miss rate is the frequency with which a cache access does not find a translation entry. In a long-running application, of course, these rates would be identical as soon as the cache filled; in applications that run only for a limited time, increasing the cache size decreases the spill rate by preventing the cache from entirely filling.

The spill rates of Figure 3-13 interest us only tangentially: we see that for cache sizes of 128 entries and above, spill frequency is essentially zero, which implies that corresponding misses are not due to cache capacity limitations, but are instead compulsory. Looking at the miss rates in Figure 3-14, we see that they level off around a cache size of 32 entries, experiencing only incremental improvement for larger cache sizes.

The miss rates reflect the different SFA utilization patterns of the benchmark applications. Using only a single SFA, the pointer-stressmark application has a miss rate of less than 1% for caches of two entries or more. Using multiple SFAs for relatively brief periods of time each, Quicksort sees a significantly higher miss rate than pointer-stressmark.

Finally, we see that, as expected, the Kd tree miss rate drops with increasing utilization of the tree after construction, i.e. it drops for the longer-running version that uses the Kd data structure more.

For these applications, the worst miss rate for a cache size of 32 is slightly over 21%. Although studies of larger applications and larger numbers of processors would be necessary to draw a definitive conclusion, it seems likely that most applications using SFAs will hit in the cache at least as often as the lightly utilized Kd tree. Although a hit rate of 79% isn't stellar, it is more than sufficient to decrease the expected translation latency in a practical implementation, as well as significantly reducing the memory bandwidth consumed by the translation process.

I will conclude with a reminder: due to the simplicity of the applications and the idealizations of the simulation framework, these results are indicative, rather than definitive, in suggesting that a reasonably-sized cache can handle the majority of SFA translations. Picking a cache size for a specific architecture will require both a precise simulation of the architecture, and a set of full-scale applications which are representative of the intended workload.

3.5 Summary

In this chapter I have introduced Sparsely Faceted Arrays, a novel, low-overhead, parallel data structure. I have given examples illustrating the use of SFAs in implementing partitioned algorithms and data structures. I have shown how to implement SFAs in practice, and provided simulation results to aid in the evaluation of specialized helper hardware.

SFAs provide several advantages over other methods of implementing distributed objects.

- Allocating an SFA is an asynchronous operation, requiring no inter-node communication; this is unique among distributed object allocation mechanisms.
- An SFA provides a simple programming model: a virtual, globally distributed array.
- An SFA only consumes actual memory on nodes which use the SFA.
- An SFA consumes $O(1)$ space on each node where it has a facet.

- Random access to an SFA's elements operates in $\mathrm{O}(1)$ communications steps; no data structure traversals are necessary.

- Intra-node references to a local facet operate at full memory rate; no indirections are required.

- An SFA which is allocated and used only within a single node consumes no resources beyond those consumed by a scalar object of the same size as the SFA's facet size.

- The data in an SFA is explicitly placed, enabling programs to build data structures that respect the spatial properties of the problem being modeled.

- SFAs are low-overhead, and may therefore be used in large numbers (i.e. they are fine-grained.)

- SFA facets may be densely packed within each node; the per-node translation table even allows local facets to be relocated by node-local garbage collection with no impact on remote references to the SFA.

The cumulative impact of these features is significant: SFA semantics make it simple to implement explicitly placed, non-uniformly distributed objects; SFA performance characteristics make fine-grain use of such objects efficient in terms of both running time and per-node memory consumption.

This chapter has left one important issue unaddressed: beneath the simple abstraction presented to the programmer, SFAs are complex data structures. Memory management with SFAs is inevitably going to be a complex task. This complexity can be hidden from the programmer with an SFA-aware garbage collection system; the next two chapters explore the issues of designing a scalable parallel garbage collector.

Chapter 4

On Garbage Collecting a Massively Parallel Computer

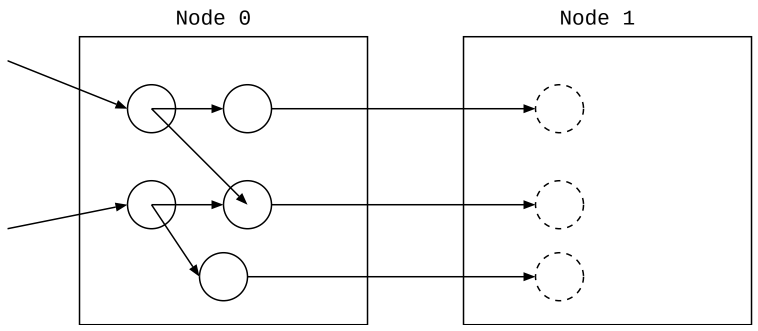
Or, how I learned to stop worrying and love reference counting

In this chapter I address some of the theoretical issues in garbage collecting a massively parallel processor. In particular, I explain why precise techniques such as copying or mark-sweep garbage collection are inappropriate for MPPs, and argue that reference counting does not suffer from the same theoretical performance issues. This argument justifies my use of reference-counting techniques for inter-node GC in the remainder of this thesis.

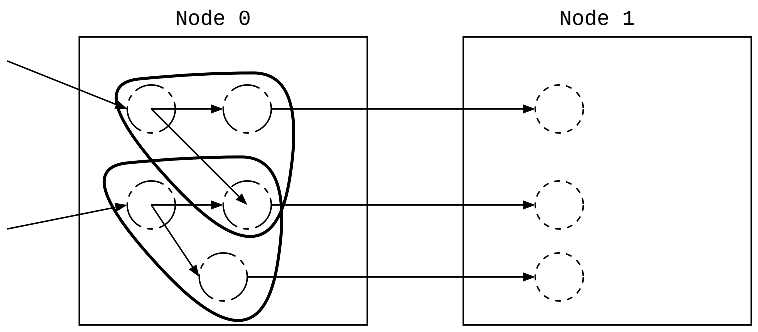
At the end of this chapter, I also describe a novel, parallel garbage collection algorithm which, although unusably conservative, has the ability to reclaim some garbage cycles, and has better theoretical running time than precise techniques on an MPP. I leave open the question of whether there are heuristic approximations to an oracle which could make my algorithm behave in a precise or nearly-precise fashion. Because this algorithm as it stands is impractical, I do not use it elsewhere in this thesis.

4.1 Intra-node and inter-node garbage collection

From the perspective of a single node in any distributed-memory system, memory falls into two basic classes: the node's local, low-latency memory; and everything else, which



(a) Node 0 contains five objects...



(b) ...which compose two meta-objects.

Figure 4-1: Inter-node GC treats these 5 objects as two meta-objects.

is relatively high-latency. This is exactly why most distributed garbage collection systems employ a two-level approach: a node-local garbage collector and an inter-node garbage collector. The inter-node GC typically manages an IN list on each node corresponding to the node-local objects reachable from other nodes, while the intra-node GC maintains an OUT list of objects on other nodes reachable from local objects.

In the remainder of this chapter, I will be speaking almost entirely about inter-node garbage collection, with little or no attention paid to intra-node GC. From the perspective of inter-node GC, a connected group of local objects on one node which are reachable from a single IN reference can be treated as a single meta-object sporting the combined set of OUT edges (see Figure 4-1).

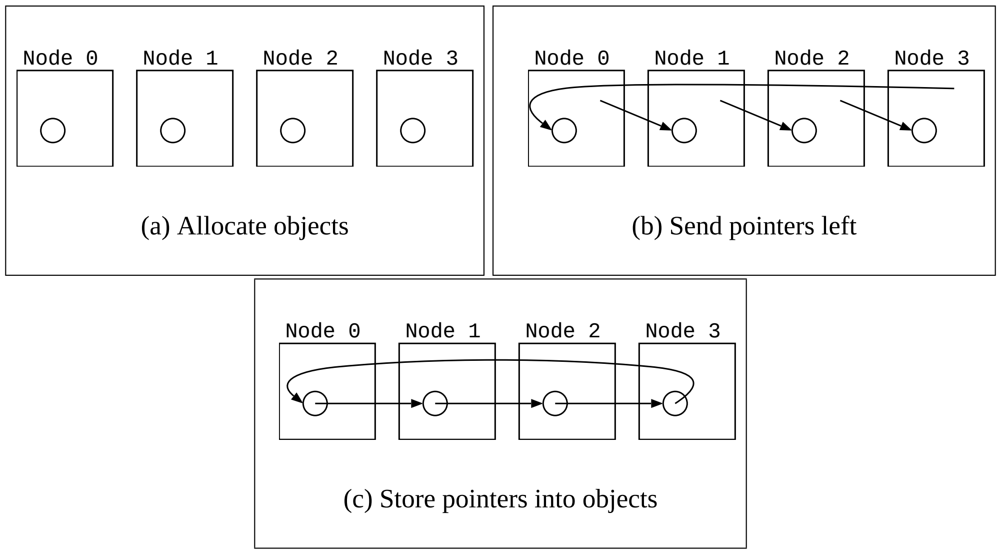


Figure 4-2: Creating O(N)-length data structures in O(1) time.

4.2 Garbage collecting Massively Serial Data Structures

Garbage collecting a massively parallel computer faces a unique challenge: Massively Serial Data Structures (MSDSs). Consider the following case, also shown in Figure 4-2 in which a process on each node performs the following sequence of operations:

1. Allocate a local object.
2. Send a pointer to the new object to the node on the left.
3. Receive a pointer from the node on the right.
4. Store the pointer received from the node on the right into the local object.

On N nodes organized in a logical ring, this sequence creates a circular linked list of length O(N) in O(1) time; every pointer in the list is an inter-node pointer. Obvious variations on this algorithm can generate a data structure of length O(KN) in O(K) time. I call such a structure a massively serial data structure or MSDS.

Now, imagine that after generating a massively serial list, only one node were to retain a direct reference to the list; we will say that the list is singly-rooted.

A singly-rooted MSDS turns out to be a terrible problem for precise, parallel GC schemes.

4.2.1 Precise garbage collection can eliminate effective parallelism

Precise garbage collection techniques must traverse the entire reachable-object graph from its roots in order to differentiate reachable data from garbage. For instance, copying GC traverses the live data graph, copying each live object thus found into a new region of memory. Similarly, mark-sweep GC traverses the live data graph, marking each live object found; after the conclusion of the mark phase, a sweep phase traverses all objects, freeing those that are unmarked.

Because they must traverse the entire live data graph, given a graph of depth L, such precise GC techniques require O(L) time to complete a single pass; no amount of parallel processing power can eliminate the need to serially traverse at least one path from a root to every live object.[1] Of course, until a pass is completed no memory may be reclaimed.

The unfortunate upshot of all this is that, in the presence of massively serial data structures, precise garbage collection can nullify the entire parallel advantage of an MPP! Here is how this can happen.

Consider a parallel program running on an N processor MPP. Suppose that after it has run for some time O(K), doing at most O(KN) work, memory is exhausted. During that time, a massively serial data structure of length O(KN) could have been constructed, and thus a precise GC pass takes time O(KN) to complete. Until the GC is complete, no memory is freed and the program is stalled.

Note that regardless of whether or not computation and garbage collection are overlapped, the cumulative time due to compute and GC is O(KN); let us call this period one *step*. The amount of work accomplished in one step is O(KN). If we divide the work accomplished in one step by the time it takes to complete one step, we find that the effective parallelism we have gotten from our N-node MPP is... O(1)!

This is already unacceptable, but in fact, the worst case is even worse (albeit unlikely to occur in practice): if, rather than generating a fresh MSDS of length O(KN), the program were to extend an existing MSDS by the same length, then after C steps, the program could in theory be maintaining an MSDS of length O(CKN); in this awful case, after a step of

[1]Although pointer jumping [36] can solve the problem for certain limited data structures, in the general case in which each object may have multiple incoming and outgoing pointers, it cannot.

time O(K), garbage collection will take time O(CKN).

Practical matters

As a practical matter, one might question whether or not a programmer would ever intentionally generate a massively serial data structure, but in practice, such structures have useful applications.

For instance, the leaves of a binary tree might be connected in a linked-list in order to facilitate rapid traversal of the leaves without needing to traverse interior nodes. The linked leaves would then compose a MSDS. As long as the tree remained rooted, the depth of the overall data structure would only be logarithmic in the total number of leaves.

The real problem arises when an MSDS is rooted in only a few nodes. Unfortunately, there are ways in which such a situation could arise by accident. Most insidious among these ways is the problem that an MSDS such as the linked leaves described above could become singly-rooted during the transition from being multiply-rooted to being unrooted. A garbage collection pass which begins during this transition may discover that only a single node retains a pointer to the MSDS, requiring the garbage collector to traverse the MSDS serially in its entirety. Thus, any program which creates and then forgets about MSDSs may intermittently be stalled for extended periods waiting for memory while the GC traverses an MSDS.

4.2.2 Reference counting and other conservative strategies

If precise garbage collection techniques are detrimental to performance, then we must examine conservative techniques as an alternative. One well-known conservative GC strategy is reference counting [27].

Reference counting has a compelling feature for use on an MPP: it does not have to traverse the entire live-object graph in order to reclaim memory. In fact, there is no notion of a reference counting “pass” as such; reference counting is a continuous process, and memory can be reclaimed as soon as an object’s reference count drops to zero.

Unfortunately, as demonstrated above in Section 4.2, it is easy to make a massively serial, circular linked list. Reference counting can never collect such a list even if the last live pointer to it is destroyed, since in general reference counting cannot reclaim memory occupied by objects in cycles or by objects reachable from cycles.

Thus, while reference counting will never reduce the effective parallelism delivered by an MPP, it may effectively leak memory to garbage cycles, potentially causing eventual failure of the system due to lack of free memory.

Other conservative strategies such as reference flagging provide the same advantages and drawbacks as reference counting in this domain, although they are notably different in terms of other costs. In particular, where reference counting imposes a fixed overhead to each pointer copy and deletion, reference flagging imposes an overhead proportionate to the length of time a pointer survives.

4.2.3 The role of programmer discipline

One can always compensate for the failings of a garbage collection scheme by imposing constraints on the programmer. For instance, in reference-counting systems, the programmer is expected to avoid creating cycles, or to explicitly destroy cycles nature before releasing them.

Under a precise GC scheme on an MPP, one could insist that programmers either never create MSDSs, or explicitly destroy the inter-structure links before releasing them as discussed in Section 3.

Although it is impossible to say with certainty which type of discipline is more “natural” to impose on programmers, it is certainly simpler to break a cycle than it is to destroy the entirety of an MSDS.

It is worth noting that in pure functional languages, and in languages that feature write-once variables (e.g. “final” variables in Java), it may be impossible for the programmer to destroy pointers within a data structure — either to break cycles, or destroy MSDSs.

4.2.4 A typical hybrid approach

A hybrid strategy provides some of the best of both worlds. One fairly typical hybrid strategy uses reference counting to provide short-term garbage collection. If, at some point, reference counting is unable to reclaim enough memory, precise mark-sweep collection is invoked to reclaim memory occupied by garbage cycles. The mark-sweep collector may also run continuously in the background at a relatively low priority in order to exploit otherwise idle processor time.

On a system employing this strategy, the programmer who avoids creating garbage cycles need never fear extended garbage collection stalls, and even the programmer who creates garbage cycles need not fear running out of memory due to uncollectable garbage cycles.

Although I am endorsing hybrid schemes here, in following chapters of this thesis, I will focus on designing a primary garbage collection mechanism based on reference counting capable of managing sparsely faceted arrays; a secondary mark/sweep strategy is beyond the scope of this thesis.

4.3 A new, parallel marking algorithm

Although I do not explore the idea further in this thesis, in this section I briefly present a novel, conservative garbage collection algorithm for parallel garbage collection. Unlike other conservative GC algorithms, this algorithm has the ability to collect some (but not necessarily all) garbage cycles; in fact, given an oracle, it is actually a precise garbage collection algorithm. In the absence of an oracle, however, it is unable to reliably collect garbage structures which other conservative strategies can reclaim. Thus, I present this algorithm here not as a finished work, but rather as as a jumping-off point for future work.

4.3.1 Parallel connected components

This GC technique is based on computing the connected components of an undirected graph in parallel. In general, a parallel connected components algorithm operates in the following fashion:

First, every node in the graph begins as the sole member of its own supernode. Then the following sequence of operations is performed:

1. Each supernode selects an edge going to some other supernode.

2. A set of supernodes all connected by selected edges are coalesced into a single supernode.

3. Iterate from step 1 until surviving supernodes have no outgoing edges.

Each iteration of this algorithm cuts the number of supernodes with outgoing edges by at least half; thus, it completes in O$(\log N)$ iterations.

4.3.2 The GC algorithm

The general idea behind this new parallel GC scheme is to coalesce connected groups of objects into supernodes. At the end of the process, all objects in a supernode which contains no objects reachable from a root are definitely garbage; all other objects are preserved. In the absence of an oracle to help pick supernode-coalescing order, the strategy is conservative; garbage objects may be coalesced into supernodes with non-garbage objects and therefore preserved through the GC pass. Below is a more detailed description of the scheme.

Preliminary assumptions

This GC scheme makes two assumptions.

1. It assumes that the nodes are able to directly, immediately access each object (or rather each meta-object, as per the discussion above in Section 4.1), whether or not they are reachable from real roots; any GC scheme in which each node maintains IN and OUT lists will meet this assumption handily, as the IN and OUT entries identify all objects involved.

2. It assumes that it is feasible to determine, for each meta-object, the set of incoming references. Given IN and OUT lists, this is not technically difficult to accomplish — for every OUT entry, send information to the IN node necessary to “point” in the other direction — but in general will take time proportionate to the maximum number of incoming references to a single node.

Connected components variation

Each meta-object is treated as a single graph node; each graph node starts as the sole component of a supernode. Each supernode reachable from a root on its local processor is marked; all other supernodes are unmarked.

Garbage collection then involves the following iteration:

1. For each marked supernode, select one outgoing edge to another supernode (if such an edge exists.)

2. For each unmarked supernode, select one incoming edge from another supernode (if such an edge exists.)

3. Perform pointer-jumping across the selected edges twice:

   (a) The first performs leader-election in newly-connected groups of supernodes.

   (b) The second propagates the identity of the leader to all supernodes as their new supernode identifier.

4. If any object in a supernode is marked, the supernode is considered to be marked in its entirety.

5. Iterate until steps 2 and 1 both fail to add new edges.

When this algorithm concludes, objects in unmarked supernodes are garbage and can be freed; objects in marked supernodes may or may not be garbage, and must be preserved.

On its own, Step 1 is simply a limited form of precise marking; by selecting only a single node, it actually foregoes opportunities for parallelism.

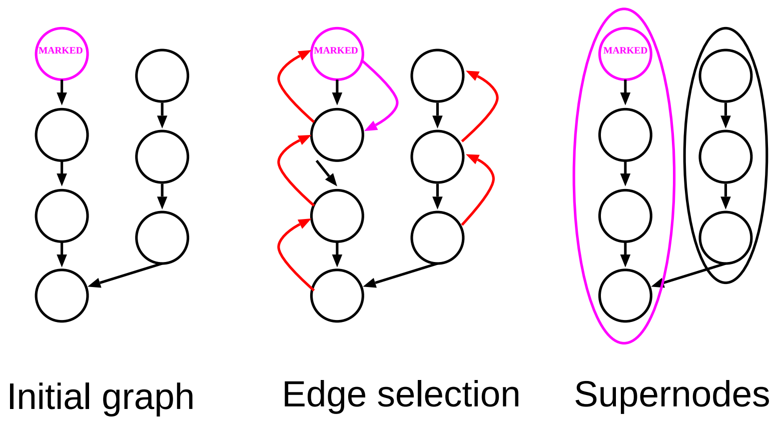


Figure 4-3: For each reachable node, an oracle picks only incoming edges which are on legitimate paths from marked supernodes. This ensures that garbage objects are never coalesced into a marked supernode.

It is Step 2 that this algorithm novel; this step causes supernodes which are not yet connected to a marked root to speculatively join with other supernodes. See Figures 4-3 and 4-4 for examples of the process of picking edges.

4.3.3 Why this algorithm is conservative

If we could somehow maintain the invariant that all supernodes contain exclusively live objects or garbage objects, but never both, then Step 1 would never break that invariant. Given such an invariant, this step simply represents tracing the graph of marked data just as any precise GC mechanism would; indeed, if we omit Step 2, this algorithm is exactly a standard precise parallel mark algorithm, and will take time proportionate to the depth of the longest serial data structure just as described above in Section 4.2.1.

While the choice of outgoing edges in Step 1 is safe with respect to the desirable invariant, Step 2 is speculative. It represents an as-yet unmarked supernode trying to climb "up" toward some marked root in order to become marked itself. Because it is speculative, Step 2 runs the risk of commingling garbage and live objects in a single supernode, and thus destroying our ability to maintain the desired invariant.

Here are the possible cases of outgoing edge selection in Step 2:

- The selected edge is between two live objects. No danger is posed to the invariant; the selection connects two supernodes of live objects.

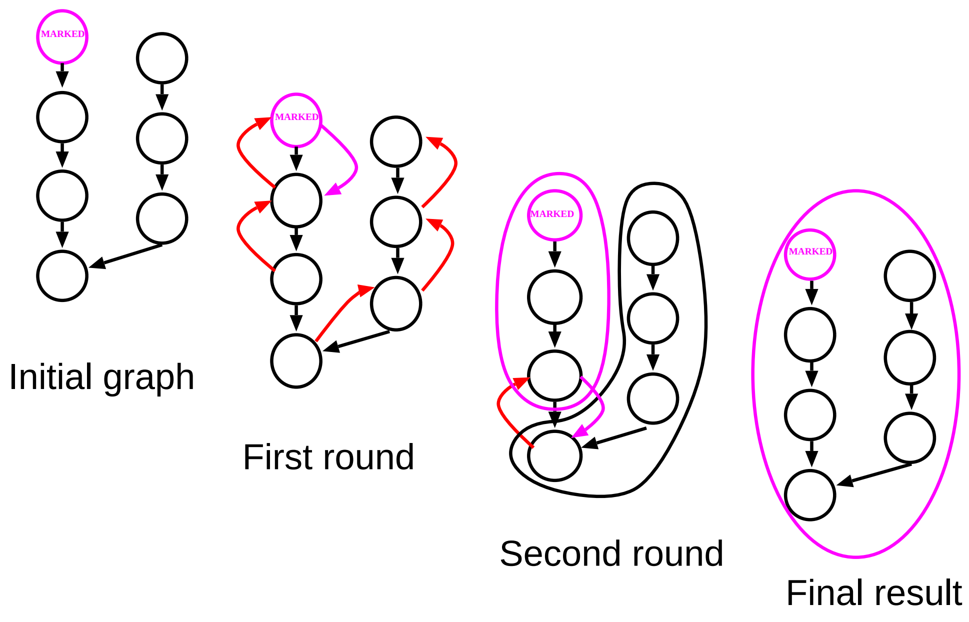


Figure 4-4: Bad decisions may pick incoming edges from unreachable objects to reachable objects, thus causing garbage objects to coalesce into marked supernodes.

- The selected edge is between two garbage objects. Again, no danger is posed to the invariant; the selection connects two supernodes of garbage objects.

- By definition of garbage object, it is impossible for the selected edge to point from a live object to a garbage object; otherwise, the target object would not, in fact, be garbage. Thus, this case cannot actually occur.

- The remaining case is the problem case: the selected edge is a pointer from a garbage object to a live object. This will destroy the invariant, as supernodes containing garbage and live objects will be coalesced together in Step 3.

Figure 4-3 illustrates the result of good choices (i.e. all choices from the first two cases); Figure 4-3 illustrates the result of a less successful set of choices (i.e. some choices from the final case.)

Running time

It is difficult to provide a satisfying bound on the running time of this GC algorithm, largely because in practice the problem does not conform to typical theoretical assumptions of one graph edge or one graph vertex per processing element. However, if we make the simplifying assumption that each processor holds O(1) objects featuring O(1) incoming and outgoing edges we can perform a rudimentary analysis. Note that on N nodes, this assumption implies that there are, overall, O(N) objects and edges.

Under this assumption, prior to a GC pass, constructing the set of an object's incoming edges will take time O(1), as each node with an outgoing pointer sends connectivity to the pointer's target node.

For a given supernode, selecting a single edge is equivalent to leader-election, requiring O$(\log N)$ communications steps. Similarly, the two subsequent pointer-jumping steps require O$(\log N)$ communications steps each.

If we treat a marked supernode with no outgoing edges as if it has been removed from the graph, and we also treat an unmarked supernode with no incoming edges as if it has been removed from the graph, then we can say that every step of the algorithm reduces the number of supernodes in the graph by at least half, since every remaining supernode is merged with at least one other supernode. Thus, the algorithm iterates O$(\log N)$ times, for a total of O$(\log N^2)$ steps.

The cost of an individual step is dependent upon the cost of inter-processor communications. Under a PRAM model, where the cost of each step is O(1), the algorithm completes in time O$(\log N^2)$ steps; on an architecture with a log-deep network such as a hypercube, butterfly, or fat-tree, the algorithm completes in time O$(\log N^3)$; etc.

In specifying and analyzing this GC algorithm, I build on the simplest possible approach to parallel connected components. There is a lot of work on improving the exponent for various types of parallel architecture (see, e.g., [28]). It is quite possible that one or more of these superior algorithms could be adapted to suit this GC algorithm. However, as will be discussed in the next section, this GC algorithm is presently conservative to the point of unusability; additional thought spent on this algorithm should be directed toward reducing the conservativism before it is spent figuring out how to reduce the exponent.

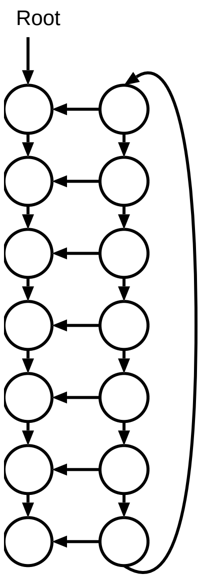


Figure 4-5: A plausible garbage data structure: each element of a garbage circular linked list points to an element in a live linked list.

The role of an oracle, and an open question

If incoming edges are simply selected randomly, any garbage structure which has a lot of pointers to live data is likely to survive indefinitely. Consider the data structure of Figure 4-5, in which there is a live linked list, and a garbage circular linked list. Every node in the live list has two incoming pointers: one from a live object, one from a garbage object. If, during a GC pass, any single live object happens to choose a garbage object in Step 2 of the algorithm, all of the garbage objects will be preserved. If each list contains N objects, then the likelihood of every live object making the correct decision is $\frac{1}{2^N}$. For even moderately-sized N, random decision-making will lead to indefinite preservation of garbage objects.

An oracle would, of course, solve this problem. In fact, given a perfect oracle to guide the choice of edges in Step 2, this garbage collection strategy would be precise; the oracular case is shown in Figure 4-3. In the absence of a perfect oracle, however, this strategy is conservative; the imperfect counterpart to the case shown in Figure 4-3 is shown in Figure 4-4.

The key question, which I leave open here, is whether or not there are heuristics, perhaps based on the results of previous rounds of garbage collection, that can successfully approximate a good oracle.

4.4 Conclusion

In this chapter I have shown why precise garbage collection schemes are a poor fit for inter-node garbage collection on massively parallel processors: in the worst case, they can effectively nullify the entirety of the machine's parallelism. Reference counting, while conservative in its memory reclamation, does not suffer from this problem. As a result, in the following chapter I focus on a reference-counting-based garbage collection strategy for managing sparsely faceted objects.

The other, minor contribution of this chapter is a a novel, parallel garbage collection algorithm which, although conservative to the point of uselessness, has the ability to reclaim some garbage cycles, and has better theoretical running time than conventional precise techniques on an MPP. Given an oracle, my new GC scheme would become precise and therefore useful; I have left open the question of whether there are heuristic approximations to an oracle which could notably improve its precision. Because in its current state this algorithm is too conservative to be useful, I do not make use of it elsewhere in this thesis.

Chapter 5

Garbage Collection of Sparsely-Faceted Arrays

5.1 Overview

A scheme for automatic memory management with sparsely faceted arrays must meet several requirements. It must ensure that for a given SFA, at most one facet is allocated on each node. It must also ensure that as long as any node in the system holds a pointer to an SFA, all of the SFA's facets remain available even when there are no pointers on some of the local nodes. Finally, it must ensure that when no nodes in the system hold pointers to an SFA, all of its facets are freed.

The first requirement is met by recording facet allocation in the translation tables discussed in Chapter 2. In this chapter I describe a method of garbage collection which accomplishes the remaining two requirements. My method extends Indirect Reference Counting (IRC) [52], a distributed reference counting scheme which tracks references between independently garbage-collected *areas*.

For clarity of exposition, in this chapter I assume a one-to-one mapping between areas and processing nodes, i.e. each node contains exactly one area. In Chapter 6, I briefly describe extensions to the garbage collection scheme of this chapter that enable independent management of multiple independent areas per node.

This chapter is organized as follows. In Section 5.2, I describe a garbage collection scheme, based on indirect reference counting, which is able to perform correctly and efficiently in the presence of sparsely faceted arrays. In Section 5.3, I discuss the implementation of the GC bookkeeping tables, and study the efficacy of dedicated bookkeeping caches in simulation. In Section 5.4, I describe an optimization which reduces the impact of the unused-facet problem. I close with a summary discussion in Section 5.5.

5.2 SFA-Aware Indirect Reference Counting

Any scheme for automatic management of SFAs should meet the following two requirements:

1. Correctness: No facet of an SFA can be destroyed while there are any live pointers to the SFA in any node.[1]

2. Efficiency: In order to free an SFA without broadcasting a message to every area/node, at free-time the system must be able to determine the location of each of the SFA's allocated facets.

As I have shown in Chapter 4, reference counting is well-suited to garbage collection in a massively parallel system because it does not have to trace the entire live-object graph in order to free memory. Thus, I meet the above requirements by extending a distributed reference counting strategy, Indirect Reference Counting (IRC) [52], such that it can manage sparsely faceted arrays in addition to scalar objects.

5.2.1 A quick review of IRC

Indirect Reference Counting is a distributed reference counting strategy intended for tracking inter-area pointers; it cooperates with an area-local garbage collection system to provide whole-system GC. In this section I provide a very brief review; a more comprehensive review is presented in Appendix A.

[1]Actually, there is an exceptional/optimization case, discussed in Section 5.4, in which it is not only acceptable but beneficial to delete an SFA facet.

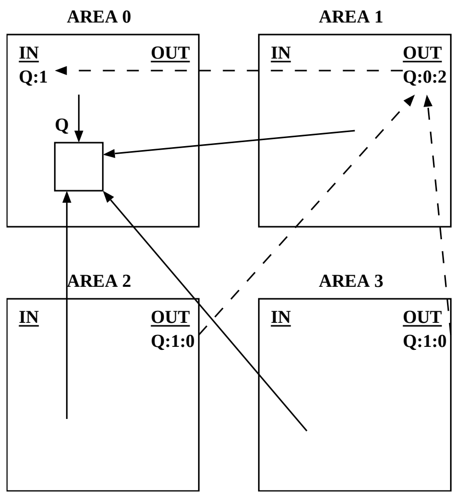


Figure 5-1: Pointer to Q copied from Area 0 to Area 1, then from Area 1 to Areas 2 and 3. The IRC fan-in copy-tree is dashed.

Under IRC, when an area first exports a pointer to a locally-allocated scalar object, it creates an IN entry containing a reference count; the reference count is incremented for every pointer sent to other nodes. When an area A first receives a pointer to a scalar object, it creates a local OUT entry recording the sending (AKA parent) area. The OUT entry also contains a reference count which is initially zero, but is incremented every time A sends a copy of the pointer to another node.

This strategy results in the construction of a fan-in copy-tree of IRC entries whose structure reflects the pattern of distribution of the pointer. See Figure 5-1.

Area-local GC uses IN and OUT entries as garbage-collection roots when GCing an area, and is responsible for identifying OUT entries for pointers of which the area no longer holds any copies.

When an OUT entry's reference count is zero — i.e. the OUT entry is a leaf of the IRC tree — and area-local GC determines that there are no copies of its pointer left in its area, the OUT entry is destroyed and its parent area is notified. The parent area decrements its corresponding IN or OUT entry's reference count, possibly reducing it to zero and making it, itself, a possible subject for destruction.

When all OUT entries have been destroyed, the IN entry in the object's home area will

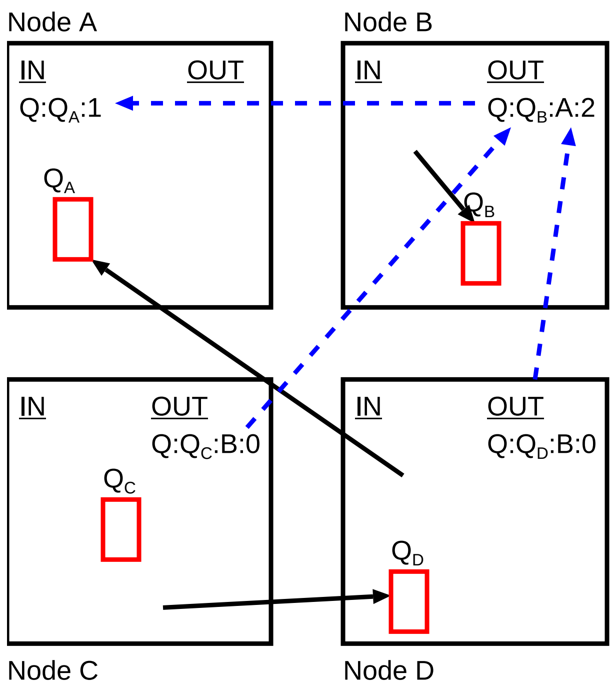


Figure 5-2: An IRC copy tree for an SFA; note facets on all nodes.

have a reference count of zero and can be destroyed regardless of whether the home area still contains copies of the pointer.

5.2.2 Making IRC SFA-aware

Unmodified, indirect reference counting is suitable only for managing scalar objects. In this section I will show how to extend IRC such that it can manage sparsely faceted arrays as well.

Copy tree construction

In SFA-aware IRC, copy-tree construction proceeds as usual. Each time an area receives a pointer to an SFA for the first time, it constructs a new OUT entry for the SFA; the OUT entry records the area from which the pointer was received as usual. At the same time, of course, the node allocates a local facet for the SFA and makes an entry in the both the translation table and in the OUT entry. (I will discuss the possibility of merging the IRC entries and translation entries in Section 5.3.2.) See Figure 5-2.

Area-local GC

Just as node-local GC uses scalar objects recorded by IN entries as garbage-collection roots when GCing an area, local garbage collection use the local facets recorded by IN and OUT entries as roots; local facets, and anything reachable from them, are therefore preserved by each local GC pass. Local GC may freely relocate a facet, but it must update the corresponding IRC entry.

Unparenting OUT entries

In order to meet Requirement 1, Correctness, we cannot destroy a OUT entry when we might destroy a scalar OUT entry; it records the existence of a local facet. Thus, we will simply *unparent* such an entry.

Unparenting a OUT entry is as simple as decrementing the reference count of its parent in the IRC reference tree. If its node later receives a pointer to the SFA again, the OUT entry is re-used, recording the sending node as its new IRC parent; this process is called *reparenting*.

Area-local garbage collection continues to use unparented OUT entries as roots for the local GC.

Aside from the fact that OUT entries are not eliminated when it sends a decrement to its parent area, IRC proceeds identically for SFA references as for scalar references: the reference count in an OUT entry increases each time its pointer is sent to another area, and decreases when decrements are received from other areas.

Anchoring OUT entries

When all OUT entries for a particular SFA are unparented, the reference count on the SFA's IN entry will be zero. This condition, in conjunction with the discovery that there are no pointers to the SFA on its home node, identifies the SFA as entirely unreachable; at this point, the SFA is garbage and all of its facets may be freed.

However, when an SFA's IN entry reference count goes to zero, the system needs to be able to discover the set of nodes which have unparented OUT entries in order to meet

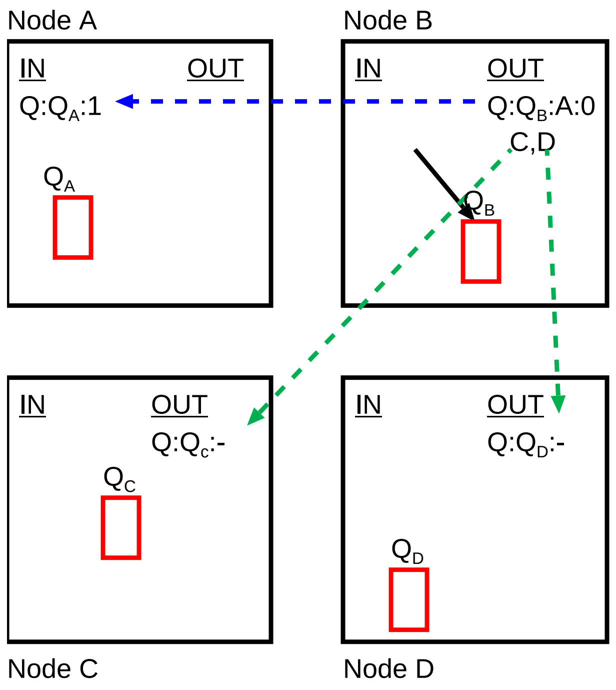


Figure 5-3: A partially-formed anchor tree; note facets are preserved on unparented nodes.

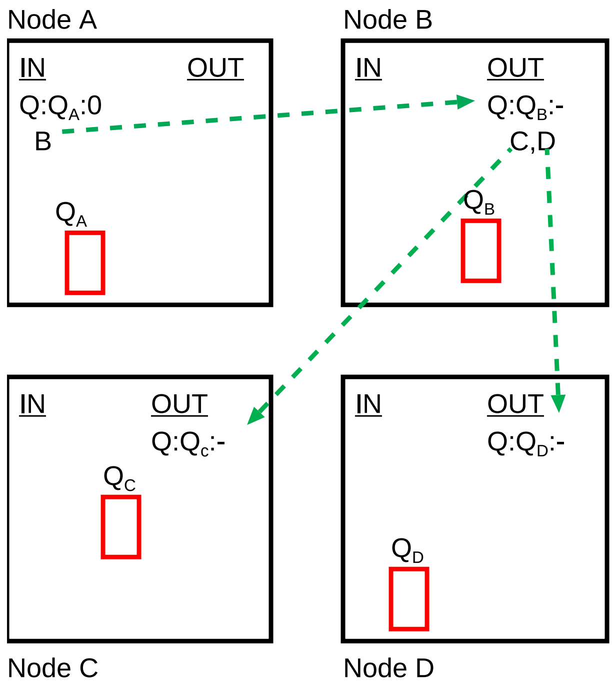


Figure 5-4: A completely-formed anchor tree.

Requirement 2. This requirement is met by *anchoring* each OUT entry the first time it becomes unparented. Anchoring is essentially the process of reversing the directionality of the edges in an IRC reference tree.

A OUT entry requests anchoring by setting an ANCHOR bit in the decrement request sent to its parent-area. Upon receiving a decrement message with set ANCHOR bit, a parent area both decrements the reference count in its IRC entry as usual, and also records the requesting area in an "anchored" list attached to the IRC entry. See Figure 5-3.

Since the direction of each link in the IRC copy-tree is reversed when a OUT entry becomes unparented, when all of an SFA's OUT entries have become unparented, they are all anchored; the IN entry is thus the root of an anchor-tree of pointers which fan out to identify every area containing one of the SFA's facet. See Figure 5-3.

One minor note for completeness: as mentioned in the previous section, a OUT entry may be unparented and reparented multiple times over the course of its lifetime; however, it is only anchored the first time it becomes unparented, and is never un-anchored thereafter.

Freeing an SFA

An SFA is garbage when its IN entry has a zero reference count and area-local GC determines that there are no pointers to the SFA within its home area. To free a garbage SFA, its home node simply sends a delete message to each anchored-child of the SFA, and deletes the IN entry; each anchored area, upon receiving such a delete message, recursively sends it to any anchored-children rooted in its OUT entry, and then destroys the OUT entry. With OUT and IN entries destroyed, node-local GC can reclaim the now-orphaned facets on its next pass.

Delete messages add another message to the communications cost of an individual pointer-copy, raising the number to as high as three: the initial copy, the subsequent decrement/anchor, and the final delete; in cases of OUT entries that are repeatedly reparented, the cost of the latter pointer copies will still only be two messages. Regardless, the aggregate communications overhead per inter-area pointer copy is still $\mathrm{O}(1)$.

Building a better anchor-tree

Unfortunately, if a particular node has a large number of children, the "anchored" list of an entry can grow to occupy space of O(N) for N areas. This eliminates one of the key advantages of reference counting. In order to avoid this problem, we can employ a more sophisticated anchoring strategy using distributed trees instead of local lists.

Under this approach, each faceted IRC entry contains two slots[2] in which to record anchored areas. An area honors the first two anchor-requests it receives for a particular pointer by recording the requesting areas in the two slots of the area's IRC entry for the pointer.

For each subsequent anchor-request the area receives, it forwards the request to one or the other of the recorded, anchored areas. Recipient areas forward the message recursively until an empty slot is found in some part of the anchor-tree.

This strategy regains the benefits of O(1) storage per node per SFA, but increases the potential messaging cost for a single pointer copy to O(N)in the worst case as anchor messages are forwarded — although in practice, these costs will almost certainly be much, much lower.

Timing

Although indirect reference counting is generally self-synchronizing, the method of forwarding anchor requests introduces the possibility of a specific problem: suppose that a OUT entry sends a decrement/anchor request to its parent in the IRC tree. Suppose that the parent must forward the anchor request; the anchor message is launched toward one of the parent's anchor-children.

Let us further suppose that the initial decrement action is sufficient to zero the reference count for the entire SFA – the SFA has become garbage. In this case, delete messages will be sent from the anchor-tree root IN entry; each receiving OUT entry will forward the deletion request to its anchored-children before destroying itself.

[2]For a shallower tree, we could use larger numbers of slots, but that would increase the per-pointer, per-node space overhead.

The problem-case arises if the forwarded anchor request is delayed in the network, and the deletion messages from parent to child overtake it. In this case, when the anchor request arrives at its destination area, the area will have no record of the SFA whose facets this request is trying to anchor.

As long as an SFA's GUID is not re-used after its destruction, this problem case is easily handled. An area which receives an anchor message for an SFA for which it has no record may assume that the SFA has been deleted; the area simply sends a delete message to the area being anchored.

On the other hand, if SFA GUIDs are re-used, then there is a risk that that this sort of message reordering could result in the facets of an old SFA becoming associated with a new, unrelated SFA. In this case, a protocol must be adopted to ensure that delete messages can never overtake anchor messages in the network.

5.3 Implementing IN and OUT sets

Each node must maintain a table of IN and OUT entries; these entries will be manipulated both by the network interface, and by the node-local garbage collector. As with SFA translation tables, these tables may be implemented as hashtables stored in the node's physical memory. In fact, although IN and OUT sets are conceptually distinct, we can implement them as a single hashtable residing entirely within physical memory; each entry in the the hashtable is keyed by the pointer (and, for SFA entries, the GUID) it documents.

Unlike SFA translation operations, whose latency directly impacts the performance of message sends and receives, IRC bookkeeping operations can be pipelined without delaying message delivery. Correspondingly, the motivation to avoid latency in performing IRC bookkeeping is therefore much lower than it is in performing translations.

However, avoiding unnecessary use of main memory bandwidth remains a concern, and an IRC entry update is expensive. In the best case, the pointer is hashed; the appropriate entry-key is read from memory; the pointer is compared against the key; the reference count is read from memory; the count is incremented; the count is written back to memory. To write a single pointer to remote memory, this adds at least three memory references –

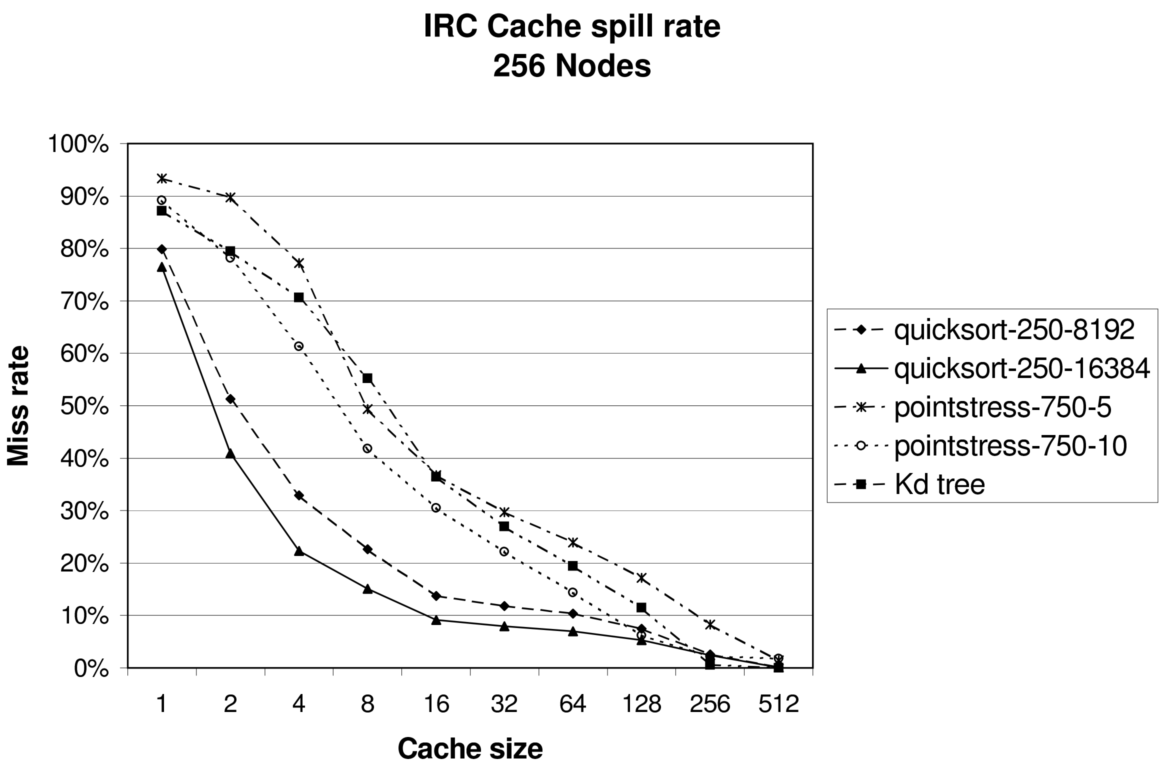


Figure 5-5: IRC cache spill rate for various applications and cache sizes.

two word-reads, one word-write – at both source and destination nodes. This effectively quadruples the memory bandwidth cost of the copy at the destination; since the sending node may have been sending the value from a register, this cost represents an infinite increase in the memory bandwidth cost at the sender.

These costs motivate the consideration of a dedicated IRC-entry cache in the network interface.

5.3.1 A cache for IRC entries

An IRC cache is keyed on object pointers, and contains abbreviated versions of IN and OUT entries. An increment or decrement request which hits in the cache requires no memory access. An increment or decrement request which misses in the cache causes an invalid (unused) or clean (contents are backed by the IRC entry in memory) cache entry to be allocated to handle the request. Note that since cache entries may be allocated to handle incoming decrements, childcounts in the cache may be negative.

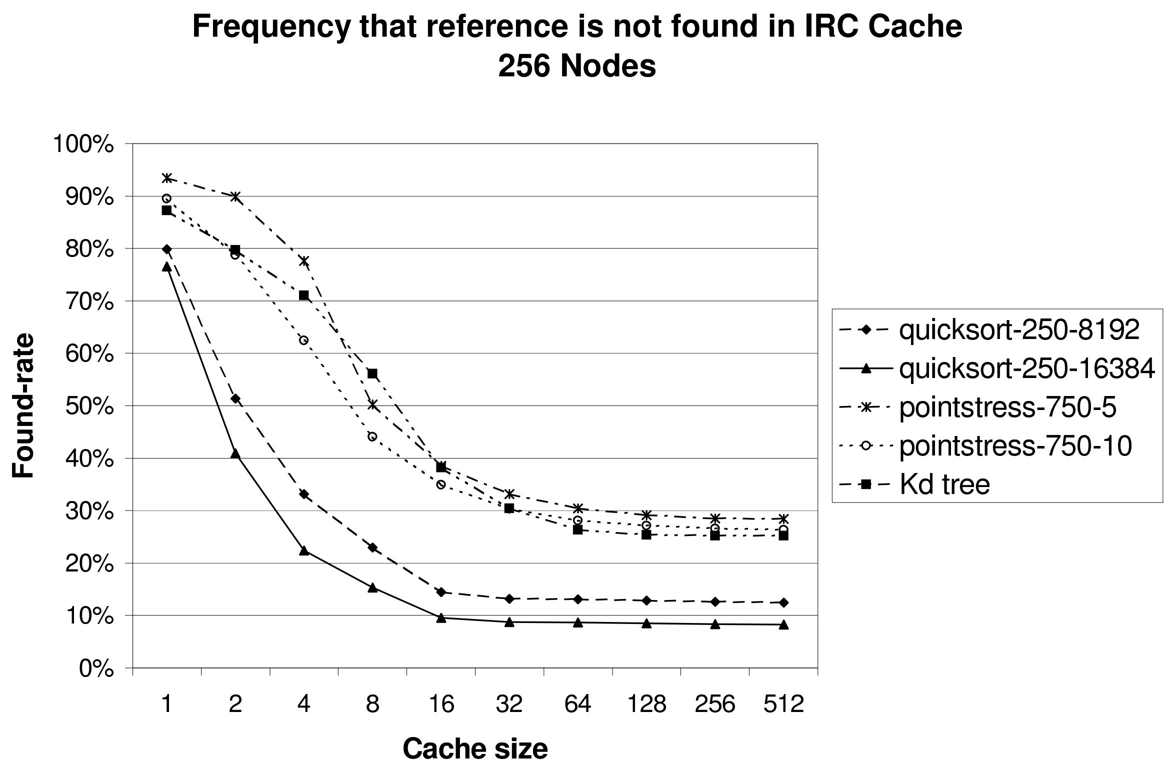


Figure 5-6: IRC cache "miss" rate for various applications and cache sizes.

A cached IN entry whose childcount goes to zero may be declared clean. However, in the presence of an incremental area-local garbage collector (e.g. Baker's copying collector), a cached OUT entry whose childcount goes to zero is dirty until the backing OUT entry in the main hashtable is marked. This is because a pointer received from a remote node can be written into an already-swept part of the area; if the IRC entry is not marked when the cache entry is flushed, when the GC pass completes, it could erroneously conclude that no local copies of the pointer still exist, and free the entry.

In the event that no invalid or clean entries are available, the NI logic must merge some or all cache entries into the main-memory hashtable, marking the merged entries clean; the cache implementation I study in this section cleans all cache entries at once.

Merging an OUT entry in the cache with an OUT entry in the hashtable may uncover an unusual situation: the two entries could refer to different copy-tree parents. In this event, the count from the cache entry is added to the count in the hashtable entry, and a decrement message is sent to the parent-area named in the cache's entry.

As long as a relatively small number of pointers are being exchanged between areas at any one time, such caches should be very successful at minimizing the cost of IN/OUT bookkeeping. The actual success of IRC entry caches is obviously dependent both on applications and on cache geometries.

Simulation results

Figure 5-5 shows the cache spill rate for a dedicated, fully associative IRC cache which is used for both incoming and outgoing pointer bookkeeping. The applications shown are the same as those used in Section 3.4.3; quicksort and pointer stressmark are each run for two problem sizes, and Kd tree for one. All are run under the Mesarthim simulator, described in Chapter 6, simulating a 256 node system.

The lesson of the spill rate graph is that by a cache size of 512 entries, the spill rate has gone to zero for these benchmarks. This implies that the miss rates shown for a 512-entry cache in the following figure, Figure 5-6, are compulsory, rather than capacity misses. Tracking backwards, we see that caches with as few as 64 entries have miss rates that are only slightly larger than the rates seen with bigger caches.

Each of the three test applications performs some of its inter-node communications via shared memory operations on scalar objects, generally allocating a new scalar object for each inter-node thread invocation; the constant generation of new scalar objects is the primary reason for the high compulsory miss rates which, for one application, exceeds 30% for a cache with 64 entries.

As noted in the previous section, cache misses do not necessarily require communications with memory; the IRC cache can simply record the information to be merged into memory later, and in some cases (i.e. cached IN entries whose reference count goes to zero), the information may never need to be written back. Thus, for this cache, in a long-running program, the steady state spill rate may be well below the miss rate; this is good news, since the actual memory bandwidth cost is related to the spill cost, not the miss rate.

Unfortunately, the benchmark applications are not sufficiently long-running in simulation to reliably illustrate the precise steady-state behavior. Thus, we must settle for the upper bound on cache spill rates provided by the cache miss rate. Although a 30% miss rate is quite high, it nevertheless implies a 70% hit rate, which would represent a significant

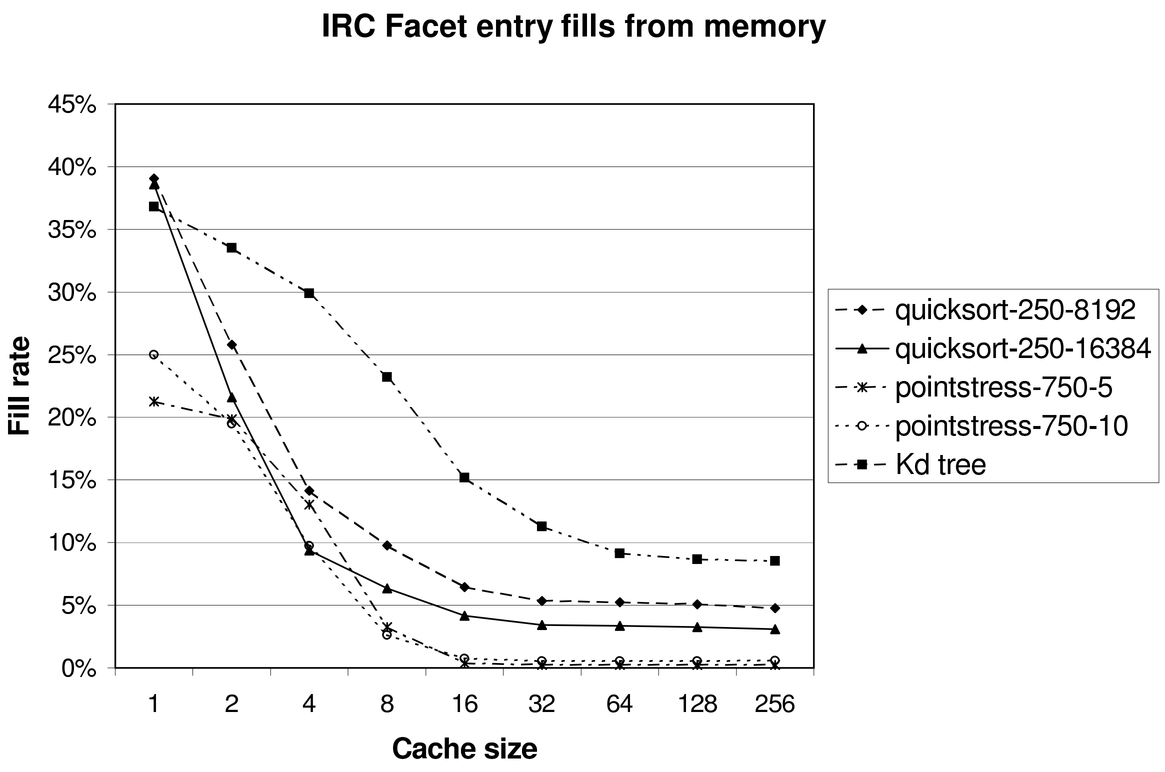


Figure 5-7: Combined IRC/translation cache translation fill-from-memory rate for various applications and cache sizes.

decrease in main-memory bandwidth consumption due to IRC bookkeeping.

5.3.2 Using IRC entries as translation records

When an SFA is first created, a single facet is allocated on the creating node. As long as pointers to the SFA don't escape the initial area, there is no need to generate a globally unique identifier for the SFA.

A GUID must be generated when a pointer to the SFA is first sent to another node/area. At the same time, an IRC IN entry must be created for the SFA. A faceted IRC entry must be placed in its area's IRC table under two keys: the GUID, and the local facet name.

Similarly, when a node/area first receives a pointer to an SFA, it must allocate a local facet to associate with the SFA's GUID; at the same time, it must create an IRC OUT entry for the SFA.

Rather than maintaining separate data structures for translation and for garbage collection, we may wish to consider using the mapping stored in the IN and OUT entries for sparsely faceted arrays.By storing translation and and IRC data in a single entry, we only need to perform a single lookup operation in an IRC entry hashtable.

By adding a few fields to the IRC cache discussed in the previous section, we can perform most SFA-related updates in cache. The rules for working with faceted IRC cache entries are slightly different from those for scalar IRC entries.

Most notably, if an SFA reference requiring translation misses in the IRC cache, the network interface is required to check the main hashtable to see if a translation (and thus an IRC entry) already exists. If so, then the appropriate information is copied from the entry into the cache from the hashtable; if not, a new IRC entry is created along with the appropriate translation information, recorded in the hashtable, and a cache entry set up.

There are two types of SFA-related references which do not require translation, and therefore even if they miss in the cache do not require references to the backing hashtable. These references are first, decrement requests received from remote areas; and second, anchor requests received from remote areas. Both types of request can be satisfied by recording information – either a decrement, or the identity of the area requesting anchoring – in a freshly-allocated cache line; the line must be marked as dirty but unsuitable for answering translation-requests.

When a cache entry which has recorded anchor-requests is merged with the IRC entry in the hashtable, if the IRC entry has already filled its anchor-slots, the requests may need to be forwarded to its anchor-children; this can be handled by the NI software/firmware that handles the merging.

Simulation results

Figure 5-7 shows the frequency with which references to a combined IRC/translation cache must retrieve facet/GUID translation information from main memory. Although this "facet fill" rate is lower than the translation cache miss rates demonstrated in Section 3.4.3, the comparison is not straightforward: the translation cache is only referenced for translation operations, whereas the combined IRC/translation cache is referenced for IRC bookkeeping operations in addition to translations. However, since the fill rate levels out for caches of 64 entries or more, we may conclude that remaining fills are compulsory, rather than due to capacity limitations, and therefore that a cache of 64 entries or more is adequate to perform both translation and bookkeeping with close to optimal cache performance.

5.3.3 Simulation caveats

I will reiterate this caveat one last time: the cache simulations performed in the preceding sections are indicative, not definitive. The applications used are simplistic, and the simulation is idealized. While these results suggest that a small, dedicated cache can handle a large majority of IRC bookkeeping and SFA translation tasks without requiring access to main memory, the task of picking a cache size for a specific architecture will require both a precise simulation of the architecture, and a set of full-scale applications which are representative of the intended workload.

5.4 An optimization

One failing of SFAs is the problem of unused facets: any node which receives an SFA pointer will allocate a facet for it, even if the facet is unused and the pointer is rapidly destroyed.

With some help from the local garbage collector, SFA-aware IRC can be modified to eliminate these unused facets, and in some cases the corresponding IRC entries as well.

Recovering local memory

Assume that a facet, when first allocated, is initialized to some initial value, e.g. 0.

At the end of any local GC pass which discovers that the area contains no pointers to an SFA, and also that the local facet contains only the initial value, the IRC entry's "local-pointer" field may be set to INVALID, and the local facet GCed. If the area later receives a new copy of the pointer, it must notice the INVALID entry and allocate a new local facet to the SFA.

This strategy adds overhead to the process of area-local garbage collection; the corresponding reward will depend heavily on the actual applications.

Eliminating redundant IRC entries

The previous section showed how to eliminate unused facets, but did not eliminate their IRC entries. In this section, I show how to eliminate a limited class of IRC entry as well.

Suppose a OUT entry is unanchored, and local GC has set its local-pointer field to INVALID. Suppose further the entry has a zero reference count, and that area-local GC has found that the area contains no copies of its pointer. Normally, SFA-aware IRC would unparent and anchor this entry. However, in this case it can unparent and eliminate the entry instead. A decrement message is sent to the parent as usual; in addition to the decrement request, the message also contains anchor-requests for any anchored-children of the dying OUT entry. This will attach any anchored-children of the dying entry to the anchored-entries subtree rooted in its former parent. Once the decrement message is sent, the OUT entry may be destroyed.

5.5 Summary

Automatic memory management is an important tool in reducing the burden of programming any system. The complexity of parallel programming is intrinsically greater than that of sequential programming, and thus any and all tools which simplify the task are inherently of great value.

In previous chapters, I presented sparsely faceted arrays, a data structure enabling straightforward partitioned programming in terms of both algorithms and data structures, and suggested that SFAs could be garbage collected. In this chapter I have followed through on that suggestion.

In particular, I have described automatic memory mechanisms capable of correctly and efficiently managing sparsely faceted arrays. My mechanism, SFA-aware IRC, meets all three requirements for automatically managing SFAs listed in the introduction to this chapter: it ensures that for a given SFA, at most one facet is allocated on each node; it ensures that as long as any node in the system holds a pointer to an SFA, all of the SFA's facets remain available; and it ensures that when no nodes in the system hold pointers to an SFA, all of its facets are freed.

As with any reference counting scheme, SFA-aware GC cannot collect inter-area garbage cycles; however, as discussed in Chapter 4, it does reclaim memory without the risk of eliminating the parallel advantage of an MPP imposed by precise GC schemes.

Chapter 6

Mesarthim: a high-level simulation of a parallel system

γ Ari: This beautiful pair is one of the best known in the sky and one of the first to be discovered; the pale yellow stars dominate a field well sprinkled with scattered stars. There has been no change since at least 1830 and common proper motions indicates a physical system of very long period.
— *Astronomical Objects*, by E J Hartung. Cambridge University Press, 1984. [22]

In this chapter I describe Mesarthim[1], a high-level simulation of a distributed shared memory parallel computer and operating system. Mesarthim is designed to accurately count heap-related events of various types, but no attempt is made at cycle-accurate simulation of a specific architecture. System aspects not directly related to heap management are not generally not simulated.

Developing Mesarthim provided an opportunity to experiment with the implementation, use, and garbage collection of sparsely faceted arrays. Lessons learned during the development experience are reflected in the final designs presented in the earlier chapters of this thesis.

[1]The research reported in this thesis was performed under the auspices of the Aries group at the MIT AI Lab. Mesarthim is a nickname for γ Ari.

An additional goal in developing Mesarthim was to experiment with garbage collection in Small Physical memory, Large Virtual memory (SPLV) systems. Since these experiments were neither as innovative nor as successful as the SFA experiments, they have gone unmentioned in the rest of this thesis; for the sake of completeness, I will briefly discuss them in this chapter.

6.1 Overview

6.1.1 Feature Overview

Mesarthim simulates a distributed shared memory parallel processor with no inter-node data caching. Each node features processor, network interface, memory, and secondary storage (i.e. disk.)

Data words are tagged with hardware recognized types. Mesarthim provides hardware support for sparsely faceted arrays, and recognizes SFA pointers as distinct from scalar pointers.

To simplify the task of programming and compiling for Mesarthim, processors are stack based, rather than register based. Application programs are written or compiled to an assembly language which is a cross between Alpha assembly [64] and Java Virtual Machine [40] instructions.

Each processor supports lightweight multithreading. Threads synchronize using empty/full bits on words in the heap.

In addition to simulating the basic hardware, Mesarthim also simulates an operating system, including thread and memory management.

6.1.2 Implementation Technology

Mesarthim consists of about 19,000 lines of C code; it runs under Linux and UNIX operating systems, and has been verified to run with identical results on Intel Pentium, AMD Athlon, and Compaq Alpha processors. It is compiled with `gcc` [66] version 2.95.2. Its graphical user interface is based on Gtk++, and was in large part designed using the Glade

Graphical User Interface designer.

Mesarthim uses the *ran1* algorithm from [57] to guarantee consistent, high-quality pseudo-random number generation across platforms.

6.2 System Details

6.2.1 Synchronization

Mesarthim provides synchronization through a distinguished, hardware-recognized EMPTY word type, and an atomic-exchange operation. A thread which attempts to read an empty word is blocked; when an active thread writes into an empty word, any threads blocked on that word are reactivated.

In order to maintain a sequentially consistent memory model, a thread is blocked whenever it has an outstanding memory operation.

This synchronization strategy is similar to that employed by the BBN Butterfly [46] and the Tera MTA [1], among others.

6.2.2 Pointer representation

Mesarthim words are 128 bits, including type tags. Addresses are word-based.

Mesarthim's pointers are *guarded pointers* [9] or *capabilities* which specify not only a target address, but also the bounds of the *segment* in which the address lies. Hardware uses these bounds to dynamically detect access violations. Thus, even without more sophisticated access-control, guarded pointers provide a much more finely-grained mechanism for inter-process protection and cooperation than the traditional page-based protections of conventional architectures.

The specific format of a guarded pointer C for an address A in Segment S is $C = [B, N, F, A]$ where S is composed of N blocks of 2^B words each (allocated block-aligned;) and F is the number of the block into which A points. See Figure 6-1. The presence of the finger-field F allows the rapid discovery of the beginning and end of the segment regardless of where the address A is pointing.

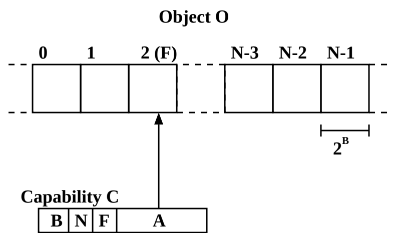


Figure 6-1: The format of a guarded pointer C pointing into an segment S.

A Mesarthim pointer is 82 bits long. An actual address is 64 bits: the upper 20 bits identify a physical node; the lower 10 bits are the page offset; and the middle 34 bits are the virtual page number. The bounds field is encoded with 16 bits: 5 bits for each of N and F, and 6 bits for B. Finally, a pointer-type tag is encoded with two bits, identifying pointers as either scalar, immutable, sfa-local, or sfa-global.

In general, a single object is stored in a given segment; because the bounds encoding has limited resolution, a segment may be up to $\frac{1}{16}$ (i.e. one block of 2^B words) larger than the object it contains.

6.2.3 User-generated messages

Mesarthim provides two types of program-generated inter-node operations: remote memory references, and remote procedure invocations.

Programs do not need to do anything special to perform remote memory accesses. When a node performs a memory access (read, write, or exchange) on an address, the hardware compares the node's ID against the address's node field. If they match, the access completes locally; if not, the appropriate message is constructed and placed into the network.

Remote invocations may be synchronous (i.e. call with return value) or asynchronous (i.e. remote spawn.) Programs must explicitly perform remote procedure invocations, specifying the remote node's identity.

6.2.4 Thread scheduling

Mesarthim provides lightweight threading which enables programs to create many short threads. There are three granularities of thread scheduling.

At the top level are *jobs*. Each application runs as a job. A node with threads from multiple jobs will assign processor time evenly partitioned between each job in a round-robin fashion.

At the next level are subjobs. A given node may have multiple subjobs for a single job; a subjob exists only on a single node. The scheduler divides a job's processor time evenly amongst the job's subjobs in a round-robin fashion. When a message received from a remote node starts a new thread within a given job, that thread starts in a new subjob; this ensures that independent parts of a subproblem can not indefinitely block one another from getting processor time.

At the bottom level are threads. A subjob may contain many threads. Within a subjob, threads are scheduled unfairly, using a scheme similar to that described in [47]. In particular, when a thread spawns a new, child thread, the child is run unfairly until it completes or blocks. This tends to minimize the number of threads spawned on a single processor by executing threads in depth-first order.

When a thread resumes from being blocked, it moves to the head of its subjob.

6.2.5 Cycles

Each node with runnable threads executes one instruction from one application thread on each machine cycle. Operating system and garbage collection operations happen outside of the cycle system; they don't take cycles away from user code. Thus, cycle counts represent an idealized measure of the time it takes user code to complete an algorithm independent of system software overhead.

6.2.6 Heap memory hierarchy

The Mesarthim heap is built on a simple two-level memory hierarchy: physical memory, and disk. Each node has a fixed amount of physical memory, and a TLB (Translation

Lookaside Buffer, a hardware page-map) with an entry for every physical page.

An attempt to access a page which is on disk results in a page-in (often preceded by a page-out); since Mesarthim is event-based, rather than cycle-based, the operations of page-out and page-in are essentially instantaneous – no attempt is made to simulate a realistic paging delay.

6.2.7 Immutable objects

In addition to normal (scalar) objects and sparsely faceted arrays, Mesarthim supports "immutable" objects. Within a node, immutable objects are treated identically to scalar objects. However, pointers to immutable objects are never sent inter-node; instead, when a network interface sees an immutable object pointer, it recursively copies the entire object (i.e. it serializes the object) into the outgoing message; the receiving network interface will deserialize the data into a newly allocated segment. This strategy ensures that references to immutable objects never travel inter-node. Some uses of immutable objects are discussed in Section 6.6.

Supporting immutable objects at the hardware level is, in this case, somewhat unrealistic. In particular, this implementation relies on the fact that paging an immutable object in from disk takes no "time"; a true implementation would need to directly address the issue of immutable data which was not in-core.

6.3 Support for SFAs

6.3.1 Typed pointers

A node's network interface must be able to translate pointers to sparsely faceted arrays from local to global representations and back. A key element in making sparsely faceted arrays work is somehow distinguishing pointers to SFAs from pointers to scalar objects. This can be accomplished in at least two different ways, using either static or dynamic typing strategies.

In the static strategy, there is no typing data associated with a pointer itself; rather, the program must manipulate pointers to SFAs using special instructions to indicate to the hardware that the pointer in question is an SFA pointer. In the dynamic strategy, each pointer is actually tagged to indicate whether or not it refers to an SFA.

In either case, after an SFA pointer has been translated into global form and placed on the network, it must be tagged as an SFA pointer to enable automated translation to local form at the recipient node.

Mesarthim employs the fully dynamic approach; SFA pointers are recognized by hardware, and translation is fully automatic on both ends of an inter-node pointer transmission. If the hardware did not enforce the scalar/SFA distinction, it would be possible for an erroneous program to treat a region of memory as both a facet and as a scalar object. This has the potential to create inconsistent garbage-collection bookkeeping situations, since GC operations are, themselves, pointer-type-dependent. Since, in Mesarthim, all programs run in a shared address space with shared garbage collection services, the conservative strategy of dynamic, hardware-recognized typing seemed the appropriate choice.

6.3.2 GUID generation

As described in Section 3.4, the first time a pointer to an SFA is sent inter-node from its home node, the home node must generate a globally unique ID (GUID) for the SFA. In Mesarthim, the GUID is simply the full address of the SFA's facet on the home node. Because each node has a very large virtual address space (2^{44} words, or 16 Terawords, per node), memory management never normally reuses a virtual address, thus guaranteeing that these GUIDs are unique.

Is it reasonable to assume that GUIDs never need to be reused? Even in a very high-performance hardware implementation, for a single node to allocate and use 2^{44} words of memory will take quite some time. Consider an extreme example: a processor which allocated an average of one word per cycle and ran at 10 Ghz would take nearly 30 minutes to exhaust its local virtual address space.

In the event that a node were to exhaust its local virtual address space, a system-wide garbage collection pass could compact all surviving data into one end of the address space in order to make the remainder of the space usable once again.

6.3.3 Translation Cache

Mesarthim provides a simulated SFA-translation cache; measurements of this cache are used in the discussion of Section 3.4.3.

A single cache is used for both incoming and outgoing translations. An obvious opportunity for future study is to study using distinct caches for incoming and outgoing translation.

6.4 Idealizations and Abstractions

As mentioned in the introduction to this chapter, Mesarthim does not attempt to simulate system aspects which are not directly relevant to the experiments of interest. This section mentions some of the “glossed over” details that a real system implementation would need to address.

6.4.1 Network

The simulated communications network between processing nodes delivers messages with no failures. Messages are enqueued at the recipient node instantaneously; each node processes one incoming messages per user-cycle. The network does not impose any notion of topology.

The network interface performs remote memory requests directly. If a remote memory request encounters an exceptional condition, e.g. it attempts to read an empty word, the network interface creates a thread to run on the local processor; the thread then re-attempts the operation, and if the exceptional condition recurs, the processor can deal with it in the usual fashion.

6.4.2 Executable code distribution

The mechanisms of distributing executable program code were not a topic of study for this thesis, and thus Mesarthim simply elides the problem by providing every node with a copy of every piece of code. Code does not occupy heap memory.

6.4.3 Stacks

Mesarthim provides each thread with its own stack. Each stack frame contains an arbitrary number of named slots. Stacks do not occupy heap memory.

6.5 Garbage Collection

Mesarthim was designed to enable the study of two relatively independent issues in programming DSM architectures. The first issue, already discussed extensively in this thesis, is the implementation, use, and garbage collection of sparsely faceted arrays. The second issue, left largely unmentioned until this chapter, is the implementation of garbage collection on an SPLV system — that is, a system with Small Physical, but Large Virtual, per-node memory.

By far the more interesting of these topics turned out to be sparsely faceted arrays, and so my strategies for local garbage collection have gone unmentioned in most of this thesis. Rather than let these other issues go entirely unremarked upon, however, in this section I provide a high-level description of the entirety of Mesarthim's garbage collection system, and provide some discussion of the qualitative properties of the global and local components.

6.5.1 Garbage collection memory hierarchy

Virtual memory is divided into several classifications for the purposes of garbage collection in Mesarthim. In decreasing order of virtual memory footprint, they are:

- System: The virtual memory of all the nodes in the system.

- Node: The virtual memory on a single node. The upper 20 bits of an address uniquely identify its node.
- Area: The virtual memory of a node is divided into a number of areas.
- Region: An area consists of one or more regions. In Mesarthim, a region is of fixed size, 2^{16} pages; since a page is 10 bits, means that the upper 38 bits of an address uniquely identify its region.
- Segment: A guarded pointer in Mesarthim denotes a segment. Segments are allocated such that each lies within a single region.
- Object: A segment contains a single object. The distinction between segment and object exists for two reasons: one technical, one terminological. The technical reason is that the segment sizes encoded in guarded pointers have limited resolution, and thus segments may have to be larger than the objects they contain. The terminological reason is that it is convenient to be able to say that when garbage collection copies data from one segment to another, that the contained object has *moved* even though the segments obviously have not.

Each node has a designated “allocation area” into which new objects are allocated. When the allocation area becomes full, and GC is unable to reclaim space, it is replaced with a new area.

6.5.2 SFA-aware Indirect Reference Counting

For inter-node garbage collection, Mesarthim employs the SFA-aware indirect reference counting scheme described in Chapter 5.

Mesarthim simulates an IRC bookkeeping cache in the network interface; this cache provides the measurements used in Sections 5.3.1 and 5.3.2. A single cache is used for both incoming and outgoing transactions. As with the SFA translation cache, an obvious opportunity for future study is to study using distinct caches for incoming and outgoing translation.

Within a node, IRC entries are owned by individual areas. However, for most purposes, the inter-node messaging treats each node as containing a single area. In particular, IN and OUT entries only record which node they received a pointer from; this avoids the need to consume network bandwidth by accompany every inter-node pointer transmission with the identity of its source-area.

6.5.3 Node-local GC

Mesarthim's node-local garbage collection is based on a combination of three garbage collection mechanisms: Baker's incremental copying collector [3], card marking [65], and SFA-aware IRC. Its design is inspired in part by the GC strategies of ORSLA [7] and the Symbolics Lisp Machine [48].

Area GC

Individual areas are garbage collected using Baker's incremental copying collector. Two factors led me to employ this algorithm. First, in a parallel system, an individual node which asynchronously stops to perform garbage collection can become a bottleneck as other nodes stall while waiting for it to begin responding to their requests [68]; an incremental collector, which nominally minimizes GC stalls, seemed likely to avoid this problem.

Second, Baker's incremental algorithm migrates objects into copyspace in the order that they are referenced; this tends to produce improved locality of reference for the working set of objects, a particularly desirable goal in an SPLV system.

Mesarthim provides hardware support for Baker's incremental algorithm. Baker's algorithm relies on a read barrier which detects when a pointer to oldspace is read from memory. Mesarthim features a hardware region-cache which records, for each region, the boundary between oldspace and current space. Since the upper bits of an address encode its region, this cache can be used as a "lookaside" buffer to check, for each such pointer, whether it is oldspace or not.

Inter-area pointer tracking

Since areas are independently garbage collected, Mesarthim provides mechanisms for tracking inter-area pointers. Mesarthim employs two separate mechanisms: a simple, low-overhead mechanism for tracking inter-area pointers within physical memory ("in-core" inter-area pointers), and a more complex mechanism for tracking inter-area pointers from secondary storage.

Card-marking [48, 65] used to track in-core inter-area pointers. Each page is divided into four cards. When a pointer to one area is written into a card belonging to another area, the card is marked as dirty. When any area performs local garbage collection, it must sweep every dirty card in other areas for incoming pointers.

Mesarthim provides three mechanisms supporting the card-marking write-barrier. First, each page table entry contains a set of dirty bits for the page's cards. Second, each page table entry identifies the area to which the page belongs. Third, each entry in the region cache identifies the area to which a region belongs. The write barrier uses these mechanisms as follows: the area of the pointer being written is looked up in the region cache, while, in parallel, the area of the page being written to is looked up in the page table. If the areas differ, the barrier sets the dirty bit for the card being written into.

Inter-area pointers stored to disk are handled differently. Before a page is written to disk, any dirty cards on the page are swept for inter-area pointers. For each inter-area pointer found, an IRC OUT entry is made (if one doesn't already exist) in the page's area; its parent in the copy tree is another area on the same node which could have provided the pointer — i.e. either the pointer's home area, or another area which has an IRC OUT entry for the pointer. Note that these *intra*-node copy-tree edges specifically identify parent *areas*, whereas the *inter*-node edges identify only parent *nodes*.

Thanks to this two level strategy, garbage collection on an individual area does not require paging in parts of other areas; inter-area pointers in-core are denoted by marked cards, while pointers on disk are recorded as part of the area's set of IRC entries.

Multiple areas on a node may possess IRC entries for the same object or facet. Each area maintains its own table of IRC entries, and the node maintains a combined table in order to service inter-node bookkeeping requests.

Area deactivation

One problem with indirect reference counting in general is that the number of IRC entries in each node is unbounded. To avoid keeping all entries stored in the active table at all times, I have developed a scheme for “deactivating” areas.

In brief, when an area’s data and IRC entries have gone unused for a period of time longer than some threshhold, the data is all paged out, and the IRC entries removed from the node’s combined IRC entries table.

During deactivation, steps are taken to maintain the system invariant that IRC entries in deactivated areas specifically record their IRC parents by area. This invariant is desired because it means that deactivated entries do not require their IRC parent entries to remain in a node table.

To make this work, each IRC entry maintains separate counts for children that know its area specifically, and children that only know the node upon which it resides. The latter class of children may be freely transferred among active IRC for the same object without effect. When deactivating an IRC entry with the latter class of children, those children are transferred to a still-active area’s IRC entry for the same object. (A new entry may need to be created.)

Also, as each IRC OUT entry is removed from the node table, its IRC parent field is examined. If the field names a specific area, no more work needs to be done. If the field names only a node, then a message is sent to that node requesting a specific area to record as the parent; this message will result in the reclassification of the deactivating OUT entry as a child which knows the precise area that is its parent. Over time, this reclassification will allow all IRC entries for a particular long-lived, seldom-used object to be deactivated; such an object will then consume no primary memory resources for data or meta-data (until it is next referenced.)

6.5.4 Qualitative Evaluation

Mesarthim's overall garbage collection strategy meets the first requirement of any GC strategy: it works.

The joy of inter-node GC

The inter-node garbage collection strategy, SFA-aware IRC, is simple and straightforward to implement. The initial implementation worked flawlessly with a minimum of tinkering.

The trouble with node-local GC

Unfortunately, Mesarthim's node-local garbage collection is not as successful. It suffers from three basic problems.

First, as is obvious from the description in the preceding section, local GC is quite complex; the ongoing interactions between nodes and areas while performing incremental GC present many opportunities for subtle bookkeeping errors. Developing a correct implementation was a taxing, time-consuming process: many an error only manifested itself after hours of simulation, leaving few clues as to its moment of origin. Ultimately, a complete simulation checkpointing system was implemented in order to be able to replay bugs on a relatively short time scale.

The second problem with Mesarthim's node-local GC is that it is inefficient; it often performs more heap memory accesses than the actual applications do. Without going into the innumerable motivating details, the fundamental problem is that whenever a remote pointer or an SFA pointer is written into a card, the card must be dirtied. Combined with intra-node, inter-area pointers, the result is that an inordinately large percentage of cards tend to be dirtied. Every dirty card must be swept by every GC pass. Thus, GCing an area which only occupies one quarter of physical memory may, and often does, require scanning the majority of physical memory.

The third problem with Mesarthim's node-local GC is that at the end of the day, it fails at the goal of being non-interrupting. In order to guarantee termination of a pass, Mesarthim sweeps all dirty cards atomically at the end of a GC pass; if other processes were allowed to continue operating, more cards can be dirtied, thus preventing GC from completing.

Potential solutions exist for the second and third problems. However, exploring them is made difficult by the first problem: a complex system is, in this case, also a brittle system; seemingly trivial changes can introduce faults which require days to track down and fix.

Based on my experiences with Mesarthim's local GC, I would offer the following recommendations for future implementations of this type of system.

- Garbage collect all active areas on a node as if they were a single unit. This eliminates the need for card-marking. This does increase the cost of deactivating an area, however, as the entire set of active areas must be swept for pointers into the outgoing area.

  This suggestion mirrors the strategy for dealing with "cabled" areas suggested in [7].

- Use a monolithic garbage collection algorithm, e.g. stop-and-copy, for node-local GC. Follow the recommendation of [68]: synchronize all nodes so that they perform local garbage collection simultaneously; this prevents individually GCing nodes from stalling other, working nodes.

  This change eliminates many of the tricky issues associated with incremental GC in a multithreaded, distributed environment, e.g. synchronization while migrating an object, remote writes into memory which has already been swept, simultaneous modification of IRC entries by the garbage collector and the network interface, etc.

  This change reduces the runtime overhead of garbage collection. Read traps, in particular, must inevitably incur a high overhead in modern processors due to flushing the processor pipeline; this change eliminates the need for such traps entirely.

  There are two drawbacks to this change: it introduces GC pauses that an incremental GC approach might avoid, and it does not provide the same locality benefits of Baker's algorithm. However, the reduced complexity, and the correspondingly reduced brittleness of implementation, should enable more experimentation and performance tuning that could offset these drawbacks.

6.6 High-level programming

In order to be able to write applications of nontrivial size, I implemented a very simple compiler for a Scheme-like language I call TupleScheme, abbreviated TScheme. TScheme extends standard Scheme [29] with several mechanisms. Some, such as deconstructors, are primarily a convenience. Others, such as immutable tuples and capturing-lambdas, are critical in enabling efficient parallel programs.

I will not discuss the entirety of TScheme here, but I will briefly describe three key features provided for efficiently programming a DSM machine.

6.6.1 Immutable closures

To begin with, closures, once constructed by lambda or capturing-lambda (see below), are immutable objects; thus, there are never inter-node pointers to closures. This ensures locality when a closure is applied, regardless of where it is originally created.

6.6.2 Capturing-lambda

In Scheme, the body of a lambda expression may refer to variables defined in an environment outside of the lambda itself. However, when a closure is created on processor A, but invoked on B, these bindings can be a source of inefficiency as all references to these variables turn into remote memory accesses. The special form capturing-lambda, abbreviated c/lambda, addresses this problem.

`(c/lambda` *(arg0 arg1 ...) (pass-name0 ...) (body... free-var0 .. free-var1...)*`)`

When a c/lambda expression is encountered, a closure is constructed. The procedure body may contain variables which aren't bound within the body. In addition to recording the enclosing environment, the closure records the current value in the environment of the free variables in the procedure body; I call these variables captured variables.

When a closure is applied, references to captured variable references in the body refer to the copied values in the closure object, rather than the original values in the environment in which the closure was constructed. Thus, the executing body need never again refer to the bindings of variables in the defining environment. This has important performance benefits when a closure is constructed on one node, but copied to and then invoked on another.

pass-name0, etc. specify free variables which should not be captured, but instead should be treated as would free variables in a regular lambda would be.

6.6.3 named-capturing-lambda

named-capturing-lambda, or nc/lambda, provides a means of naming a closure in a captured fashion — otherwise, a tail-recursive closure invoked on a different node than it was constructed on might have to refer back to its constructing node every time it needed to look up its own name and call itself.

`(nc/lambda` *name (arg0 arg1 ...) (pass-name0 ...) (body... free-var0 .. free-var1...)*`)`

Much like a named let, this syntax binds name within the body of the procedure to the procedure itself.

Chapter 7

Conclusions

In this chapter I briefly review the major research contributions of this thesis, then conclude with a high-level summary of the impact.

7.1 Contributions

7.1.1 Sparsely Faceted Arrays

As the most important contribution of this thesis, I have defined Sparsely Faceted Arrays (SFAs), a new data parallel data structure. A sparsely faceted array names a virtual global array, but facets on individual nodes are allocated lazily. As a result, SFAs present extremely simple shared-memory semantics, while making efficient use of memory. Additionally, since an SFA is referenced within a node by a pointer to its local facet, intra-node references proceed at full memory rate – no additional indirections are incurred.

SFAs enable the efficient implementation of explicitly-placed, non-uniformly distributed objects and data structures on an important class of parallel architectures. To drive home their importance, I have described the specific application of SFAs in implementing the quicksort algorithm and a distributed, replicated version of a Kd tree data structure.

Hardware support evaluation

In order to be implemented as a true shared-memory mechanism, SFAs require hardware support for translating between global and local names. This support is most naturally provided at the network interface on each node. Although the logic for accessing a translation table in memory is simple, we would prefer to avoid the latency of unnecessary memory accesses, and thus I have suggested the use of a dedicated translation cache.

Using a high-level simulation, I have evaluated the effectiveness of such a cache, empirically demonstrating that for several applications run on a 256-node machine, a 32-entry cache achieves a nearly optimal translation hit rate. The specific hit rate is highly application dependent; applications which build an SFA-based data structure, then use it repeatedly, have much higher hit rates than those which build and discard SFAs in rapid succession. With the 32-entry cache, different applications in my experiments demonstrate hit rates varying from 79% to 99%, with the majority of misses being mandatory due to the allocation of new SFAs. Although a hit rate of 79% is far from perfect, it is more than sufficient to decrease the expected translation latency in a practical implementation. Due to the simplicity of the applications and the idealizations of the simulation framework, these results are not definitive; rather, they merely suggest that a reasonably-sized cache can handle the vast majority of SFA translations.

The GC imperative

Although the high-level semantics of SFAs are quite simple, the underlying lazily-allocated data structure is inherently complex; automatic management, including garbage collection, is essential to preserving the simplicity of the mechanism presented to the parallel programmer. This need leads to the other major contributions of this thesis

7.1.2 Evaluation of precise parallel garbage collection

The traditional approach to garbage collection in parallel computers has been based on parallelizing precise, tracing garbage collectors, e.g. mark/sweep; by contrast, the usual approach to garbage collection in distributed systems has been based on conservative garbage collectors, e.g. reference counting.

As another of my contributions in this thesis, I have shown that in the worst case, precise tracing garbage collectors can eliminate the effective parallelism gains on an arbitrarily large parallel architecture. I have argued that bad cases are likely to crop up in practice. I have also shown that reference counting does not suffer from this problem.

7.1.3 An SFA-aware, scalable parallel garbage collection strategy

My third major contribution is a scalable parallel garbage collection strategy capable of managing sparsely faceted arrays. My strategy extends a reference counting strategy to perform inter-node pointer tracking, and is therefore conservative with respect to inter-node references; it operates in conjunction with an arbitrary, presumably precise, node-local garbage collection system.

My strategy meets two key requirements for managing sparsely faceted arrays: correctness, which requires that no facets be garbage collected until there are no live references to the SFA anywhere in the system; and efficiency, which requires that freeing an SFA does not require communication with nodes that never received pointers to the SFA.

I meet these requirements by extending Indirect Reference Counting [52] to incrementally reverse the fan-in copy-tree structure built by pointer distribution. This reversal constructs a fan-out *anchor-tree*; by the time an SFA has become garbage, its home node is the root of an anchor tree which identifies all facet-bearing nodes.

This SFA-aware garbage collection strategy is intended to track inter-node pointer references, cooperating with a node-local garbage collection strategy to provide overall system garbage collection. The strategy requires various bookkeeping operations due to each inter-node pointer copy — both immediate (e.g. reference count increment), and delayed (e.g. reference count decrement and possible anchor-tree manipulation.)

The logic of these operations is generally quite simple for the network interface to perform, and unlike SFA translation, latency is not a pressing concern, as such bookkeeping can be pipelined without delaying message transmission or reception. However, it is still desirable to avoid the unnecessary consumption of memory bandwidth, and so I have run simulations to evaluate the effectiveness of dedicated hardware caches. For different applications I have studied, caches of 64 to 128 entries provide hit rates approaching the ideal possible — but the overall hit rate ranges from around 70% to only a bit over 90%. Most operations on an IRC entry are incremental adjustments that can be recorded in a write-back fashion; thus, misses do not necessarily require a read from memory. However, in system running at steady-state, each miss which creates a new entry must spill an old one to memory.

Whether the memory bandwidth saved by a GC bookkeeping cache is worth the additional design complexity will be entirely dependent on the characteristics of a specific implementation.

7.1.4 Additional contribution

In addition to the major contributions detailed above, this thesis makes an additional minor, yet noteworthy, contribution.

A novel, conservative, impractical GC algorithm

In reaction to the problem of precise parallel garbage collection, I have developed a novel garbage collection algorithm based on a parallel connected components algorithm. As with typical parallel connected components algorithms, this GC algorithm runs in polylogarithmic steps.

In general, this new algorithm is extremely conservative, and may often fail to free garbage objects which hold pointers to live objects. However, I also show that, given an oracle, the algorithm becomes precise. I leave open the question of whether there exists a heuristic which can approximate an oracle sufficiently well to get good performance in practice.

Because I have not conceived of an adequate heuristic, this algorithm remains so conservative as to be highly impractical, and I have not attempted to implement it in this thesis.

7.2 Summary

At a high level, the entirety of this thesis contributes to a single goal: raising the level of abstraction available to the programmers of an important class of parallel architectures. Overall, the result of raising the abstractions available to the programmer simplifies the task of parallel programming.

SFAs and garbage collection provide important abstractions. Implemented in a large-scale parallel system, these tools will enable the straightforward use of fine-grained distributed/partitioned objects and partitioning algorithms, both of which are ill-supported by existing mechanisms.

Appendix A

Review of Indirect Reference Counting

In order to make this thesis self-contained, this chapter contains a review of *indirect reference counting* (IRC), a distributed garbage collection technique first reported by Piquer in [52]. Like many other distributed GC strategies, IRC is designed to keep track of references between nodes or other logical areas which also undergo independent local garbage collection.

The main idea behind IRC is this: each area maintains a reference count for each pointer it has sent to another area, and remembers from which area it received each foreign pointer. For each pointer, this generates a copy-tree or diffusion-tree of reference counts; the tree's shape mimics the pattern in which the pointer is initially distributed. As copies of a pointer are destroyed, the leaves of the tree prune themselves and notify their parents, resulting in a collapse of the tree which mirrors its construction both temporally and in terms of communications pattern. Once the reference count at the root hits zero, area-local memory management has complete control over the object.

In the following subsections I review IRC in more detail. Section A.1 describes the construction of an IRC copy-tree;, Section A.2 describes the deconstruction of an IRC copy-tree; Section A.3 discusses the handling of a few corner-cases; finally, Section A.4 discusses some of the key benefits of indirect reference counting.

The interested reader is referred both to the original work on indirect reference counting [52], and the related papers [53, 54, 55] on other indirect GC schemes.

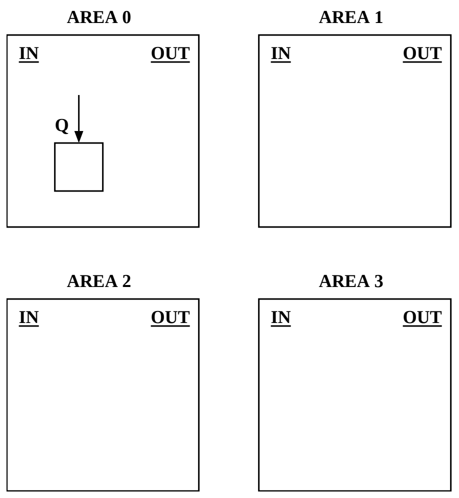


Figure A-1: Object Q allocated in Area 0.

A.1 Counting up

Consider the example in Figures A-1-A-3.

In Figure A-1, an object Q is allocated in Area 0. Note that each area maintains an IN set and an OUT set; entries in the IN set correspond to pointers coming into the area pointing at local objects, while entries in the OUT set correspond to pointers pointing out of this area to remote objects.

In Figure A-2, a pointer to Q is copied from Area 0 to Area 1, resulting in entries in the OUT set on Area 0 and the IN set on Area 1.

The IN list record format is simple: the object identifier is followed by the count of how many times its pointer has been sent to a remote area. The OUT list record is only slightly more complex: the object identifier is followed by the identity of the area from which the pointer was initially received, and then by a reference count of the number of areas to which the pointer has been subsequently sent. In this case, one copy of the pointer has been sent out of Area 0, and no copies of the pointer have been sent out of Area 1, so the IN entry on Area 0 is Q:1, and the OUT entry on Area 1 is Q:0:2.

In Figure A-3, a pointer to Q is copied from Area 1 to Areas 2 and 3; the reference count in Area 1's OUT entry for Q is incremented twice to record the transfers, and each

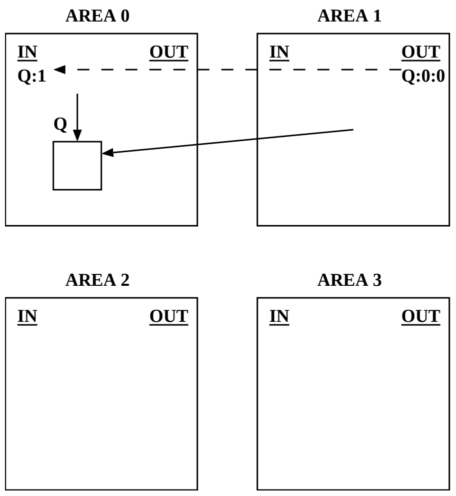


Figure A-2: Pointer to Q copied from Area 0 to Area 1. The copy-tree links are shown as dashed lines.

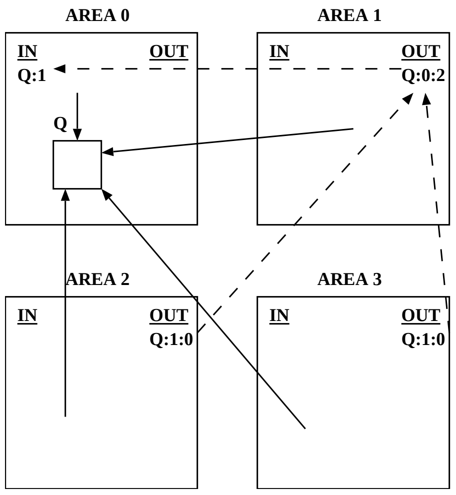


Figure A-3: Pointer to Q copied from Area 1 to Area 2 and 3.

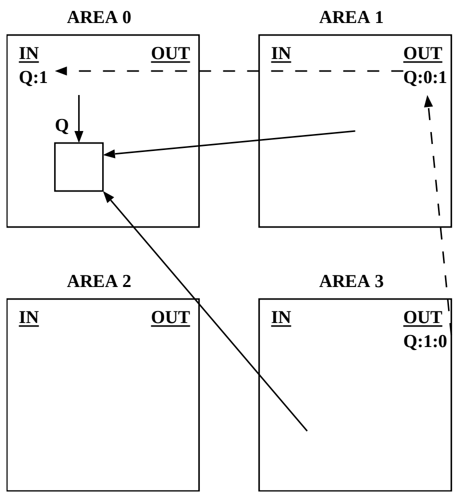


Figure A-4: All pointers to Q in Area 2 have been destroyed.

of Areas 2 and 3 record the fact that they received the pointer from Area 1.

Note that constructing the IRC tree requires no communications overhead – its construction results from the transmission of pointers from one area to another. In particular, unlike conventional reference counting schemes, no special communications are needed with Area 0, even though the copied pointer refers to an object stored there.

A.2 Counting down

Let us now examine how the IRC tree of the previous section collapses as local copies of pointers to Q are destroyed.

In Figure A-4, all pointers to Q in Area 2 have been destroyed; the entry for Q in Area 2's OUT set has been removed, and Area 1's reference count for Q has been decremented. This operation requires one message to be sent from Area 2 to Area 1 containing the decrement order.

Area-local garbage collection is responsible for discovering that all pointers to Q in Area 2 have been destroyed;. Such a discovery does not have to be immediate; as long as the discovery is made eventually, the reference counts will be decremented appropriately.

In Figure A-5, all pointers to Q in Area 1 have been destroyed. When this fact is

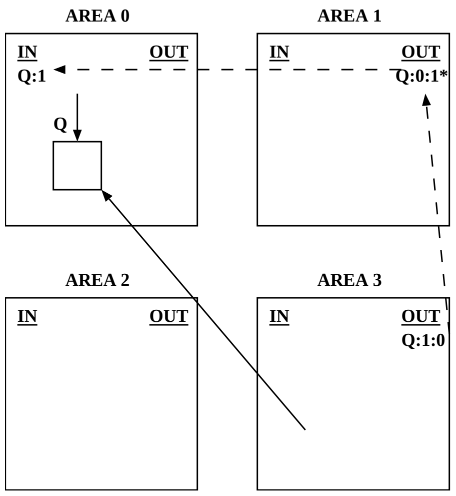


Figure A-5: All pointers to Q in Area 1 have been destroyed.

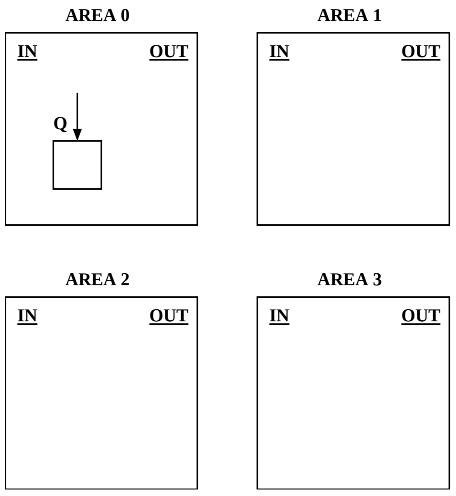


Figure A-6: All pointers to Q in Area 3 have been destroyed.

discovered, the OUT entry for Q is flagged, but since its reference count is non-zero, it is not deleted, and no messages are sent.

In Figure A-6, all pointers to Q in Area 3 have been destroyed. The decrement message sent from Area 3 to Area 1 allows 1 to finally clear its OUT entry and send a decrement message to Area 0 which can in turn clear its IN entry.

Overall, one decrement message is sent for each initial inter-area pointer-copy between the same two areas in each case; multiple decrement messages from one area to another may, of course, be batched into a single, larger message, but the overhead remains the same: $O(1)$ for every inter-area pointer-copy.

A.3 Corner cases

A.3.1 Receiving the same pointer multiple times.

Consider the hierarchical reference tree in Figure A-3. The question arises: in the course of continuing computation, what happens if a pointer to Q is copied from, say, Area 2 to Area 3, or even from Area 1 to Area 3 a second time?

The answer is straightforward: as soon as the copy arrives at Area 3, Area 3 looks up the pointer in its OUT list and finds that 3 has already received a copy; Area 3 then immediately sends a decrement message back to the source area. This ensures that there is at most one path in the copy-tree between any area and the area where Q is stored.

Once again, one decrement message is sent for each inter-area pointer copy – there is simply no delay between the copy and the decrement when the receiving area has previously received a copy of the pointer.

A.3.2 An object with only remote pointers.

It is possible for an object to be reachable only from remote pointers. In this case, the fact that remote pointers exist is recorded by the object's entry in the area's IN set. Area-local garbage collection must therefore use the IN set as (part of) the root set.

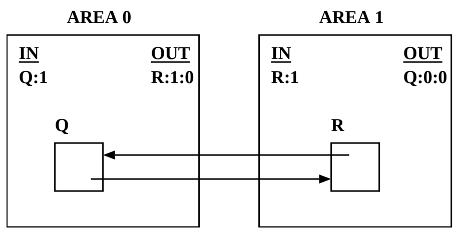


Figure A-7: Objects Q and R form an inter-area cycle.

A.3.3 Inter-area cycles.

Indirect reference counting suffers from the classic reference counting problem: the reference counts on inter-area cycles such as shown in Figure A-7 will never go to 0, even though neither object is really reachable anymore. Thus, in order to identify and eliminate garbage cycles, an additional garbage collection mechanism must be employed.

A.4 Benefits Summary

Indirect reference counting offers several properties which are beneficial in a distributed/parallel environment. I mention them here with a minimum of discussion.

- O(1) space overhead per pointer per area. Each pointer an area sends or receives requires only one IRC IN or OUT entry, and the entry itself is of fixed size, containing as it does a reference count rather than a list of children. By comparison, reference-listing schemes maintain entries identifying each of their children, thus bounding the per-area space overhead due to a single pointer at O(N), given N areas.

- O(1) GC work overhead for every inter-area pointer copy, regardless of the longevity of the copied pointer. This contrasts with tracing schemes such as mark/sweep where the GC cost per inter-area pointer is proportional to the number of GC passes through which the pointer survives.

- No additional synchronization requirements. In conventional distributed reference counting [38], bookkeeping for a single inter-area pointer transfer can involve three areas: the source, the destination, and the area into which the pointer points. This imposes synchronization requirements to function correctly in the presence of, say, network delays between nodes.

- The messaging pattern associated with IRC tree collapse is the reverse of the pattern of tree formation; thus, if a pointer is distributed via a fan-out tree pattern, the IRC tree will form and later collapse along the same tree pattern. By contrast, although weighted reference counting [6, 70], avoids the synchronization issues of the previous point, all messages associated with the deletion of a pointer must be sent to the pointer's home area; this can create a hot-spot.

- Conservative in the case of node failure or system partition. If an area in an IRC tree becomes unreachable, any object to which the area, or any of its children in the IRC tree, holds a pointer will be preserved indefinitely, since an unreachable area will never send a decrement. However, IRC trees which do not pass through the missing areas will not be affected, and garbage collection may continue uninterrupted elsewhere as a result.

Bibliography

[1] Gail Alverson, Preston Briggs, Susan Coatney, Simon Kahan, and Richard Korry. Tera hardware-software cooperation. In *Proceedings of Supercomputing 1997*, November 1997.

[2] Atlantic Aerospace Electronics Corporation. *DIS Stressmark Suite*, July 2000.

[3] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[4] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The SCHEME-81 architecture – system and chip. In *Conference on Advanced Research in VLSI*, pages 69–77, January 1982.

[5] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14:239–256, February 1992.

[6] D. I. Bevan. Distributed garbage collection using reference counting. In *PARLE '87 Parallel Architectures and Languages Europe*, pages 176–87, June 1987.

[7] Peter B. Bishop. *Computer Systems With A Very Large Address Space And Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, May 1977.

[8] G. E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[9] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the 6th International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASP-LOS VI), pages 319–27, October 1994.

[10] Andrew Chien. *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. MIT Press, 1993.

[11] H. Corporaal, T. Veldman, and A. J. van de Goor. An efficient, reference weight-based garbage collection method for distributed systems. In *PARBASE-90 International Conference on Databases, Parallel Architectures and Their Applications*, pages 463–5, March 1990.

[12] Inc. Cray Research. Application programmer's library reference manual. Technical Report SR-2165, Massachusetts Institute of Technology, 1994.

[13] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: algorithms and applications. Second edition.* Springer, 2000.

[14] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7):403–12, July 1974.

[15] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pages 146–156, November 1995.

[16] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: The design of a parallel functional language implementation on workstations. In *Proceedings of the first ACM international conference on functional programming*, pages 169–79, May 1996.

[17] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Application*, 8((3/4)):165–416, 1994.

[18] MPI Forum. MPI-2: Extensions to the message-passing interface. Technical report, june 1997.

[19] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[20] Leonard Gilman and Allen J. Rose. *APL: an interactive approach. 3rd ed.* John Wiley, 1974.

[21] Benjamin Goldberg. A reduced-communication storage reclamation scheme for distributed memory multiprocessors. In *Proceedings of the fourth conference on hypercubes, concurrent computers, and applications*, pages 353–9, March 1989.

[22] E. J. Hartung. *Astronomical Objects*. Cambridge University Press, 1984.

[23] Waldemar Horwat. Revised Concurrent Smalltalk manual. Technical report, MIT Concurrent VLSI Architectures Group, April 1993.

[24] Waldemar Horwat, Andrew Chien, and William J. Dally. Experience with CST: Programming and implementation. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.

[25] Waldemar Horwat, Brian Totty, and Willian J. Dally. COSMOS: An operating system for a fine-grain concurrent computer. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 452–476. MIT Press, 1993.

[26] John Hughes. A distributed garbage collection algorithm. In *Functional Programming and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*, pages 256–72, September 1985.

[27] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Dynamic Memory Management*. John Wiley & Sons, 1996.

[28] David R. Karger, Noam Nisan, and Michal Parnas. Fast connected components algorithms for the EREW PRAM. *SIAM Journal on Computing*, 28(3):1021–1034, 1999.

[29] Richard Kelsey, William Clinger, and Jonathan Rees, editors. *Revised^5 Report on the Algorithmic Language Scheme*. Februrary 1998.

[30] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[31] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, , and John Hennessy. The stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[32] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In *12th International Conference on Distributed Computing Systems*, pages 708–15. IEEE Computer Society, June 1992.

[33] Bernard Lang, Christian Queinnec, and Jose Piquer. Garbage collecting the world. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–50, January 1992.

[34] Stéphane Lavallée, Richard Szeliski, and Lionel Brunie. Anatomy-based registration of three-dimensional medical images, range images, X-ray projections, and three-dimensional models using octree-splines. In Russel H. Taylor, Stéphane Lavallée, Grigore C. Burdea, and Ralph Mösges, editors, *Computer-Integrated Surgery: technology and clinical applications*, pages 115–143. MIT Press, 1996.

[35] Fabrice Le Fessant, Ian Piumarta, and Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. In *Programming language design and implementation*, pages 152–61, May 1998.

[36] F. Thomson Leighton. *Introcution to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, 1992.

[37] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance.

In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 1992. ACM.

[38] Claus-Wener Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Proceedings of the 1986 ACM conference on lisp and functional programming*, pages 343–350, August 1986.

[39] Henry M. Levy. *Capability-based computer systems*. Digital Press, 1984.

[40] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[41] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. T. Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, 1992.

[42] *The Essential *LISP Manual: Release 1, Revision 7*. Thinking Machines Corporation, July 1986.

[43] Umesh Maheshwari. *Distributed garbage collection in a client-server, transactional, persistent object system*. Master of engineering thesis, Massachusetts Institute of Technology, February 1993.

[44] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Principles of distributed computing*, August 1995.

[45] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. Technical Report MIT/LCS/TR 699, Massachusetts Institute of Technology, October 1996.

[46] John M. Mellor-Crummey. Experiences with the BBN Butterfly. In *Proceedings of the 1988 COMPCON*, pages 101–104, February 1988.

[47] Eric Mohr, David Kranz, and Jr. Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. Technical Report MIT/LCS/TR 449, Massachusetts Institute of Technology, June 1991.

[48] David Moon. Garbage collection in a large lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, August 1984.

[49] Luc Moreau. A distributed garbage collector with diffusion tree reorganization and mobile objects. In *Proceedings of the 3rd ACM international conference on functional programming*, pages 204–15, September 1998.

[50] Tony C. Ng. *Efficient garbage collection for large object-oriented databases*. Masters thesis, Massachusetts Institute of Technology, May 1996.

[51] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 224–235, 1993.

[52] Jose M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 Parallel Architectures and Languages Europe*, pages 150–165, June 1991.

[53] Jose M. Piquer. Indirect mark and sweep: A distributed GC. In *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 267–82, September 1995.

[54] Jose M. Piquer. Indirect distributed garbage collection: Handling object migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–47, September 1996.

[55] Jose M. Piquer and Ivana Visconti. Indirect reference listing: A robust, distributed GC. In *Euro-Par '98 Parallel Processing*, pages 610–19, September 1998.

[56] David Plainfosse and Marc Shapiro. A survey of distributed garbage collection techniques. In *Memory Management*, volume 986 of *Lecture Notes in Computer Science*, pages 211–249, September 1995.

[57] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, October 1992.

[58] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17(1):43–6, July 1983.

[59] C. Research. Cray T3D system architecture overview, 1993.

[60] Gary W. Sabot. *The paralation model: architecture-independent parallel programming*. MIT Press, 1988.

[61] Nandakumar Sankaran. a bibliography on garbage collection and related topics. *ACM SIGPLAN Notices*, 29(9):149–158, September 1984.

[62] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[63] March Shapiro, Peter Dickman, and David Plainfosse. Robust, distributed references and acyclic garbage collection. In *Principles of distributed computing*, August 1992.

[64] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[65] Patrick G. Sobalvarro. A lifetime-based garbage collector for lisp systems on general-purpose computers. Bachelor's thesis, Massachusetts Institute of Technology, 1988.

[66] Richard M. Stallman. *Using and Porting GCC Version 2*. Free Software Foundation, 1992.

[67] Silicon Graphics Computer Systems. Origin ccNUMA servers: true scalability with a difference. white paper.

[68] Kenjiro Taura and Akinori Yonezawa. An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In *Proceedings of PPOPP '97*, 1997.

[69] Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Proceedings of Supercomputing 2000*, 2000.

[70] Paul Watson and Ian Watson. An efficient garbage collection scheme for parallel computer architectures. In *PARLE. Parallel Architectures and Languages Europe*, volume 986 of *Lecture Notes in Computer Science*, pages 432–43, June 1987.