A New Approach to Parallelising Tracing Algorithms

Cosmin E. Oancea Alan Mycroft
The University of Cambridge,
Computer Laboratory, Cambridge, UK
{Cosmin.Oancea, Alan.Mycroft}@cl.cam.ac.uk

Stephen M. Watt
The University of Western Ontario,
Department of Computer Science, London, Canada
Stephen.Watt@uwo.ca

Abstract

Tracing algorithms visit reachable nodes in a graph and are central to activities such as garbage collection, marshalling *etc*. Traditional sequential algorithms use a worklist, replacing a nodes with their unvisited children. Previous work on parallel tracing is *processor-oriented* in associating one worklist per processor: worklist insertion and removal requires no locking, and load balancing requires only occasional locking. However, since multiple queues may contain the same node, significant locking is necessary to avoid concurrent visits by competing processors.

This paper presents a *memory-oriented* solution: memory is partitioned into segments and each *segment* has its own worklist containing *only* nodes in that segment. At a given time at most one processor owns a given worklist. By arranging separate single-reader-single-writer forwarding queues to pass nodes from processor i to processor j we can process objects in an order that gives lock-free mainline code and improved locality of reference. This refactoring is analogous to the way in which a compiler changes an iteration space to eliminate data dependencies.

While it is clear that our solution can be more effective on NUMA systems, and even necessary when processor-local memory may not be addressed from other processors, slightly surprisingly, it often gives significantly better speed-up on modern multi-cores architectures too. Using caches to hide memory latency loses much of its effectiveness when there is significant cross-processor memory contention or when locking is necessary.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Processors—Memory Management; D.1 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Design, Performance

1. Introduction

The tracing of a graph of objects lies at the heart of many processes in computing, from garbage collection to web-page ranking. Improvements in the ability to trace graphs efficiently can therefore have a significant impact across a number of areas. The cost of tracing with a single processor in a flat memory model is well understood and has readily achieved lower bounds. Matters are less clear, however, in a more sophisticated model. Parallel hardware and multi-level memory hierarchies are now the norm. Moreover, we can expect both the degree of parallelism and the memory architecture complexity to grow over time. We are therefore interested in how parallel tracing of a graph of objects can be performed in a scalable manner in multi-level or heterogeneous memory.

The cost of tracing a given graph depends on how the objects are placed in memory, the order the objects are examined and by which processor. We consider the tracing problem where the objects have already been placed in memory. While memory management is the primary motivation of this work, we do *not* claim immediate significant improvements to state-of-the-art garbage collectors whose optimization depends on a number of factors. We rather take the view that tracing of some form or another is a problem in all collectors and understanding its scalability is important for the future. We have nevertheless evaluated our tracing strategy in the context of an existing, mature semi-space copying collector on commodity hardware and have seen good scalability.

To frame our approach, we find it useful to start with an abstract definition of tracing:

- `mark` and `process` any unmarked `child` of a marked node
- repeat until no further marking is possible.

The `mark`, `process`, and `child` are generic functions that operate on exactly one node at a time, and we assume that an initialisation phase has already marked and processed some root nodes. Descending one step on the abstraction scale, without losing generality, one can implement the above implied fix-point via `worklists`:

- take a node from an arbitrary `worklist`; if unmarked then `mark` it, `process` it, and add any unmarked `child` to one or more arbitrary `worklists`, where arbitrary stands for not yet fixed to a particular implementation rather than random
- repeat until all `worklists` are empty.

The next step in refining the algorithm binds the worklist's semantics. We identify an important *divergence* point, related at a high-level to code-data duality, in which only one direction seems to have been satisfactorily explored. Should worklists model primarily processing or data-placement, and are the two dual in practice?

The classical approach assigns to worklists the natural processing semantics: since they hold to-be-processed nodes they should relate to the computational/processor space. It follows that a sequential algorithm employs one worklist. Parallelising the algorithm in a straightforward manner implements the worklist as a shared data structure that can be safely accessed, via locking, by different processors. Optimising this shared access usually leads to assigning one worklist per processor, while load balancing is achieved by allowing processors to steal nodes from neighbours' worklists — for example, the double-ended queue of Arora *et al*. (1) or the idempotent "work tealing" mechanism exhibiting minimal locking of Michael *et al*. (22) The problematic synchronisation, which cannot be optimised, is the one that may appear inside the `process` function: for example a parallel copy-collector needs to ensure that an object is not concurrently copied twice. This synchronisation is particularly frustrating since, although necessary for safety reasons, such sharing conflicts arise very rarely in practice.

The main contribution of this paper is to investigate in the parallel context *the other direction* in which worklists are associated with the memory rather than processing space. More precisely, under a convenient memory partitioning, a worklist stores the to-be-processed elements belonging to exactly one partition, as in (10). Now objects are processed in a different order that implicitly exploits data locality. We impose the invariant that one partition may be processed by at most one processor, its owner, at any given time, although ownership may dynamically change to enhance load balancing. With this refinement alone, our direction seems to be just the dual of the classical approach: we obviate the need for synchronisation inside `process` – since data is not shared, but only at the similarly expensive cost of locking concurrent access to worklist[1]. We are not truly stuck, however: we merely need to allow nodes to be effectively forwarded among processors – worklists are now non-shared. To our knowledge, only Chicha and Watt (10), Demers *et al*. (11) and Shuf *et al*. (24) have explored this direction, but in the simpler *sequential* case, where load-balancing is not an issue.

Before summing up and comparing at a high-level the two approaches, we make the observation that, as hardware complexity increases, the cost of executing an instruction is less and less uniform. For example, in practice the cost of inter-processor communication – cache conflicts, locking, memory fences – continues to grow with respect to raw instruction speed (i.e. speed times number of processors times instruction level parallelism). In this context, we argue that binding worklists to memory-space semantics gains the upper hand, since it translates into a hardware-friendly behaviour as it naturally exploits locality of reference and obviates the need for locking. We thus trade-off instructional overhead for the likeliness that these execute at arithmetic speed.

As detailed in Sections 3.3 and 4.3, the mechanism for forwarding remote nodes between processors expresses a useful level of abstraction, is efficient (free of locks and expensive memory fences), is amenable to dynamic optimisations, and can adapt to exploit hardware support. For example, intensive forwarding between a source and a target worklists is optimised by transferring ownership of the target to the owner of the source.

The classical processing semantics of the worklist has the advantage of better load balancing – at the object as opposed to partition level, but compromises algorithm robustness: locking or cache conflicts are left unoptimised. Fix-up strategies, such as used in the Hertz *et al* bookmarking collector (16), which relies on an extension to the virtual memory manager to reduce page-thrashing, seem unlikely to solve the mentioned concerns. In comparison, enforcing locality of reference does not generate page thrashing to begin with.

To evaluate the effectiveness of our parallel tracing scheme, we have analysed how it improves the performance of a mature semi-space copying collector. This is described in Section 5. Throughout the paper we use copying collectors as an example of a general form of tracing; mark-and-sweep collectors will benefit less from our approach since the idempotent mark operation requires no locking. The empirical results support our high-level characterization: Not only does our tracing method usually show better absolute tracing time than the classical method on most examples with six or eight processors, it almost always shows significantly better scalability.

We summarise the important contributions of this paper:

- We explore and demonstrate the effectiveness of a memory-centric approach to parallel tracing.
- We introduce a high-level mechanism for forwarding remote nodes between processors that is efficient and free of both locks and expensive memory barriers (e.g. `mfence` on `X86`). This mechanism can be applied directly under global cache coherency and can be adapted to work on a hierarchy of caches. This step is essential in eliminating locking from the hot path.
- We present high-level optimisations to reduce forwarding, and the worklist size. The former may prove important as we move towards more heterogeneous platforms.
- Finally, we test both approaches on a range of benchmarks to demonstrate our robustness claim: our algorithm scales well in both data-size and number of processors. On the tests exhibiting scalable speed-up, our algorithm runs on average $4.44\times$ and as high as $5.9\times$ faster than the synchronisation-free sequential MMTk algorithm, on eight processors. This is comparable to average and maximal speed-ups of $2\times$ and $3.2\times$ on four processors by Marlow *et al*. (21), and with an average speed-up of $4\times$ on eight processors by Flood *et al* (14), albeit the comparison suffers due to different benchmarks and hardware.

[1] Multiple processors may need to insert in the same worklist since a node and its children do not necessarily belong to the same partition, and this synchronisation cannot be optimised via doubled-ended queues (1).

The rest of the paper is organised as follows: Section 2 reviews the classical parallel semi-space collection at a high level from our point of view, and discusses related work. Section 3 presents a simplified view of our algorithm. Section 4 enhances the basic design with a few dynamic high-level optimisations, describes implementation details, evaluates the design trade-offs, and discusses further considerations. Section 5 gives an empirical comparison of our algorithm with that of MMTk, and Section 6 concludes.

2. Background and Related Work

2.1 Parallel Copying Collector at a High-Level

A semi-space copying garbage collector (4) partitions the space into two halves: memory is allocated out of the from-space, and when this becomes full, collectors copy the live objects of the from-space to the to-space, and then flips the roles of the two spaces. Some solutions employ a partitioning of the to-space into blocks and implement the to/from space separation at a higher-semantic level, to the effect of a much reduced space overhead. We simply call all of these copying collectors.

The common technique for parallelising copying collectors is to use a shared `queue` and worker threads running identical code:

```
while (!queue.isEmpty()) {
  int ind = 0;
  Object from_child,to_child,to_obj = queue.deqRand();
  foreach (from_child in to_obj.fields()) {
      ind++;
      atomic{ if( from_child.isForwarded() ) continue;
              to_child = copy(from_child);
              setForwardingPtr(from_child,to_child); }
      to_obj.setField(to_child, ind-1);
      queue.enqueue(to_child);
} }
```

A to-space object is randomly taken from the queue, and, if its fields have not already been copied, the children are copied to the to-space, the to-space object's fields are updated, and the newly copied objects are enqueued.

This code hides two layers of synchronisation. The queue access synchronisation has been shown to be amenable to implementation with small overhead: Arora *et al*. (1) apply a double-ended queue data-structure, while Marlow *et al*. (21) and Imai and Tick (18) amortise the locking cost by dividing the to-space into blocks that are processed in parallel.

The more problematic synchronisation is that of the fine-grained, per-object locking needed when copying an object – without the `atomic` block the same object o_1 may be copied to two to-space locations with the initial o_1 references split between the two. Marlow *et al*. (21) estimate this sequential-case overhead to be around 25% of the collection run time.

2.2 Related Work

We consider parallel copying collectors that implement heap exploration by reachability from root references (20). We study copying over mark-and-sweep collection because the latter is a simpler case of tracing that does not fully highlight locking issues.

Halstead (15), in the context of Multilisp, was one of the first to employ a parallel semi-space copying collector. His design assigns each processor its own to/from spaces and allows an incremental, per-processor, object copying phase, while the semi-space swap is coordinated among all processors. We observe that this approach is perhaps the closest one to ours in spirit, since it implicitly assumes that each processor traces mostly local data: there is one worklist per processor, with worklists' semantics related to the memory-space. However the dual space is left unoptimised: as Halstead acknowledges, this approach may lead to work imbalance; also fine-grained locking is needed to synchronise from-space accesses, although contention is rare.

It is widely accepted that methods aimed at avoiding work imbalance have been a significant challenge due to the fact that in general it is impossible to determine which roots lead to small or large data structures. Dynamic re-balancing is implemented with two main techniques. One employs work stealing at a per-object granularity in conjunction with data-structures exhibiting small-locking overhead per access. The other groups work into blocks, and thus amortises the locking overhead by copying several objects for every synchronisation. These batching schemes might make termination criteria easier and, as Siegwart and Hirzel observe, may provide flexibility in expressing traversal policies (25).

The object-granularity stealing was explored by Arora *et al*. who propose a one-to-one association between processors and worklists, which are implemented via the double-ended queue data-structure (1). The double-ended queue interface exhibits three main operations: `PushBottom` and `PopBottom` are usually local and do not require synchronisation, while `PopTop` is used to steal work from other processors, when processor's own worklist is empty.

Flood *et al*. (14) present a parallel semi-space collector and a mark-compact copying collector that statically parallelise root scanning by over-partitioning the root set and employ dynamic per-object work stealing via double-ended queues. To gracefully handle worklist overflow, they propose a mechanism that exploits a class pointer header word present in all objects under their implementation. The allocation synchronisation overhead is reduced by atomically over-allocating per-processor "local allocation buffers" (LAB); each processor then allocates locally inside LABs.

Endo *et al*. (13) propose a parallel mark-and-sweep collector, in which work stealing is implemented by making processors with work copy some work to auxiliary queues. Processors without work lock one auxiliary queue and steal half of its elements. This approach makes the transition to batch-based systems, since large objects are sub-divided into 512-byte chunks.

The first *block-based approach* is Imai and Tick's parallel copying collector (18).Their approach divides the to-space into blocks, where the block's size gives the trade-off between load balancing and synchronisation overhead. Each processor scans a scan-block (from-space) to a copy-block (to-space). If the copy-block is filled, it is added to a shared pool and a new one is allocated; if the scan-block is completed a fresh one is grabbed from the shared pool.

Attanasio *et al*. (2) propose a copying collector for Java server application running on large symmetric multiprocessor platforms that reportedly scales as well as that of Flood *et al*. Load balancing is implemented by maintaining a global list of work buffers containing multiple pointers to objects, from which work is stolen.

Cheng and Blelloch's parallel copying collector (9) requires processors to push periodically part of their work onto a shared stack, which is used for work stealing. A gated-synchronisation mechanism ensures the atomicity of push and pop operations.

Barabash *et al*. employ "work packets" (5), similar at a high-level to Imai and Tick's blocks, that makes it easy to detect GC termination and provides flexibility in adding/removing processors from the system. Their design differs in that packets are shared between processors at a higher, "whole-packet" granularity and the scan-packet and copy-packet remain distinct.

Marlow *et al*. developed a parallel copying collector (21) for Haskell that, similar to Imai and Tick's work, implements Cheney's elegant technique (8) of representing the to-be-processed objects as an area in the to-space. This eliminates the overhead, found to be at least 8%, of maintaining a different structure (queue). To further reduce the work-distribution latency, they allow incomplete blocks to be added to worklists, if there are idle processors.

We observe that all the above collectors associate the worklist to the processor space, with the result that the problematic synchronisation cannot be eliminated. The only success in this direction was achieved in the context of functional languages, by exploiting the fact that most data is immutable: Doligez and Leroy (12) allow multiple threads to collect in parallel their private heaps, which contain only immutable objects, while the concurrent collector of Huelsbergen and Larus (17) allows immutable data to be copied concurrently with mutators' accesses. They do not address handling the other negative memory effects, such as page thrashing.

Several solutions come close, at least in spirit, to our approach of *binding the worklist semantics to the memory space*, but they all treat the simpler, sequential case. Chicha and Watt (10) and Demers *et al*. (11) both optimise memory hierarchy issues by partitioning the heap into regions and enforcing localised tracing inside each region. The first approach stores pending remote pointers (remote to the currently scanned region) into the "localised tracing queue" of the region the pointer addresses. The second approach, in a generational context, achieves locality of reference by limiting the tracing activity to one card of the remembered set at a time.

The work of Shuf *et al*. (24), although focused on the sequential case, also discusses the design of a memory-centric, parallel collector. The difference with respect to our approach is one of perspective: Shuf *et al*. rely on a placement technique to reduce the number of remote objects; our method does not require this regular placement. Our tests show, on average, that one object in four or five is remote, hence locking overhead corresponding to handling remote objects in Shuf *et al*. may still be significant.In contrast our forwarding queues eliminate locking from the hot execution path. Furthermore, if P and N denote the number of processors and partitions, we keep exactly one worklist per partition and $P \times P$ forwarding queues, as opposed to $P \times N$ worklists in Shuf *et al*. Since effective load balancing usually requires $N \gg P$ our approach seems to save metadata space. Finally, as opposed to the design of Shuf *et al* that allows exactly one worklist to be owned by a certain thread, our approach encourages threads to own many partitions in a producer-consumer relation. This allows us to (i) reduce forwarding, since remote objects belonging to owned partitions are directly inserted – without locking – in their associated worklists, and (ii) to copy co-referenced objects together via Cheney's trick, as described in Section 4.4.

Attardi and Flagella also partition the heap into regions but with the intent to provide flexibility in choosing the most suitable memory management scheme for each region (3). Other solutions propose fix-up strategies to reduce the cost of negative memory effects. Hertz *et al*. (16) developed the bookmarking collector that records

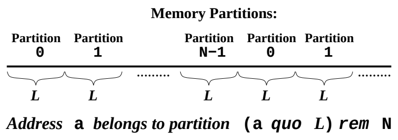

Figure 1. Partitioning memory into N regions of local size L.

summary information about outgoing pointers from to-be-evicted pages to reduce the probability of that page being reloaded again. Their implementation relies on modifications of the low-level layer to gain control over the paging system. Boehm (7), in the context of a mark-and-sweep collector, optimises (i) cache behaviour by exploiting hardware support to pre-fetch object's children in the cache, and (ii) paging behaviour during the sweep phase by using a bit per page to encode whether the page contains any live objects.

While distributed garbage collectors come close in spirit to our approach in that the memory is implicitly partitioned between remote computers, a survey study by Plainfosse and Shapiro (23) indicates a different focus. The approach is to employ a regular collector to do the local work and a special one to handle remote references. The focus is on how to represent remote references such that their local representations are not garbage collected by the "regular" collector and to study mechanisms for garbage collecting remote references that are scalable, efficient and fault-tolerant. In some ways, the attention to communication costs in this setting is similar to our memory access considerations.

Finally, we briefly recall here the Memory Management Toolkit, MMTk (6), since Section 5 compares our approach against their semi-space collector. MMTk is a research-oriented, composable, extensible and portable framework for building garbage collectors for Java in Java, that has been found to give comparable efficiency to those of monolithic solutions. Its tracing scheme, an instance of the classical approach, employs (i) work stealing at page-level granularity in which worklists are implemented by the double-ended queue data structure of Arora *et al*. (1), (ii) the LAB (14) of Flood *et al*. to reduce the locking overhead at allocation, and (iii) the special treatment of large objects (13) by Endo *et al*.

3. Simplified, High-Level Design

Our survey of parallel copying collectors shows that all are instances of the classical approach that binds the worklist semantics to the *processor* space. We present now the basic design of our algorithm that explores the dual direction – the *memory* space. This section first introduces some notations and makes some simplifying assumptions, then presents the core of the algorithm, how inter-processor communication is handled, and the termination condition. Representing inter-processor communication explicitly removes locking from the execution's hot path; alternatives move equally expensive locking from one place to another. Various high-level optimisations are left to Section 4.

3.1 Notations, Invariants and Simplifying Assumptions

We consider a heap of arbitrary size with N disjoint partitions, each containing contiguous intervals of size L. The distance between two consecutive intervals belonging to the same partition is LN so partition i contains addresses in the set $\cup_{k\in\mathbb{N}}[iL + kLN, (i+1)L+kLN)$, as illustrated in Figure 1. It follows that an address $\mathtt{a}_1$ belongs to partition $(\mathtt{a}_1 \ \mathtt{quo}\ L)\ \mathtt{rem}\ N$. A condition to have good load balancing is for N to be significantly greater than the number of processors P; this is known as over-partitioning the heap. As discussed in Section 4.4 the value of L naturally leads to a trade-off between the algorithm's locality of reference and load balancing. We improperly call L the "partition size" – a perhaps better, but more awkward, name would be "local size" since the partitions of an infinite heap have infinite size.

At this point we need to clarify *what do our worklists store*? Section 2.1 shows that the classical worklists store to-space live objects. In our approach, a worklist stores *slots*, where a slot is a to-space address that holds a pointer to a live object in the from-space. For example, for a to-space object $\mathtt{o}_1$ having only one field $\mathtt{f}_1$, the slot $\mathtt{s}_1$ to be enqueued when tracing $\mathtt{o}_1$ children is the address of $\mathtt{o}_1$'s field $\mathtt{f}_1$, which stores a pointer to a from-space object. Our collection algorithms dequeues $\mathtt{s}_1$ at a later time from a worklist, then the from-space object pointed by $\mathtt{s}_1$ is copied in the to-space object $\mathtt{o}_2$, and $\mathtt{s}_1$ is updated to point at $\mathtt{o}_2$ – thus semantically performing $\mathtt{o}_1.\mathtt{f}_1 = \mathtt{o}_2$. Finally the addresses of the $\mathtt{o}_2$'s fields are added as slots to their corresponding worklists, and repeat.

Worklists store slots because our algorithms uses a one-to-one association between worklists and partitions, and the important invariant to be preserved is that any slot $\mathtt{s}_1$ in worklist i stores an address $\mathtt{a}_1$ that belongs to partition i, i.e. $(\mathtt{a}_1\ \mathtt{quo}\ L)\ \mathtt{rem}\ M = i$.

Having defined slots, we make the observation that while the classical algorithm may result in worklists of smaller sizes than ours, in the worst case both approaches exhibit a $O(S_l)$ worklists' combined size, where S_l denotes the number of non-null slots in live objects. To see this, it is enough to observe that: (i) at most S_l slots are going to be processed, and thus enqueued in worklists, and (ii) when only one worklist is used to collect a balanced binary tree containing Q non-aliased objects, the worklist will reach size $Q/2$ in both cases.

Due to their one-to-one relation, we freely interchange the notions of collector and processor. Our design enforces that at most one collector `c` may access a worklist at any time. We call `c` the owner of that worklist and of its associated partition. We allow the case of a partition not being owned. We call *forwarded* slots of collector `c`, the slots storing addresses belonging to partitions owned by `c`, but that were reached by other collectors when tracing their partitions. We make several simplifying assumptions, which we treat at length later, in Section 4: worklists are represented as unbounded queues, no dynamic load-balancing mechanism is assumed, and roots have been already placed in their worklists and all non-empty worklists have been distributed among collectors.

3.2 Handling Localised Tracing

Figure 2 shows a simplified version of the tracing algorithm. The `run` function, which implements one collector tracing, repeatedly performs the following operation sequence: First, its forwarded objects are placed in their corresponding worklists. This is achieved by the call to `processForwardedSlots` function, whose implementation we defer to Section 3.3. Second, objects belonging to one of the partitions owned by the current collector are traced by the `processOwnedWorklists` function. This section explains this aspect. Finally, if the current collector does not have any immediate work to do, either because it owns no queues or all owned queues are empty, `test_termination` checks whether there is still work globally. If not, the collector is allowed to end. The termination condition is explained in Section 3.4.

Our simplified implementation of `processOwnedWorklists` assumes that each collector holds a list of owned worklists and `getWorkList` simply returns the current, non-empty worklist. When this becomes empty, it is removed from the owned list, added to a to-be-released list and the next non-empty worklist is chosen. To-be-released worklists are freed in `processForwardedSlots`.

To `trace` a worklist, we repeatedly remove and process its `slots`. If the slot refers to an object that has been already copied to the to-space (`isForwarded(obj)==true`) then we just update the slot reference to the to-space object. Otherwise, the object is copied to the to-space (`new_obj`), its slot's reference is updated

```
void run() {
  boolean is_work_left = true;
  while(is_work_left) {
    processForwardedSlots();
    is_work_left = processOwnedWorklists();
    if(!is_work_left) is_work_left = !test_termination();
} }
boolean processOwnedWorklists() { return trace( getWorklist() ); }

boolean trace(Worklist wq) {
  if(wq == null) return false;
  int counter = 0;
  while(counter < wq.quantum && !wq.isEmpty()) {
    Address   slot     = wq.dequeue();
    Reference obj      = slot.loadRef(), new_obj;
    if(isForwarded(obj))
        { slot.storeRef(obj.getForwardingPtr());  continue; }
    new_obj = copy(obj);
    slot.storeRef(new_obj);
    setForwardingPtr(obj, new_obj);

    for_each(Address child in new_obj.children())
        { dispatch_enq(child); counter++; }
  }
  return true;  }
```

Figure 2. High-level, simplified tracing algorithm

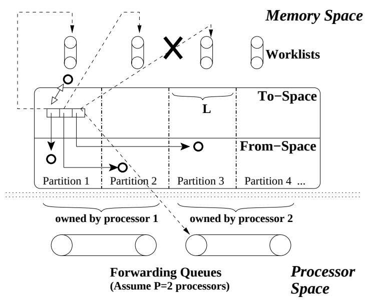

Figure 3. Dispatching slots to worklists/forwarding queues. *Arrow semantics*: double-ended = copy object to to-space; dashed = insert in queue; solid = object's fields point to from-space objects.

(storeRef), the forwarding pointer is installed to the old object (setForwardingPtr), and finally the new object's fields are dispatched to their corresponding worklists (dispatch_enq). Figure 3 depicts the dispatch mechanism: the object corresponding to the slot just dequeued from worklist 1 is copied to the to-space. Its fields point to objects in partitions 1, 2 and 3. Since each worklist knows its owner and partitions 1 and 2 are owned by collector 1, the corresponding slots are inserted directly in worklists 1 and 2. Since the third object field belongs to a partition owned by a different collector, it is inserted in the forwarding queue of the owner. If partition 3 were not owned at that moment, collector 1 would attempt to acquire ownership, using partition-level locking. A useful optimisation is to check whether the object's children have already been copied to to-space; if so just discard them after updating their (object's field) slot, thus reducing redundant inserts.

Important Remark. Consider an object o, already in to-space, that has a field f that points to the from-space object old_f. Let cp_f be the to-space copy of old_f. The slot update o.f = cp_f is still safe in our concurrent context because: (i) o.f is updated exactly once during the entire garbage collection, (ii) o has been already copied to to-space prior to dispatching its fields o.f to worklists, and (iii) field locations are word-aligned and thus updated atomically on Intel architectures starting with 486 (19). Section 4.4 discusses how to update the slot in the NUMA case.

```
volatile int tail=0,head=0, buff[F]; next : k -> (k+1)%F;

bool enq(Address slot)     {      bool is_empty()
    int new_tl=next(tail);            { return head == tail;    }
    if(new_tl == head)            Address deq()                 {
        return false;             //  while(buff[head]==null)
//  while(buff[tail]!=null)      //    isync;
//      isync;                        Address slot= buff[head];
    buff[tail] = slot;            //  buff[head] = null; lwsync;
    tail = new_tl;                    head = next(head);
    return true;                      return slot;
                           }                                    }
```

Figure 4. Forwarding queue implementation for X86. The commented lines are required for PowerPC.

3.3 Inter-Collector Communication – Forwarding Queues

Given P processors, a $P \times P$ matrix of *forwarding queues* is used so that the ij entry holds items enqueued by processor i to be dequeued by processor j. Diagonal entries are unused. We call processor i the producer and processor j the owner or consumer of queue ij. We consider forwarding queues to be circular, of fixed-size F. Associating one queue with exactly one producer and one consumer permits the free-of-locks and wait-free implementation shown in Figure 4, which complies with the Java Memory Model: all shared data is declared volatile. However, this algorithm requires only a weaker notion of volatilty, closer to the C++ one; writing this code in assembly would be more efficient.

For Intel/AMD X86 architectures (19) the only problematic memory re-ordering pattern arises when initially a=b=0 and, concurrently, processor 1 writes a=1 then reads b and processor 2 writes b=1 then reads a. Then processors 1 and 2 might find b==0 and a==0, respectively. Our code exhibits this pattern between two enq calls executed concurrently with two is_empty followed by deq calls:

```
// Processor i              // Processor j
1. buff[tail] = ...         head = next(head);
2. tail       = ...         if( head != tail)
3. if(new_tl == head)           slot = buff[head];
```

A JVM fixes this by inserting a potentially expensive mfence instruction after the writes to tail and head. We observe however that the algorithm is still correct without mfences. First, a problematic memory re-ordering between head and tail may result in processor i finding the queue full when it is not, and processor j finding the queue empty when it is not. This does not affect correctness, just delays the forwarding or processing of objects. Second, the problematic pattern cannot appear on buff. Assuming that it does then, in order for both processors i and j to access the same element in buff, we have tail==next(head) before executing line 1. There are two cases: (i) if processor j reads the value of tail updated by processor i at line 2 then, since two writes of processor i cannot be observed re-ordered by processor j, processor j must read the updated value of buff[head], (ii) otherwise processor j finds at line 2 head==tail and buff[head] is not read.

On platforms that do not enforce "total store order" (26), where two writes to different locations of the same processor can be observed in reverse order by another processor, the commented lines in Figure 4 provide *part* of the fix-up. With the original enq/deq code, it is possible that a processor performing is_empty then deq sees the queue non-empty before another one performing enq writes the only element of the queue. The result is that deq returns a garbage value and one slot might not be processed (copied), breaking the algorithm correctness. The partial fix-up enforces that all non-valid slots in buff are null. Under this weaker memory consistency, a consumer waits for valid slots to appear in a queue, and they will appear because the queue is not empty, and similarly a consumer waits for invalid slots to disappear from a queue.

The other part of the fixup requires that architecture specific instructions are added so that (i) writes become eventually visible to other threads and (ii) the read from `buff[head]` has completed before the write to `head` goes through. For example, as remarked by an anonymous reviewer, the PowerPC `lwsync` and `isync` instructions achieve this behaviour.

Forwarded slots are dispatched to their corresponding worklists by their consumer collector (owner) in `processForwardedSlots`. The dispatch manner has already been discussed and is depicted in Figure 3. Since we have considered bounded queues, we employ, for simplicity, an unbounded per-collector buffer to store slots for which forwarding failed because the corresponding queue was full. This buffer is also visited at this stage and its slots are similarly dispatched. Finally, `processForwardedSlots` releases ownership of non-empty worklists in the to-be-released list. It is necessary that these non-empty worklists contain forwarded slots. There are two possible strategies: either retain ownership and push those worklists back to the owned list, or transfer the worklist's ownership to one of the forwarding collectors (see Section 4.3).

We have seen in Figure 2 that the `trace` function processes slots from the current worklist until `quantum` local or remote slots have been produced. So how is `quantum` chosen? Empirical results suggests that effective parallelisation is achieved when the ratio of forwarded to local slots is less than $1/4$. We chose thus `quantum` $= P \times F \times 4$. On the one hand this amortises well the overhead of processing forwarded slots. On the other hand this allows us to easily identify consumer-producer relations between worklists owned by different threads: a forwarding queue is found full. In this case, Section 4.3 shows that forwarding can be optimised by transferring a worklist's ownership to the producer collector.

We conclude this section with three important observations. *First*, the use of forwarding queues is essential. The alternative of allowing collectors to concurrently insert slots into one worklist cannot be optimised – the double-ended queue does not apply here because it requires one writer and multiple readers. Also, splitting worklists in the manner of forwarding queues may waste significant space since we have many more partitions than processors.

Second, our design enforces (i) data separation – synchronisation is necessary only when acquiring ownership, and (ii) locality of reference – processing is localised at partition-level, hence for example page thrashing is unlikely to appear when L is large (10). Note that although batching-approaches (18) rely on a memory segmentation, they are fundamentally different in that they do not fully exploit neither data separation, as two concurrently processed blocks may point inside the same block, nor data locality, as their blocks ($4 - 32$K range) are smaller than our partitions (≥ 32K).

And third, the forwarding mechanism does not apply more cache-pressure than needed since the common case is that one processor communicates mostly with one other, rather than with arbitrarily many, in a time window. Furthermore, the buffering technique of processing forwarded slots reduces the possibility of cache-lines being invalidated due to concurrent accesses.

3.4 Termination Condition

A collector `c` is allowed to exit the `run` method's `while` loop when: (i) it does not own any worklist – owned and to-be-released lists are empty, and (ii) all the forwarding queues `c` may consume from are empty, and (iii) all the forwarding queues `c` may produce to are empty, and (iv) all worklists are empty. Upon exiting, `c` makes its exit status visible by setting the globally-readable `exited` field.

If condition (i) is false then either there is more work, or not all worklists have been released; the latter case will be remedied by a new execution of `processForwardedSlots`. If condition (ii) is false, then more work will be unveiled in `processForwardedSlots`.

If condition (iii) does not hold, then a collector might have exited and `c` has subsequently forwarded slots to it. If this is true, then `c` dispatches those slots as explained in Section 3.2; mutual exclusion to those slots is ensured since the forwarding queue owner has exited. It follows that forwarded slots cannot exist after both their producer and intended consumer collectors have exited.

Condition (iv), rarely tested, is necessary under a load-balancing mechanism in which non-empty worklists may be released (see Section 4.2). If a non-empty, unowned worklist is found, then `c` tries to acquire it. With the simplified version, it is an optimisation that does not allow `c` to exit while work might still be available.

We observe that, if a thread is not delayed forever, collection terminates: our approach differs at a high-level from the classical one in that it changes the traversal ordering by delaying processing of remote slots (to enhance locality), but not indefinitely so. Indeed, all reachable objects are inserted into their corresponding worklists, which are eagerly acquired, and all non-empty worklists are processed. This process terminates: (i) marking an object before copying it breaks cyclic references, and (ii) there are finitely many pointers to a live object, directly gives (iii) there are finitely many live objects inserted in worklists.

Finally, even under a load-balancing mechanism the algorithm is livelock-free once we enforce the invariant that transferred, non-empty, worklists need to be processed at least once by the new owner. This ensures that the system is making progress, hence worklists cannot be indefinitely switched between collectors. The invariant above also guarantees that objects are not indefinitely forwarded between forwarding queues: a worklist will eventually become empty, and hence not owned, and thus one collector that attempts to forward the slot will succeed in doing so by acquiring the partition – thus progress is always made.

4. High-Level Optimisations

This section refines the basic design of the previous section and presents how (i) non-empty worklists are initially distributed to collectors, (ii) dynamic load-balancing is achieved, (iii) forwarding is optimised, and (iv) worklist size is reduced. We finally discuss other potential lower-level optimisation and future work directions.

4.1 Optimising Initial Granularity

As presented in the next section, our dynamic load-balancing scheme is applied at partition-level and is thus less effective than classical approaches that steal work at object or block level. The consequence is a significant start-up overhead – corresponding to the time by which all collectors perform useful work concurrently.

To optimise this start-up overhead we employ an initialisation phase that: (i) processes in parallel a number (30000) of objects under the classical algorithm, then (ii) places in parallel the resulted grey objects to their corresponding worklists via partition-level locking, and (iii) distributes worklists among processors by iteratively assigning several consecutive non-empty worklists to each processor. The last step is similar to the static roots partitioning of Flood *et al.* (14). With this refinement, our load balancing works reasonably well on medium memory partitions ($\geq$ 512K) since it is a rare event that usually occurs to one collector at a time. This optimisation is unlikely to be effective on small memory partitions.

4.2 Dynamic Load Balancing Mechanism

We have already said that our dynamic load-balancing mechanism is employed at the partition level. We make the trivial but perhaps not obvious observation that work stealing is not compatible with our design. To see this assume collector c_1 has been preempted just before copying an object o_1 belonging to its owned partition p, and collector c_2 steals p. Observe that c_2 may acquire and copy the same object o_1, and that without a CAS instructions, c_1 cannot be prevented, upon being rescheduled, to also copy o_1.

Our scheme requires c_1 to release w ownership before c_2 can acquire ownership of w. More precisely, if c_2 is out of work then it indicates to the neighbour collectors the worklists it would like to acquire. The helper collectors, such as c_1, decide whether the requested worklist w is expandable. If so and w is not empty, c_1 releases ownership of w and places w's head slot in the corresponding forwarding queue of c_2; c_2 may acquire ownership of w when it processes its forwarded slots. In our simple implementation c_2 requires from c_1 any worklist w that is not the one c_1 currently processes, and c_1 releases it under the same condition.

We recall from Section 3.1 that a memory partition consists of a union of equidistant intervals of size L. It is important to remark that for small values of L, say less than 128K, dynamic load balancing might not be needed because grey objects will tend to be rather randomly distributed among N consecutive intervals of small size. However, for each grey object we still expect to find at least several of its children in the same interval, otherwise the forwarding overhead will seriously impact performance. Our empirical results confirm that in many cases small L's give good speed-up with static assignment of partitions to collectors.

As the interval size L grows the probability that grey objects are randomly distributed among partitions decreases, but the locality of reference increases, in that it is more likely that scanning an object will result in grey objects in the same partition. If grey objects are concentrated in sufficiently many partitions, our dynamic mechanism attempts to fairly distribute partitions with grey objects among collectors. Otherwise, collectors will starve. Imai and Tick (18) demonstrate that the block size of batching solutions naturally defines the trade-off between locking overhead and dynamic load-balancing effectiveness. In our case the partition size mediates between effective load balancing and locality of reference, since by design locking was eliminated from the hot execution path.

Although left as future work, a simple strategy to dynamically adapt L to graph-heap tracing would be to start with a "large" value, in the megabyte range, thus exploiting data locality, and to monitor the forwarding ratio, FR, and the load balancing, LB, computed as the standard deviation of the number of slots processed by the threads in a small time interval. If poor load balancing is observed then L is decreased. If $\texttt{FR} > \texttt{FR}_m$ or $L < L_m$, for empirical $\texttt{FR}_m$ and $\texttt{L}_m$, then we can switch back to an instance of the classical approach. Our experiments found $1/4$ and 32K to be good values.

Comparing our memory-centric load balancing against the processor-centric one at a high level, we observe that there are both favourable and less favourable cases. For example in the case of an array holding the starting pointers of eight lists, the classical block-based approach will uncover no parallelism since the number of grey objects at all times is eight, hence all grey objects will fit into one block and there would be no possibility to steal a block. The classical solutions that steals work at object level will achieve good speed-up on up to eight processors. Our memory centric approach would likely give some speed-up.

4.3 Optimising Forwarding

Excessive forwarding is usually a result of partitions owned by different collectors being in a circular producer-consumer relation. Figure 5 presents two such cases: the fields of objects in partitions 1, 2 and 3 arbitrarily point to objects in partitions 1, 2 or 3; while partitions 4 and 5 exhibit a more structured topology that corresponds to a list (partition 4) whose values are placed in partition 5.

Remember that our design tolerates a certain ratio of forwarded to produced slots (1/4). If this ratio is significantly exceeded, as

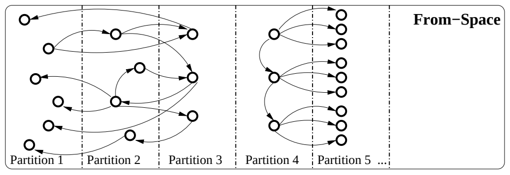

Figure 5. Producer-Consumer Relations Between Partitions. Circles depict objects; arrows depict object's fields.

is likely with the memory configuration in Figure 5, at least one collector c_1, fills the forwarding queue of another collector c_2 with slots belonging to partition w. Now c_2 may transfer w's ownership to c_1 if: (i) c_2 has not acquired during the current step ownership of a partition from c_1 – this breaks a potential transfer cycle, and (ii) the ownership transfer does not significantly affect load-balancing.

Our simple implementation assumes w to be the partition of the forwarded tail slot and, as already stated, condition (ii) holds when w does not correspond to the current worklist of c_2. Thus, partitions in a producer-consumer relation are eventually owned by the same collector, if this does not result in immediately starving another collector. Under small partitions this optimisation will probably not prove effective because of the random distribution of grey objects to partitions (see Section 4.2), and because a contiguous interval of a partition will contain relatively few live objects that would have likely already been forwarded by the time the producer-consumer relation is discovered. We predict that optimising forwarding will be important when larger partitions do not restrict load balancing (and on NUMA). Encouraging evidence in this sense is that, on a multi-core, we have found cases when this optimisation accounts for 15% speed-up on 6 processors, under a $512K$ partition size.

The consumer-producer inter-partition relation can also be exploited when partitions have the same owner. The goal is to decrease the worklist size by encouraging the processing of the producer worklist to keep pace with the processing of the consumer.

4.4 Discussion and Future Work

We have assumed our data-structures (queues, buffers) to be logically unbounded. Our implementation under MMTk reserves in advance about 5M for our algorithm's metadata, and increases this size dynamically if needed. To our knowledge MMTk also logically represents their worklists unbounded. A safety mechanism that permits bounded worklists is proposed by Flood *et al.* (14) at the cost of one extra word per object.

An architectural issue that affects our algorithm is that MMTk does not provide an inter-collector `yield` facility—since the classical approach employs a more collaborative algorithm that is not sensitive to this feature. With our approach, preempting a collector that owns many worklists will result in a significant slow-down, since transferring those worklists to starving collectors will be delayed. Thus, allowing a collector to yield in favor of any other collector is fundamental to having effective load-balancing,

While we have restricted ourselves to exploring high-level optimisations applicable to the most general form of tracing, one specialised technique warrant further attention: it seems possible to implement Cheney's trick of representing grey objects as an area in to-space without affecting load-balancing. Since tests show that generally one in four-to-six objects is forwarded, if a collector is starving, there will be enough slots in forwarding queues to indicate which partitions are non-empty and can be requested.

While, as a future direction, we intend to examine in more detail garbage collection for NUMA systems, the rest of this section provides some rationale into why we believe our algorithm is suitable and how to adapt it for NUMA platforms. *First*, we observe that re-ordering object processing to improve locality of reference is essential in this context since concurrent accesses to the same memory segment is at best inefficient (and at worst not supported[2]).

Second, forwarding queues give an elegant way to abstract and make explicit inter-processor communication. Rather than assuming that hardware can efficiently handle any memory configurations, we use caches where they are most needed – for inter-process communications. A small shared cache, or even a hierarchy of small caches still allows the forwarding queues application.

Third, we have observed at the end of Section 3.2 that it is safe for collector `c` to update an object `o` field `f` with its corresponding to-space object cp_f (`o.f=cp_f`), where `c` has created `cp_f`. However, this "in-place" update is quite innefficient on NUMA when `o` has not been also processed by `c`. An elegant solution is to use a similar forwarding mechanism to send these updates back to the collector that has written `o`, which now accesses its local space.

Fourth, the high-level optimisation to reduce forwarding makes our algorithm less dependent on the existence of efficient hardware support. Furthermore, the same optimisation also reduces the forwarding introduced in the previous paragraph.

Finally, NUMA would probably require a relatively large partition size L to reasonably amortise the copy-in and copy-out memory overhead. It follows that the heap graphs that can be effectively traced are those for which L still allows satisfactory load balancing.

5. Evaluation

We have implemented our tracing scheme in the form of a semi-space copying collector within the mature, research oriented platform offered by Jikes RVM (version 3.0.0), which includes MMTk. This section compares our algorithm against the original semi-space collector of MMTk on two platforms. The first has two quad-core AMD Opteron processors—model 2347 HE, and 16Gb of memory. The second is a quad-core commodity machine, having Intel Quad CPUs—model Q6600 running at 2.4GHz, and having 8Gb of memory. Both platforms run linux—Fedora Core 8.

The tests were aimed at demonstrating that our algorithm is more effective than the classical one on applications exhibiting large live data-sets, while still being competitive on the ones with small data-sets. On large data-sets, we report speed-up as high as $5.9\times$ and $5\times$ faster on 8 and 6 processors, and on average $4.44\times$ respectively. With small data-sets, we are generally within 20% of MMTk. For `Dacapo` tests, with the exception of `hsqldb`, even with modification of the installation scripts, we have not succeeded in increasing the live data-set. Increasing the heap size does not affect in these cases the size of the live data set and does not seem to affect speed-up. Consequently we use tests from `GCBench` and `JOlden`, since they allow one to easily vary the live data-size.

The MMTk documentation acknowledges scalability problems, but we did not expect MMTk's lackluster parallel speed-up on the `Jolden`, `GCBench` and `Hsqldb` tests. The cause is likely a poor load balancing mechanism. Unfortunately, MMTk is our only candidate for *fair* comparison. Although we still seem to get the upper hand, albeit less dramatically, when compared against the results reported in related work, the comparison suffers because of different applications, or heap sizes or architectures. On tree based applications, Flood *et al.* report a $4\times$ average speed-up against our $4.44\times$, when compared against the classical sequential collector.

Results for several `Dacapo` tests were omitted on the AMD platform because of their inappropriateness (filesystem/discIO-bound – 'lusearch' and 'luindex'), or unresolved installation dependencies ('chart' and 'eclipse'). With the exception of 'chart' we provide results for them on the `Intel` machine. Apart from that we use the `Intel` machine to validate the behaviour of large live data-sets; although we do not show all the results, they are consistent with those of AMD. We first introduce the testbeds and describe their characteristics in Section 5.1, then compare the running times of the two approaches in Section 5.2. Finally, Section 5.3 underlines several trade-offs and discusses the impact of our high-level optimisations.

[2] For example under the copy-in, process, copy-out-memory strategy. By NUMA we mean here platforms in which cores primarily access memory directly via DMA while communicating via message passing or small caches.

TESTS	$S_{\mathtt{heap}}$	PARAM	1/LR	$N_{\mathtt{gc}}$	L	$N_{\mathtt{obj}}$	IFR	MD	M/C
Antlr	120	def.	5.1	13	64	1.3	5.6	4.5	18
Bloat	150	def.	4.9	22	64	4.3	4.9	5.0	9
Pmd	200	def.	4.4	10	64	2.8	3.6	6.0	14
Xalan	150	def.	5.0	17	64	3.6	3.1	4.8	15
Fop	120	large	2.5	3	64	0.5	5.4	4.9	14
Jython	200	def.	5.6	18	64	3.6	3.0	6.4	15
luindex	120	def.	2.6	13	64	0.5	4.6	2.9	22
lusearch	150	def.	4.3	40	64	1.2	3.4	3.1	3
Eclipse_s	400	small	4.7	7	64	1.3	3.2	5.6	32
Eclipse_l	400	def.	4.7	29	512	11	2.6	12	15
Hsqldb_s	500	small	1.1	6	64	1.0	6.7	4.6	3
Hsqldb_l	500	large	1.5	9	128	22	5.6	5.9	5
GCbench_s	200		9	4	128	0.3	6.8	4.5	21
GCbench_l	999		4.3	44	128	81	4.2	15	5
Voronoi	500	-n $3*10^5$	2.1	3	128	6.7	3.7	21	9
TreeAdd	200+	-l 24	1	4	128	14	11	10	3
TSP	200+	-c $3*10^6$	1.2	3	128	4.5	12	6.1	23
MST	200+	-v 3500	1.1	4	128	19	3.6	41	4
Perimet	200+	-l 17	1	3	128	5.7	11	7.2	4
BH	850	-b $3*10^4$	14	11	128	1.9	4.1	4.7	33

Table 1. Testbed Properties (Benchmarks: `dacapo`, `GCBench`, and `JOlden`).
$S_{\mathtt{heap}}$ = max size of the heap (in Mb); Param = testbeds' parameters;
LR = ratio of live objects;
$N_{\mathtt{gc}}$ = the number of collections; L = partition size (in Kb);
$N_{\mathtt{obj}}$ = the number of to-space copied objects (in millions);
IFR = inverse forwarding ratio = $N_{\mathtt{obj}}$ / num of forwarded slots.
MD = our metadata size; M/C = mutator time divided by GC time
The double line separates the small and large data-set applications.

5.1 Benchmarks, Methodology and Trade-off Parameters

Table 1 introduces the programs on which we test our algorithm, together with their characteristics. We use 15 applications: rows 2–13 belong to `Dacapo`, rows 14 and 15 to `GCBench`, and rows 16–21 to `JOlden` benchmarks. The double line separates the large and small data-set tests. *Column* 2 shows the heap size, in `Mb`. We use the default heap sizes suggested by JikesRVM for `Dacapo`; for the rest we choose sizes that reduce the garbage collection overhead but still exhibit at least a few collections. For the tests exhibiting a live-object ratio close to one, we start with a heap of size 200`Mb` and allow it to increase dynamically.

Column 3 shows programs' parameters. `Dacapo` supports three workloads: `small`, `default` and `large`, but unfortunately the live data-set size does not vary (except for `Hsqldb` and `Eclipse`). We test $\mathtt{GCBench}_s$ on the original parameters (18, 16, 4, 16), and then we increase the data-set by modifying the tree-depth related parameters (23,21,4,23) – $\mathtt{GCBench}_l$. We also test `JOlden` applications on increased data-sets – see their corresponding parameters.

Column 4 shows the inverse of the live ratio: number of allocated objects to number of live objects per collection. This varies between 1 and 14, the most common case being between 4 and 5. `TreeAdd`, `MST`, `TSP` and `Perimeter` exhibits virtually no garbage. We still test them since (i) our algorithm targets a general form of tracing (e.g. data-serialisation, graph-traversal) that exhibits this pattern, and (ii) often VMs apply an incremental heap-size policy, and hence these cases need to be collected at least once.

Column 5 shows $\mathtt{N}_{gc}$:the number of collections per test run for our algorithm. In some cases MMTk performs fewer collections –

this is due to the fact that our metadata is bigger (partly because we use a naive worklist page-management strategy).

Column 6 shows `L`, the local size in `K` of our partitions. Since the hardware platforms we use exhibit fast cache coherency, we give small values to `L`. Our simple strategy selects 64`K` for small data-sets and 128`K` for large data-sets. Since `L` implements the trade-off between locality and load-balancing, NUMA architectures that use primarily DMA accesses would probably benefit from larger partitions (in the megabyte range) to amortize the cost of copying in and out memory segments. Section 5.3 shows that producer-consumer related optimisation can be effective even on our hardware.

Column 7 shows $\mathtt{N}_{obj}$–the total number of millions of objects copied to the to-space, over all collections of a test run. This is an estimate since collections do not occur at identical execution points.

Column 8 shows the average inverse forwarding ratio `IFR` over all collections of a test run on 6 processors: $\mathtt{N}_{obj}$ divided to the total number of inserts into the forwarding queues. Larger `IFR` values corresponds to less forwarding overhead, and hence to *good speed-up*. `IFR` is influenced primarily by `L`: large partitions usually lead to less forwarding but they may affect load-balancing. Garbage collection timing results in Tables 2 and 3 demonstrate that the forwarding ratio is a good indicator of the parallel speed-up. *On small data-set programs*, values between 3 and 4 correspond to slowdowns between 15% and 20%, when compared with MMTk's parallel version running on 4 or 6 processors. Values under 3 correspond to more significant slowdowns: 24% and 43%. *On large data-set programs*, even values between 3.5 and 4 generate acceptable speed-ups.

Column 9 shows the size of our metadata in `Mb`, and *column* 10 shows the ratio between the mutator time – i.e. total application time minus GC time, and the collection time. Both columns 9 and 10 correspond to the maximal tested parallelism; the mutator time differences between our approach and MMTk's are insignificant (+-5%). Finally, we point out that we compare against an out-of-the-box installation of JikesRVM with MMTk. Our semi-space collector uses MMTk's infrastructure – only the tracing scheme was modified. Both collectors are run under the `FastAdaptive` default configuration, with no replay compilation.

5.2 Empirical Results

Tables 2 and 3 show the total, per application run, garbage collection timings obtained with our and MMTk's tracing schemes, on the AMD and Intel platforms, respectively. Timings are measured via `-c MMTkCallback` and `-X:gc:verbose=1` options. `P` is the number of used processors. For each application/row, the timing of the sequential, free-of-locks version of MMTk's semi-space collector is considered to be 100. Table entries consist of `x:y` pairs, where the first `x` and second `y` number denote the normalised timings of our and MMTk tracing, respectively. For example the pair `20:108` means that we were $5\times$ $(100/20)$ faster than the optimal sequential execution, while MMTk was 8% slower. We use **bold** fonts for table entries for which our approach wins against MMTk at a margin higher then 10%. A double line separates the small and large data-set applications. In addition to the normalized collection times for each test, Tables 2 and 3 have three lines, labelled "Min || Eff.", "Avg || Eff." and "Max || Eff." that measure the parallel efficiency in each test. The entries are also of the form $x : y$, but in this case the numbers measure *time on* n *processors / time on 1 processor* $\times n$. Perfect parallelisation would give a value of 1, while no effective parallelisation would give a value of n. We see that our parallel tracing method is much more effective as the number of processors increases. We run each test 3 times and choose the best time result.

We make the observation that, in general, our tracing achieves *significantly* better speed-ups on large data-set, tree-based applications – up to $5.9\times$ and $5\times$ faster than the optimal sequential execution on eight and six processors. In most cases, on these applications, MMTk exhibits a rather mystifying *serial* behaviour, in that it narrowly fluctuates around the timing of its sequential execution.

Timing	P = 1	P = 2	P = 4	P = 6	P = 8
Antlr	130:112	93:83	63:58	52:51	n/a
Bloat	136:106	95:82	60:53	51:47	n/a
Pmd	141:108	92:74	62:48	50:43	n/a
xalan	142:110	93:77	58:54	46:45	n/a
Fop	157:128	93:76	64:57	56:61	n/a
Jython	145:108	102:70	64:58	53:44	n/a
Hsqldb$_s$	127:110	79:79	48:52	**39:50**	**36:46**
Hsqldb$_l$	125:111	**84:106**	**37:95**	**23:105**	**19:107**
GCben$_s$	115:109	78:71	**54:63**	**53:69**	**49:77**
GCben$_l$	121:121	**87:123**	**42:123**	**28:124**	**23:122**
Voronoi	118:110	**74:107**	**41:109**	**28:113**	**24:114**
TreeAdd	126:111	**65:108**	**30:109**	**20:108**	**17:106**
TSP	124:108	**65:102**	**35:88**	**23:86**	**19:76**
MST	172:107	92:50	46:49	**28:35**	**21:70**
Perimet	145:113	**84:111**	**42:109**	**31:108**	**26:113**
BH	117:109	77:68	**51:58**	**40:63**	**36:57**
Min || Eff.		1.03:0.93	0.95:1.78	0.95:1.96	0.98:3.35
Avg || Eff.		1.27:1.56	1.49:2.65	1.75:3.86	1.72:6.38
Max || Eff.		1.44:2.03	1.94:4.07	2.77:6.16	3.41:8.29

Table 2. Timings for the 8-core AMD Machine.
`P` is the number of processors used. The optimal (no locking), sequential execution time is 100 for all rows.
Table entries are pairs of the form `x:y`, where x and y correspond to the normalised collection times of ours and MMTk tracing algorithm, respectively. When $x < .9y$ **bold** is used.

Timing	P = 1	P = 2	P = 3	P = 4
Eclipse$_S$	138:112	89:73	76:63	66:53
Eclipse$_L$	148:116	100:69	81:57	69:48
Luindex	150:115	106:96	83:79	72:76
Lusearch	153:116	115:97	86:69	79:66
Antlr	139:119	100:87	75:66	66:63
Bloat	151:114	105:90	77:66	64:56
Pmd	141:114	85:68	69:53	64:53
Xalan	147:122	105:78	80:58	66:54
Fop	130:112	84:74	68:66	59:59
Jython	155:113	110:84	80:62	72:62
Hsqldb$_S$	131:116	83:95	**63:72**	58:60
Hsqldb$_L$	119:122	**85:118**	**45:117**	**38:113**
GCben$_L$	110:111	**78:115**	**45:117**	**40:115**
Voronoi	135:118	85:85	**58:111**	**52:115**
TreeAdd	112:120	**63:120**	**43:118**	**33:117**
Min || Eff.		1.13:1.19	1.13:1.39	1.18:1.66
Avg || Eff.		1.35:1.55	1.48:2.02	1.72:2.55
Max || Eff.		1.50:2.07	1.69:3.16	2.07:4.14

Table 3. Timings for the 4-core Intel Machine.

In contrast, the results show that our algorithm exhibits *robust scalability*. The inefficient sequential case and the big jump in speed-up when passing from 1 to 2 processors are partly consequences of the fact that 64K partitions are too small for the sequential version to be effective. We run the tests with all discussed optimisations on: although we do not expect them to be effective on small partition sizes, they incur small overhead. Furthermore, we believe that the absence of an inter-collector `yield` mechanism also incurs a non-negligible slowdown.

MMTk gets the upper hand on small data-sets, however, in most cases with a relatively small, under 22% margin with respect to our approach. As already remarked in the previous section, the forwarding ratio `IFR` (see Table 1), accurately relates with collector's speed-up on small data-sets: large values (≥ 4) result in comparable speed-ups – within 10%; smaller IFR values – i.e. between 3 and 4 – accentuate this difference to values between 13%-to-24% in MMTk's favour. $\texttt{Eclipse}_l$'s forwarding rate is *only* 2.6 and accordingly our algorithm is 44% slower than MMTk in this case.

To demonstrate the different algorithmic behaviour between applications with small and large data-sets, we include in Table 2 two versions of both Hsqldb and GCBench, that differ only in their data-set size. While our algorithm wins in all four cases, the small data-sets provides the tighter race. We attribute the good results to the fact that our algorithm exhibits neither locking nor cache conflicts on the hot execution paths.

5.3 Trade-off Parameters, High-Level Optimisations Impact

To demonstrate that the partition size L mediates between locality-of-reference and load balancing we observe that (i) a too small value, such as 4K incurs a 73% overhead on $\texttt{Hsqldb}_l$ when run on 8 processors on AMD and (ii) a too big value, such as 512K incurs a 50% slowdown on Bloat when run on 6 processors on AMD.

We bring two more arguments to underline the importance of the forwarding ratio. First, $\texttt{Eclipse}_l$ uses L=512K because we observed that its corresponding IFR was at least 0.5 bigger than the ones corresponding to other reasonable values of L. The next best speed-up is 17% slower than the one shown in row 9, Table 3, P=4. Second, we have observed that Voronoi's IFR also increases by $+1$ for L=512K, without affecting load-balancing. On the AMD machine this leads to a new, better value of 19 rather then 24, counting to a $5.26\times$ speed-up – see row Voronoi, P=8, in Table 2.

The high-level optimisations discussed in Sections 4.3 are not effective on small partitions (32 to 128K) – their impact is a negligible -2% to $+5\%$. However, on larger partitions (L=512K) they might prove useful even on our platform. For example the forwarding optimisation is accountable for a 10% and 15% increase in speed-up on 6 processors, on Pmd and Voronoi respectively.

6. Conclusions

Much previous work has explored parallel algorithms under the assumption of fairly uniform memory access cost. However, the hardware trend, particularly on commodity multicore processors, is that memory accesses to an area of memory is cheap only in the case that only one processor is accessing that area. Now that these architectures are mainstream, it is important to explore alternative algorithms which are serially monogamous in that an area of memory is used by one processor for a significant time.

In the context of parallel tracing algorithms, this paper has exhibited an alternative scheme which shows potential to be more effective on such architectures and has validated these predictions on current Intel and AMD multicore processors. More precisely, this paper has presented how to explicitly implement, exploit and optimise at a high level two abstractions: *locality* of reference of *non-shared data* and the inter-collector *communication*. The results demonstrate robust algorithm behaviour that scales well with both the data-set size and the number of processors. Our tracing algorithm does seem to exhibit the desired parallel efficiency. On our experiments with standard consumer computers, using six and eight processors, our algorithm was five and six times faster than the synchronization-sequential timing. Further experiments will be required to assess the upper limits of its scalability.

Acknowledgments

We thank the ISMM'09 reviewers for their useful comments and suggestions. The first and third authors gratefully acknowledge the Natural Science and Engineering Research Council of Canada for financial support.

References

[1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *SPAA*, 1998.

[2] C. Attanasio, D. Bacon, A. Cocchi, and S. Smith. A Comparative Evaluation of Parallel Garbage Collectors. In *LCPC, Springer Verlag*, pages 177–192, 2001.

[3] G. Attardi and T. Flagella. A Customisable Memory Management Framework. In *USENIX C++ Conference, Cambridge, MA*, 1994.

[4] H. Baker. Actor Systems for Real Time Computation. In *Tech. Rep. TR-197*, 1978.

[5] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A Parallel, Incremental Mostly Concurrent Garbage Collector for Servers. In *ACM Trans. Program. Lang. Syst. 27(6)*, pages 1097–1146, 2005.

[6] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE*, 2004.

[7] H. Boehm. Reducing Garbage Collector Cache Misses. In *ISMM*'00.

[8] C. J. Cheney. A Nonrecursive List Compacting Algorithm. In *Communications of the ACM 13 (11)*, pages 677–678, December, 1970.

[9] Perry Cheng and Guy E. Blelloch. A Parallel, Real-Time, Garbage-Collector. In *PLDI*, pages 125–136, 2001.

[10] Yannis Chicha and Stephen Watt. A Localised Tracing Scheme applied to Garbage Collection. In *APLAS, LNCS 4279*, 2006.

[11] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. G. Bobrow, and S. Shenker. Combining Generational and Conservative Garbage Collection: Frameworks and Implementations. In *POPL*, 1990.

[12] D. Doligez and X. Leroy. A Concurrent, Generational Garbage Collector for Multithreaded Implementation of ML. In *POPL*, 1993.

[13] T. Endo, K. Taura, and A. Yonezawa. A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared Memory Machines. In *SC*'97.

[14] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel Garbage Collection for Shared Memory Multiprocessors. In *JVM*, 2001.

[15] Robert H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. In *ACM Trans. Program. Lang. Syst. 7(4)*, pages 501–538, 1985.

[16] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage Collection Without Paging. In *PLDI*, 2005.

[17] Lorenz Huelsbergen and James R. Larus. A Concurrent Copying Collector for Languages that Distinguish Immutable Data. In *SIGPLAN Not., 28(7)*, pages 73–82, 1993.

[18] A. Imai and E. Tick. Evaluation of Parallel Copying Garbage Collection on a Shared Memory Multiprocessor. In *IEEE Trans. Parallel Distrib. Syst. 4(9)*, pages 1030–1040, 1993.

[19] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide. In *http://www.intel.com/products/processor/manuals/index.htm*, 2008.

[20] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. In *John Wiley and Sons*, July, 1996.

[21] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. In *ISMM*, 2008.

[22] M. M. Michael, M. T. Vechev, and V. A. Saraswat Idempotent Work Stealing. In *PPoPP*'09, pages 45–54.s

[23] David Plainfosse and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. In *Broadcast Technical Report*, 1994.

[24] Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. Pal Singh. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In *OOPSLA*, 2002.

[25] David Siegwart and Martin Hirzel. Improving Locality with Parallel Hierarchical Copying GC. In *ISMM*, pages 52–63, 2006.

[26] SUN. The SPARC architecture manual (version 9). In *Prentice-Hall, Editors: D. L. Weaver and T. Germond*, 1994.