# Topology-Aware Parallelism for NUMA Copying Collectors

Khaled Alnowaiser and Jeremy Singer

University of Glasgow, UK
k.alnowaiser.1@research.gla.ac.uk
jeremy.singer@glasgow.ac.uk

**Abstract.** NUMA-aware parallel algorithms in runtime systems attempt to improve locality by allocating memory from local NUMA nodes. Researchers have suggested that the garbage collector should profile memory access patterns or use object locality heuristics to determine the target NUMA node before moving an object. However, these solutions are costly when applied to every live object in the reference graph. Our earlier research suggests that connected objects represented by the *rooted sub-graphs* provide abundant locality and they are appropriate for NUMA architecture.

In this paper, we utilize the intrinsic locality of rooted sub-graphs to improve parallel copying collector performance. Our new topology-aware parallel copying collector preserves rooted sub-graph integrity by moving the connected objects as a unit to the target NUMA node. In addition, it distributes and assigns the copying tasks to appropriate (i.e. NUMA node local) GC threads. For load balancing, our solution enforces locality on the work-stealing mechanism by stealing from local NUMA nodes only. We evaluated our approach on SPECjbb2013, DaCapo 9.12 and Neo4j. Results show an improvement in GC performance by up to 2.5x speedup and 37% better application performance.

**Keywords:** NUMA, multi-core, work-stealing, runtime support, garbage collection

## 1 Introduction

Managed runtime systems—such as the Java Virtual Machine (JVM) and Common Language Runtime (CLR)—successfully abstract low-level platform-specific details such as hardware configuration and memory management. However development efforts for these runtime systems may struggle to cope with rapid evolution and diversity in hardware deployments. Contemporary multicore processors are often designed with a distributed memory architecture to improve memory bandwidth. This architectural layout means that individual processor cores may incur non-uniform memory access (NUMA) latency. Therefore, multi-threaded applications running on several cores may access remote memory. A garbage collected runtime may cause non-determinism in data placement, which will lead to unpredictable, suboptimal application performance, if the runtime system is not adapted to be aware of the underlying NUMA hardware.

A large body of research attempts to tackle data placement on NUMA architectures by means of improving locality and balancing allocation across memory nodes, e.g. [6]. A data placement policy that allocates data close to the core most frequently accessing it should minimize access time. However, locality-aware data placement policies could conflict with NUMA, perhaps through imbalance of access causing memory bus traffic saturation to some NUMA nodes. Other problems with NUMA imbalance include cache capacity issues, whereas using off-node caches may provide abundant memory space, e.g. [17].

In OpenJDK Hotspot (like many Java runtime systems) the generational Garbage Collector (GC) moves objects between spaces in the heap. For NUMA platforms, the existing memory placement and movement policies of the GC require re-engineering. Initially, the mutator threads use thread-local allocation buffers (TLABs) to allocate new objects in the young generation. Hotspot devolves memory mapping to the operating system. For example, Linux uses the *first-touch* policy as the default NUMA placement policy, which means that memory pages are mapped to the NUMA node associated with the core that first accesses a memory address in that page. As an advanced HotSpot configuration option, the user can choose a pre-defined JVM NUMA allocation policy (`-XX:+UseNUMA`) to map TLAB memory pages to local nodes.

Furthermore, GC threads also require local buffers, called promotion local allocation buffers (PLABs). A PLAB is used to move objects to the survivor spaces (in the young generation) and to the old generation. Mapping PLABs to NUMA regions remains the responsibility of the OS. Thus, the GC has the potential to *change* an object's NUMA node location after moving that object, which means subsequent mutator operations may incur remote access overhead. There is a need for topology awareness in the GC, which must take into account the NUMA architecture.

This paper extends our earlier work [1] which provided empirical observations of strong object locality in portions of the reference graph reachable from a single root reference. We refer to these graph components as *rooted sub-graphs*. In this paper, we modify the copying collector of the Hotspot JVM and implement a topology-aware parallel copying collector to preserve sub-graph locality and integrity. Our solution does not require any programmer intervention. We evaluated our algorithm with various benchmarks and the results show that leveraging rooted sub-graph locality improves substantially the GC performance (**up to 2.5x speedup**) and consequently improves application performance by **up to 37%**.

In this paper, we describe the following key contributions to the HotSpot GC:

a) We improve access locality by making the collector threads process *mostly* local objects.

b) We improve work-stealing locality such that idle threads fetch work from local threads' queues.

## 2 Motivation

The existing Parallel Scavenge copying GC in the Hotspot JVM uses conventional techniques for:

1. task generation: scanning memory areas that contain root references, e.g. mutator stacks, static areas, JNI handlers. At least one GC thread is used to scan each memory area. These task-generating threads enqueue root references locally, in a per-thread queue. *Our implementation distinguishes between root references and non-root references by storing them in different queues; the default scheme does not do this.*

2. distribution: each GC thread processes its own local queue of references—following references and processing (e.g. copying objects). *Our implementation directs references to appropriate queues, based on the underlying NUMA topology.*

3. load balancing: when a GC thread's local reference queue is empty, it randomly steals a single reference from the back of another thread's queue. This is a typical work-stealing approach. *Our implementation steals from nearer thread queues in terms of the NUMA topology, whereas the default scheme steals from an arbitrary queue in a NUMA-agnostic fashion.*

The key objective is to keep the GC threads busy collecting the heap regardless of the complex NUMA architecture. However, if a GC thread processes distant objects, it incurs remote memory access overhead. Further, the GC may relocate objects to a different NUMA node (e.g. during a copy-promotion); hence degrading mutator thread locality. Our topology aware GC scheme aims to alleviate both these problems.

Existing NUMA locality improvements for GC copying algorithms have a per-object granularity of work. Tikir and Hollingsworth [29] calculate the target NUMA node for an object copy by profiling thread access patterns to each object. Ogasawara [22] identifies the *dominant thread* of an object, which is likely to access the object most frequently. This analysis is based on references from thread stacks or the object's header.

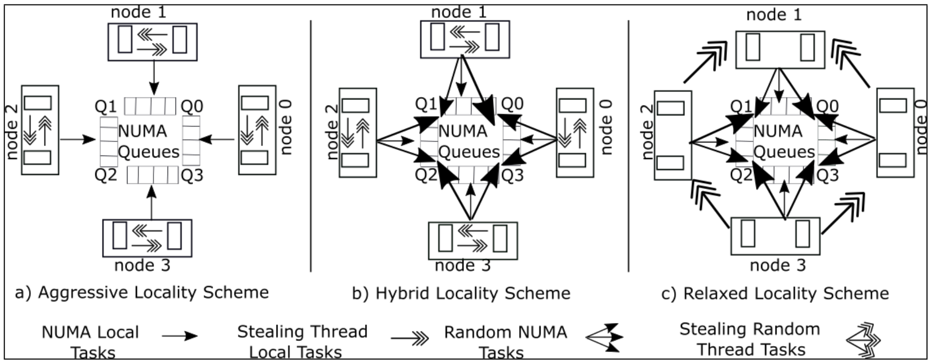
Conversely, Alnowaiser [1] identifies *rooted sub-graphs* which contain a root reference and its descendant references in the reference graph. Rooted sub-graphs are shown to exhibit abundant locality, i.e. the majority of objects in a sub-graph are located in the same NUMA node as the root of that sub-graph. Selection of the *rooted sub-graph* as the work granularity for GC is appropriate for NUMA systems for two reasons:

1. When a GC thread processes a task, i.e. a rooted sub-graph, it is likely to be processing objects in a single NUMA region—ideally local for that GC thread.
2. If parallel GC threads operate in different NUMA regions on thread-local data, there is a reduction in cross-node memory traffic, reducing bus contention.

## 3 Topology-Aware Copying Collector

Our proposed topology-aware parallel copying collector leverages the locality that exists in rooted sub-graphs. Since we set the work granularity to be a rooted sub-graph, the main principle in our approach is to preserve the subgraph integrity by processing its connected objects in a single NUMA node. As a result, a GC thread would move the entire rooted sub-graph to a single new location. For further locality gains, GC threads should process *local-node* root objects. We achieve this by organizing root objects according to NUMA nodes.

Moreover, when GC threads exhaust their local work queues, they should prefer to steal references from non-empty local queues of *sibling* cores, i.e. cores that are in the same NUMA node. This mechanism enables low access latency for work-stealing threads, and benefits from accessing shared resources (e.g. caches). Moreover, stolen objects will be moved to the same NUMA node as non-stolen objects in the same rooted sub-graph. Therefore the locality remains consistent.



a) Aggressive Locality Scheme

b) Hybrid Locality Scheme

c) Relaxed Locality Scheme

Fig. 1: Various topology-aware GC schemes. a) aggressive scheme only processes local-node tasks b) hybrid scheme distributes tasks across all nodes but steals from local node only. c) relaxed scheme processes random tasks from any node

### 3.1 Data Structures

Figure 1 illustrates the data structures used in our scheme. At VM initialization, we create as many double-ended queues as there are NUMA nodes, to store root references for processing. Since GC threads run concurrently, we need to ensure that enqueue and dequeue operations are thread safe. For this purpose, we use the OpenJDK `Arora` queue which supports single producer/multiple consumers. GC threads pop root references from one end safely using atomic operations; however, the other end is guarded such that a thread must lock the queue before doing any enqueue operation.

To reduce lock contention on Arora queues, we buffer root references in thread-local queues. When a GC thread completes its root scanning task, it enqueues discovered root references into the corresponding Arora queue. The memory footprint of this design is small since the root set size is small compared to the live object set.

Threads that complete root scanning tasks obtain root references from Arora queues. In order to shorten the time between root scanning and enqueue operations, when a thread-local queue reaches a threshold length, the GC thread enqueues all references in that thread-local queue to the corresponding Arora queue. We set the threshold to 100 references.

### 3.2 Algorithm

Copying collection starts with a set of predefined tasks that are created in a sequential block of code. The VM thread, which runs the sequential code, populates a shared queue with three different kinds of tasks to handle the parallel copying collection:

1. *root scanning* tasks to discover roots in various JVM data areas.
2. *stealing* tasks to balance the load among threads
3. a *finalizer* task to terminate the parallel phase.

These tasks are present in the default ParallelScavenge GC, however we have modified their behavior to implement topology-awareness as follows. Root scanning threads classify roots according to NUMA nodes and insert the references into the appropriate local queue. Once a local queue reaches a threshold, the thread locks the corresponding Arora queue and enqueues all discovered references. Stealing threads compete on dequeuing a reference from non-empty queues. When references in Arora queues are consumed, threads attempt to acquire work from pending queues of NUMA-local threads. The thread that acquires the final task performs the parallel phase termination.

Listing 1.1: Topology-Aware Copying Algorithm Pseudo Code

```
Task = acquire_gc_task()
switch (Task)
case scan_roots:
   for(all_root_areas){
      root = discover_roots()
      node = retrieve_root_node(root)
      enqueue_local_queue(root, node)
      if(queue(node)_size()>threshold)
        for(i =0; i<threshold; i++)
           enqueue_Arora_queue(root, node)
   }
case steal_work:
   node = get_thread_node()
   while(Arora_queue(node) != empty){
      ref = dequeue(node)
      follow(ref)
      }
   while(NUMA_local_queue(node) != empty){
      ref = dequeue()
      follow(ref)
      }
case final_task:
   wait_until_all_threads_terminate()
   hand_control_to_VM_thread()
   end
```

### 3.3 Optimization Schemes

We implement topology awareness for task distribution and work-stealing. However, retrieving an object's NUMA-specific location requires an expensive NUMA system calls. Therefore, we also explore various optimization schemes that preserve rooted sub-graph integrity but may not support locality for task distribution or work-stealing. Since we are optimizing two parallel techniques, we will have three optimization schemes, as illustrated in Figure 1:
**Aggressive:** GC threads look up an object's NUMA node at task generation phase, and only steal references from NUMA-local threads as described in Section 3.1 and Section 3.2.
**Hybrid:** GC threads process roots randomly however they steal from sibling (NUMA-local) queues only.
**Relaxed:** GC threads process roots randomly and steal work from any queue.

## 4 Experimental Setup

### 4.1 System Configuration

We evaluated our work on an AMD Opteron 6366 system. The NUMA topology consists of eight nodes on four sockets, with 64 cores in total. NUMA nodes are connected by Hyper-Transport links with transmission speed up to 6 GB/sec. Each node incorporates 64GB RAM, i.e. 512GB in total. The 64 cores are clocked at 1.8 GHz, and the machine runs Linux 3.11.4 64-bit. We set the OS memory policy to *interleaved*, which maps the memory pages to each memory node in a round-robin order. We use OpenJDK 8 for all our experiments. The 'original' JVM results use changeset 6698:77f55b2e43ae (jdk8u40-b06). All our modifications are based on this changeset.

### 4.2 Benchmarks

We use a variety of memory-intensive workloads to test our topology-aware copying collector:

**Neo4j/LiveJournal**: Neo4j is an embedded, disk-based, fully transactional Java persistence engine that stores data structures in graphs instead of tables [20]. The graph nodes and relationships are represented in the JVM heap. We use the LiveJournal social network data set, which consists of around 5 million nodes and 68 million edges [16]. We have a Java app that embeds Neo4j 2.2.1 as a library and queries the database to find all possible paths between two randomly selected nodes. The program uses 64 threads to drive the workload and uses a minimum of 150GB heap size. The all-paths operation is repeated twice and the total execution time is reported.

**DaCapo 9.12**: We run applications from the DaCapo 9.12 benchmark suite [3] that are compatible with JDK8, namely: avrora, pmd, xalan, sunflow, h2, lusearch, and jython. The heap size for each program is set close to minimum and the input size is large.

**SPECjbb2013**: SPECjbb2013 [27] is a server business application that models a world-wide supermarket company. In our experiments, SPECjbb2013 executes the full workload with a heap size of 3GB.

### 4.3 Evaluation Metrics

We use three different metrics to evaluate our GC implementation.

**NUMA Locality Trace:** Since our approach relies on rooted sub-graphs, we want to summarize quantitatively the NUMA locality of rooted sub-graphs. Our metric represents the locality richness in each sub-graph. To calculate the percentage of NUMA-local objects in a rooted sub-graph, we retrieve the NUMA node of the root and also the NUMA node of each descending object in the rooted sub-graph. For all rooted sub-graphs, the locality is recorded in an n-by-n square matrix, where n represents the number of NUMA nodes. Matrix element $a_{ij}$ records the proportion of objects residing in node $j$ that belong to a rooted sub-graph with root in node $i$.

We use the *Matrix Trace* property from Linear Algebra to calculate the NUMA locality of a program. The trace of an n-by-n square matrix $A$ is defined by the sum of the elements on the leading diagonal, i.e.

$$tr(A) = a_{11} + a_{22} + ... + a_{nn} = \sum_{i=1}^{n} a_{ii} \qquad 0 \leq tr(A) \leq n \times 100 \qquad (1)$$

In our system with eight nodes, $tr(A) = 800$ represents perfect NUMA locality, whereas $tr(A) = 0$ means that no objects are allocated in the same node as the root. However, due to the memory allocation policy and program behavior, some NUMA nodes might not be used at all. Thus we define the relative NUMA Locality Trace metric such that:

$$loc(A) = \frac{tr(A)}{n \times 100}, \qquad 0 \leq loc(A) \leq 1 \qquad (2)$$

where $n$ is the number of nodes that contain roots.

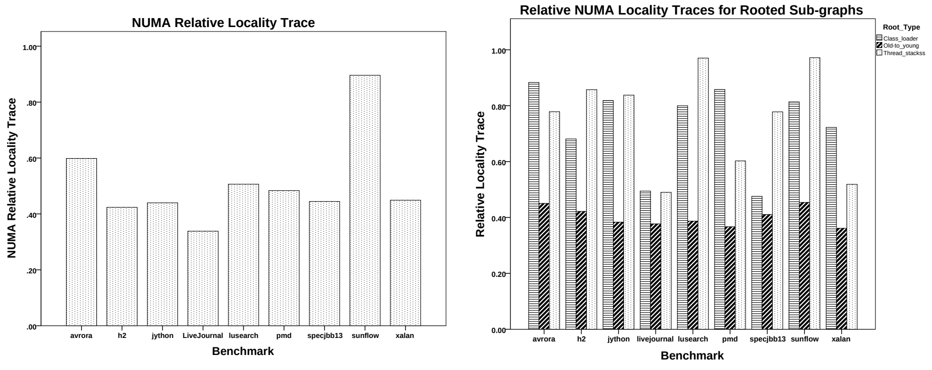
E.g. a program $p$ uses six nodes for object allocation and $tr(p) = 450$, thus, $loc(p) = 0.75$ and we interpret the result as 75% of objects are allocated in the same node as the root.

**Application Pause Time and Total Execution Time** We measure and report the pause time caused by the (stop-the-world) GC and the end-to-end execution time of the JVM. All timing measurements are taken five times. We report arithmetic means, and plot 95% confidence intervals on graphs.

**Scalability:** We run as many GC threads as the number of cores available to the system; however large heaps incur a scalability bottleneck. Roots available in the old generation are discovered by scanning the card table, which is a data structure used to record old-to-young pointers. As the heap size increases, the time consumed by scanning the card table grows; hence, we analyze the responsiveness of our optimization schemes to the increased heap size.

# 5 Evaluation

## 5.1 NUMA Locality Trace



(a)

(b)

Fig. 2: (a) Relative NUMA Locality Traces for evaluated benchmarks. On average, 50% of objects are local within rooted sub-graphs. (b) Relative NUMA Locality Traces for various root types: old-to-young, thread stacks, and class loader roots. An old-to-young rooted sub-graph exhibits relatively low locality.

Figure 2a shows the relative NUMA locality trace, see Section 4.3. For Neo4j/LiveJournal, we are unable to process all the data collected due to the huge size; however, we use the data from the fifth GC cycle only as a sample of the application's GC phase. DaCapo/Sunflow obtains the best relative NUMA locality trace results. Approximately, 90% of objects are co-located in the same node as the root. On the contrary, objects in DaCapo/h2 are dispersed across NUMA nodes and rooted sub-graphs provide low locality traces: 42%. For all benchmarks, the relative NUMA locality trace is 53% on average. These results differ from our earlier empirical study [1], which demonstrated higher locality. The main difference is that we now examine rooted sub-graphs from the young generation in our samples. This may suggest that we cannot rely on the locality features of rooted sub-graphs to optimize the copying GC. However the following experiment gives more insight on rooted sub-graph locality for different kinds of roots.

Recall that at the start of parallel GC, various root scanning tasks are inserted in shared queues. These tasks direct the GC threads to different JVM data areas which contain potential root references. These areas include mutator thread-local stacks, card table (for old-to-young references), class loader data, JNI handlers, etc. We calculate the NUMA locality traces for prevalent root kinds and plot the results in Figure 2b. For all evaluated benchmarks, the old-to-young rooted sub-graphs consistently obtain low locality results, whereas other roots show high locality.

These results suggest that aggressive locality optimization can be applied on selected root types. In the next section, we show that GC performance increases when applying locality optimization on *all root types except old-to-young references*. For old-to-young root, we randomly assign root references to any queue.

### 5.2 Pause Time and VM Time Analysis

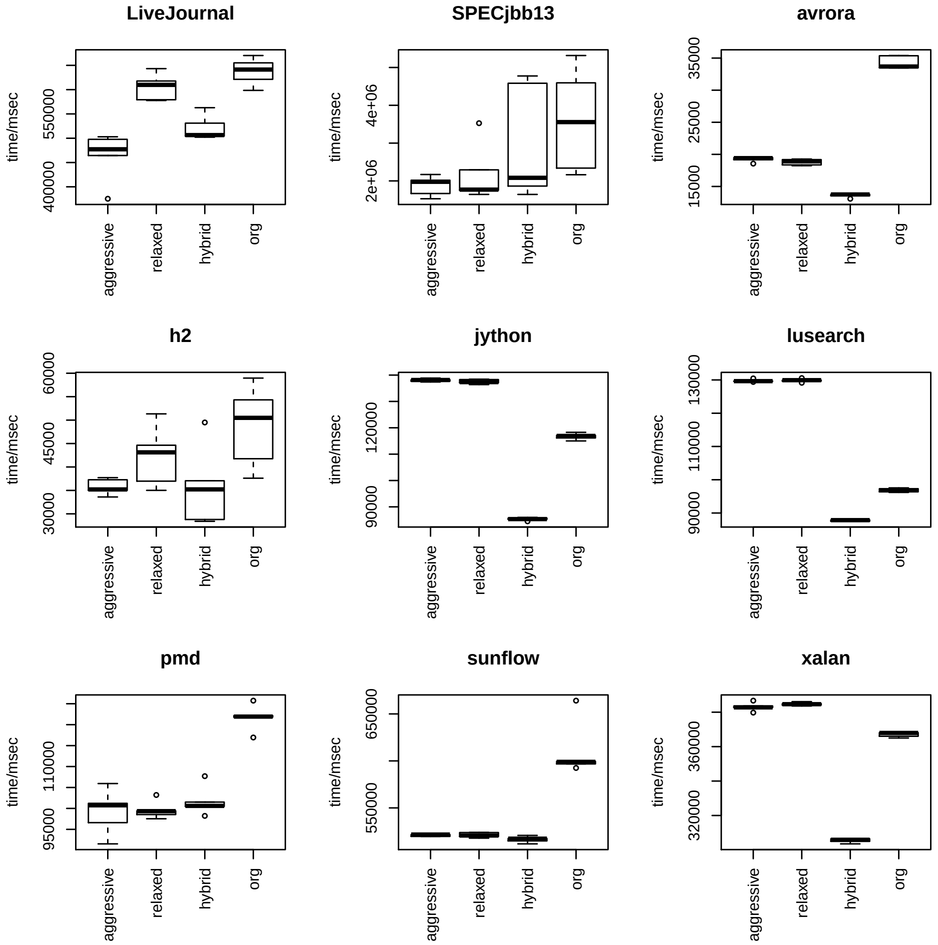


Fig. 3: GC time (i.e. pause time) for our three schemes. For small heaps (e.g. DaCapo programs), hybrid scheme gives the best results, whereas aggressive scheme is more effective for apps with larger heaps. (The default JVM is labeled Org.)

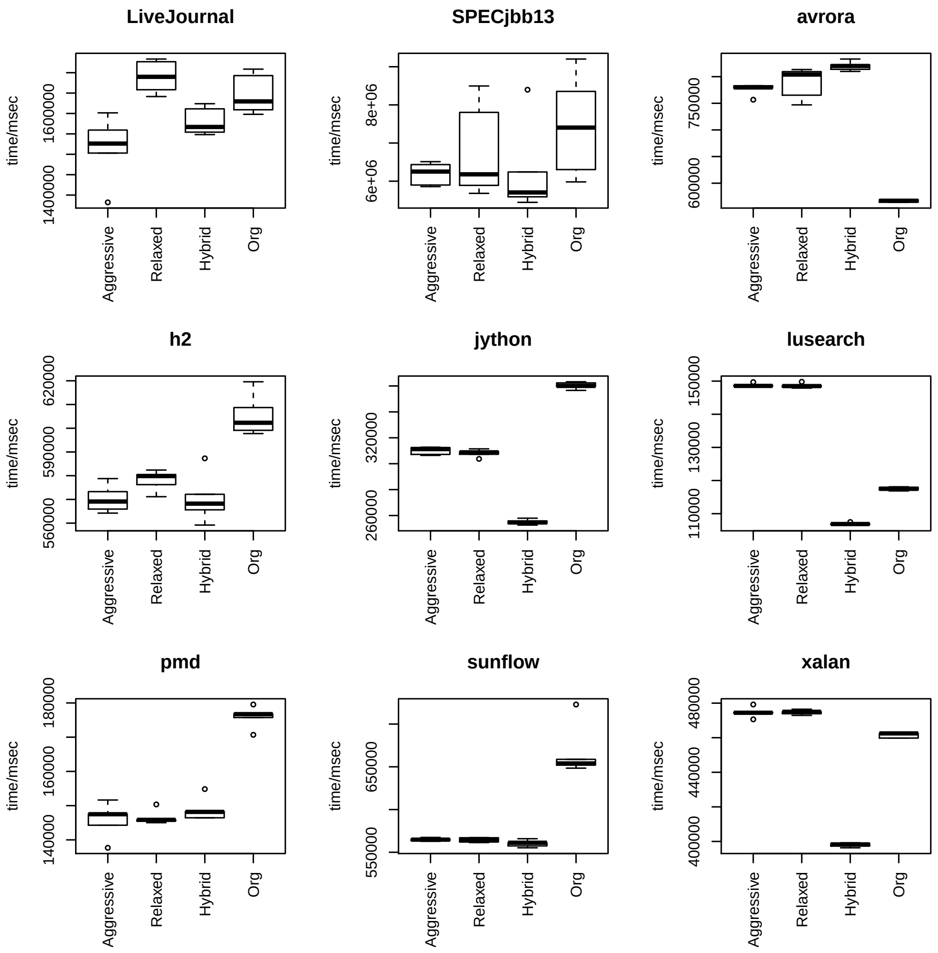


Fig. 4: VM time (i.e. end-to-end execution time) for our three schemes. At least one scheme provides better VM execution time than default (labeled Org) in most cases.

Figures 3 and 4 plot the GC pause time and VM execution time results for the Java benchmarks. Proposed topology-aware parallel techniques for task distribution and work-stealing outperform the default Hotspot ParallelScavenge GC (labeled *org*).

For programs with small heap sizes, represented here by DaCapo benchmarks, we observe that programs take the best advantage from Hybrid scheme. Hybrid optimization scheme speeds up the GC performance by up to 2.52x and never degrades it significantly. However, not all DaCapo programs follow the same performance trend. For instance, DaCapo/avrora gains 2.5x GC speedup but the VM performance degrades by 31%. Avrora simulates a number of applications running on a grid of micro-controllers. Previous studies [15] [24], report that DaCapo/avrora incorporates extensive inter-thread communications and the application threads benefit from increased cache capacity. Thus, efforts to improve locality counteract this cache optimization.

We note that locality is vital to programs that have large heaps. Our approach improves Neo4j/LiveJournal GC performance by 37%, 22%, and 5% for aggressive, hybrid, and relaxed optimization schemes respectively. With the aggressive scheme, SPECjbb2013 records improvement in GC and VM performance by 91% and 20% respectively.

### 5.3 Scalability

When the heap size gets large, the copying collector might spend much time scanning roots for old-to-young references in the card table. Our experience is that many live objects are discovered through the card table; thus, the card table scanning accounts for the majority of GC pause time. On heap sizes above 100GB, we found that card table handling often takes hundreds of seconds.

In this section, we study the scalability of our optimization schemes as the heap size increases. The experiments were run on Neo4j/LiveJournal with heap sizes of 100, 150, and 200 GB. Figure 5 shows the GC time and VM time scalability results. Ideally, as the heap size increases, the number of GC cycles decreases. However, the original GC implementation shows a slight increase in the GC time. We argue that this increase is due to the time consumed by processing the card table—in particular due to three factors. First, old-to-young rooted sub-graphs tend to be deep and require time for processing. Second, we have shown in Section 5.1 that such type of roots possess poor locality between objects; hence, incur significant remote access overhead. Third, deep sub-graphs are susceptible to work-stealing, thus, object connectivity will be broken and objects are scattered across NUMA nodes.

Our three optimization schemes improve the second and third aspects and provide better scalability results. Preserving the rooted sub-graph integrity and enforcing topology awareness on work-stealing scales the GC time substantially. In fact, relaxed scheme which aims only at processing connected objects as a unit outperforms the original GC at 200GB heap size. These results show that large heap sizes necessitate knowledge of the NUMA architecture to improve memory access behavior.

For VM scalability, the original JVM implementation provides a steady VM time over the three heap sizes and is not affected by the GC pause time. Hybrid and relaxed optimization schemes observe reduction in VM time but moderate scaling with increased heap size. Aggressive scheme follows the GC result's trend and obtains better scalability results.

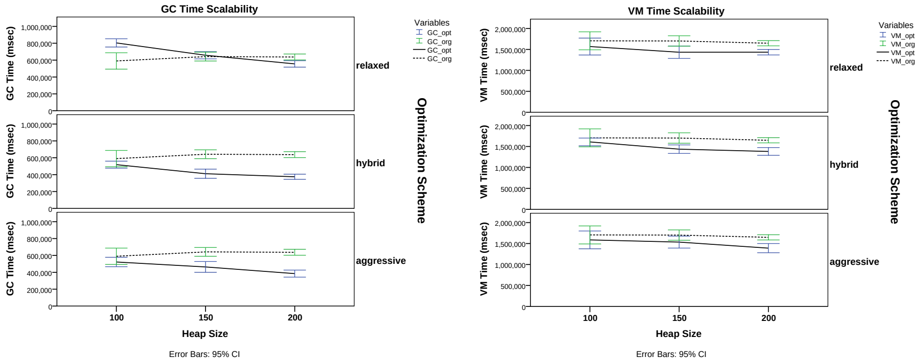


Fig. 5: GC (left) and VM (right) time scaling with heap size for Neo4j/LiveJournal. GC time decreases with heap size for our optimized versions, whereas the original implementation does not show any scaling.

## 6 Related Work

Prior work proposes allocating related objects close to each other to provide locality. Objects are co-allocated based on various criteria including temporal access patterns [5] and types [25]. Graph traversal order can improve object locality. Wilson et al. [30] suggest a hierarchical decomposition traversal order. This involves two different queues: small queues for descendant objects of some particular object in order to group them in a memory page, and a large queue to link these small queues. In our algorithm, we use two queues: NUMA queues for roots and local queues for rooted sub-graphs. Huang et al. [13] attempt to group frequently accessed fields in hot methods by sampling the program execution. At GC time, referents of hot fields are copied with their parents.

Thread-local heaps enforce local access to thread-specific objects [2,14,7,18,28]. New objects are initially allocated in thread-local heaps until they are referenced by non-local objects. Such objects are promoted to a shared heap. Zhou and Demsky [32] implement master/slave collector threads and thread-local heaps. Each slave thread collects its own heap only. Any reference to a non-local object is sent to the master thread, which communicates with the appropriate thread to mark it live. In our algorithm, every GC thread is associated with a particular NUMA node and processes objects in that node only.

NUMA-aware collectors take into account object location before and after collection time. Tikir and Hollingsworth [29] sample memory accesses and move objects to the memory node of the thread accessing the object most frequently. Ogasawara [22] uses the dominant-thread information of each live object, e.g. thread holding the object lock, to identify the target memory node.

Connected objects in the object graph share various attributes. Hirzel et al. [12] examine different connectivity patterns with relation to object lifetime. They conclude that connected objects that are reachable only from the stack are shortlived; whereas, objects that are reachable from globals live for long time, perhaps immortally. In addition, objects that are connected by pointers die at the same time. Alnowaiser [1] studies the locality of connected objects and reports that the majority of connected objects, which form a sub-graph of a root, reside in the same memory node as that root.

Parallel GC algorithms aim to keep all cores busy processing the live object graph. The fundamental technique of task distribution is to create a per-thread work list that contains a set of tasks accessible by the owner thread. For load balancing, threads that complete processing their own queues steal tasks from non-empty queues [9,8,26,31]. However, such *processor-centric* algorithms do not consider object locality and may incur additional overhead for processing distant objects.

*Memory-centric* parallel GC algorithms take the object location into consideration. The heap is segregated into segments and each GC thread processes one or more segments. References to local objects in each segment are processed, whereas references to remote objects are pushed into a queue of the corresponding segment [4]. Alternatively, Shuf et al. [25] push references to remote objects into a shared queue enabling other threads to process them. For load balancing, GC threads need to lock unprocessed queues to trace live objects [21]. However, a memory segment boundary might not match the physical memory page size nor it is assigned to local threads; therefore, further locality improvements are required.

Work-stealing algorithms negatively affect object locality by separating child objects from their parents. Gidra [10] remarks that disabling work-stealing improves program performance for some applications. Ananya et. al. [19] suggest a locality-aware work-stealing algorithm, which calculates the distance between NUMA nodes in a system with multi-hop memory hierarchy. An idle thread on a node attempts to steal work from the ‘nearest’ non-empty queues. Olivier et. al. [23] propose a hierarchical work-stealing algorithm to improve locality. They enable one third of running threads to steal work on behalf of other threads in the same chip and push stolen work into a shared queue for local threads. Our approach allows threads to steal from local threads only, to preserve NUMA locality.

## 7 Conclusion

We have shown that a NUMA topology-aware copying GC based on per-node reference queues is able to preserve much of the rooted sub-graph locality that is inherent from mutator allocation patterns. Our improved copying GC has significant benefits—with improvements in GC performance up to 2.5x speedup and up to 37% faster application runtime for non-trivial Java benchmarks.

We argue that further improvements are possible based on not only preserving locality of reference sub-graphs in single NUMA nodes, but also using local GC threads to operate on these sub-graphs. At present, we rely on expensive system calls to identify local work for GC threads—but cheaper techniques are presented in recent literature [11].

In summary, GC implementations should attempt to preserve intra-node reference graph locality as much as possible, to enable subsequent low-latency access times for both mutator and collector threads.

## Acknowledgments

We would like to thank the University of Prince Sattam bin Abdulaziz for funding this research. We also thank the UK EPSRC (under grant EP/L000725/1) for its partial support.

## References

1. Alnowaiser, K.: A Study of Connected Object Locality in NUMA Heaps. In: Proc. MSPC. pp. 1:1–1:9 (2014)
2. Anderson, T.A.: Optimizations in a Private Nursery-based Garbage Collector. In: Proc. ISMM. pp. 21–30 (2010)
3. Blackburn, S.M., et al.: The Dacapo Benchmarks: Java Benchmarking Development and Analysis. In: Proc. OOPSLA. pp. 169–190 (2006)
4. Chicha, Y., Watt, S.M.: A Localized Tracing Scheme Applied to Garbage Collection. In: Proc. APLAS. pp. 323–339 (2006)
5. Chilimbi, T., Larus, J.: Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In: Proc. ISMM. pp. 37–48 (1998)
6. Dashti, M., Fedorova, A., Funston, J.: Traffic management: a holistic approach to memory placement on NUMA systems. In: Proc. ASPLOS. pp. 381–393 (2013)
7. Domani, T., Goldshtein, G., Kolodner, E.K., Lewis, E., Petrank, E., Sheinwald, D.: Thread-local Heaps for Java. In: Proc. ISMM. pp. 76–87 (2002)
8. Endo, T., Taura, K., Yonezawa, A.: A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines. In: Proc. SC. pp. 1–14 (1997)
9. Flood, C., Detlefs, D., Shavit, N., Zhang, X.: Parallel Garbage Collection for Shared Memory Multiprocessors. In: Proc. JVM (2001)
10. Gidra, L., Thomas, G., Sopena, J., Shapiro, M.: Assessing the Scalability of Garbage Collectors on Many Cores. In: Proc. PLOS. pp. 1–7 (2011)
11. Gidra, L., Thomas, G., Sopena, J., Shapiro, M., Nguyen, N.: NumaGiC: A garbage collector for big data on big NUMA machines. In: Proc. ASPLOS. pp. 661–673 (2015)
12. Hirzel, M., Henkel, J., Diwan, A., Hind, M.: Understanding the Connectivity of Heap Objects. In: Proc. ISMM. pp. 36–49 (2002)
13. Huang, X., Blackburn, S.M., McKinley, K.S., Moss, J.E.B., Wang, Z., Cheng, P.: The Garbage Collection Advantage: Improving Program Locality. In: Proc. OOPSLA. pp. 69–80 (2004)

14. Jones, R., King, A.: A Fast Analysis for Thread-local Garbage Collection with Dynamic Class Loading. In: Proc. SCAM. pp. 129–138 (2005)
15. Kalibera, T., Mole, M., Jones, R., Vitek, J.: A Black-box Approach to Understanding Concurrency in DaCapo. In: Proc. OOPSLA. pp. 335–354 (2012)
16. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data` (Jun 2014)
17. Majo, Z., Gross, T.R.: Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In: Proc. ISMM. pp. 11–20 (2011)
18. Marlow, S., Peyton Jones, S.: Multicore Garbage Collection with Local Heaps. In: Proc. ISMM. pp. 21–32 (2011)
19. Muddukrishna, A., Jonsson, P.A., Vlassov, V., Brorsson, M.: Locality-aware Task Scheduling and Data Distribution on NUMA Systems. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) OpenMP in the Era of Low Power Devices and Accelerators, LNCS, vol. 8122, pp. 156–170 (2013)
20. Neo4J: (2015), `http://www.neo4j.com/`
21. Oancea, C.E., Mycroft, A., Watt, S.M.: A New Approach to Parallelising Tracing Algorithms. In: Proc. ISMM. pp. 10–19 (2009)
22. Ogasawara, T.: NUMA-aware Memory Manager with Dominant-thread-based Copying GC. In: Proc. OOPSLA. pp. 377–390 (2009)
23. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling Task Parallelism on Multi-Socket Multicore Systems. In: Proc. ROSS. pp. 49–56 (2011)
24. Sartor, J.B., Eeckhout, L.: Exploring Multi-threaded Java Application Performance on Multicore Hardware. In: Proc. OOPSLA. pp. 281–296. New York, USA (2012)
25. Shuf, Y., Gupta, M., Franke, H., Appel, A., Singh, J.P.: Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In: Proc. OOPSLA. pp. 13–25 (2002)
26. Siebert, F.: Limits of Parallel Marking Garbage Collection. In: Proc. ISMM. pp. 21–29 (2008)
27. SPECjbb2013: Standard Performance Evaluation Corporation Java Business Benchmark (2013), `http://www.spec.org/jbb2013`
28. Steensgaard, B.: Thread-specific Heaps for Multi-threaded Programs. In: Proc. ISMM. pp. 18–24 (2000)
29. Tikir, M.M., Hollingsworth, J.K.: Numa-aware Java Heaps for Server Applications. In: Proc. IPDPS. p. 108.b (2005)
30. Wilson, P.R., Lam, M.S., Moher, T.G.: Effective Static-graph Reorganization to Improve Locality in Garbage-collected Systems. In: Proc. PLDI. pp. 177–191 (1991)
31. Wu, M., Li, X.F.: Task-pushing: A Scalable Parallel GC Marking Algorithm without Synchronization Operations. In: Proc. IPDPS. pp. 1–10 (2007)
32. Zhou, J., Demsky, B.: Memory Management for Many-core Processors with Software Configurable Locality Policies. In: Proc. ISMM. pp. 3–14 (2012)