NumaGiC: a Garbage Collector for Big Data on Big NUMA Machines

Lokesh Gidra
LIP6-UPMC/INRIA
lokesh.gidra@lip6.fr

Gaël Thomas
SAMOVAR-Telecom SudParis
gael.thomas@telecom-sudparis.eu

Julien Sopena
LIP6-UPMC/INRIA
julien.sopena@lip6.fr

Marc Shapiro
LIP6-INRIA/UPMC
marc.shapiro@acm.org

Nhan Nguyen
Chalmers University of Technology
nhann@chalmers.se

Abstract

On contemporary cache-coherent Non-Uniform Memory Access (ccNUMA) architectures, applications with a large memory footprint suffer from the cost of the garbage collector (GC), because, as the GC scans the reference graph, it makes many remote memory accesses, saturating the interconnect between memory nodes. We address this problem with NumaGiC, a GC with a mostly-distributed design. In order to maximise memory access locality during collection, a GC thread avoids accessing a different memory node, instead notifying a remote GC thread with a message; nonetheless, NumaGiC avoids the drawbacks of a pure distributed design, which tends to decrease parallelism. We compare NumaGiC with Parallel Scavenge and NAPS on two different ccNUMA architectures running on the Hotspot Java Virtual Machine of OpenJDK 7. On Spark and Neo4j, two industry-strength analytics applications, with heap sizes ranging from 160 GB to 350 GB, and on SPECjbb2013 and SPECjbb2005, NumaGiC improves overall performance by up to 45% over NAPS (up to 94% over Parallel Scavenge), and increases the performance of the collector itself by up to 3.6× over NAPS (up to 5.4× over Parallel Scavenge).

Categories and Subject Descriptors D.4.2 [*Software*]: Garbage collection

General Terms Experimentation, Performance

Keywords Garbage collection; NUMA; multicore

ASPLOS '15, March 14 - 18 2015, Istanbul, Turkey.

http://dx.doi.org/10.1145/2694344.2694361

1. Introduction

Data-analytics programs require large amounts of computing power and memory. When run on modern multicore computers with a cache-coherent Non-Uniform Memory Access (ccNUMA) architecture, they suffer from a high overhead during garbage collection (GC) caused by a bad memory access locality. A ccNUMA architecture consists of a network of nodes, each comprising several cores and a local memory bank. A ccNUMA architecture hides the distributed nature of the memory from the application. The application thus unknowingly creates *inter-node references* when it stores a reference to an object located on a given node into the memory of another node. In turn, when a GC thread traverses the object graph to identify live objects, it silently traverses inter-node references and thus processes objects on any memory node. Consequently, GC threads access remote memory often, which increases memory access latency, and causes a large network traffic that can saturate the network between the nodes.

Intuitively, to access only local memory, a GC thread that encounters a reference to a remote object might delegate the processing of that object to a GC thread running on the remote node. This design, unfortunately, degrades parallelism, because a GC thread remains idle when it does not have local objects to collect, waiting for another GC thread to reach some object located on its own node.

Based on this observation, we designed the NumaGiC collector to aim for memory access locality during the GC, without degrading GC parallelism. It features a *mostly-distributed* design, in which a GC thread can be in two modes. *Local mode* focuses on memory access locality and avoids remote object access. In this mode, when a GC thread finds a reference to an object located on a remote node, it notifies a GC thread on the remote node, which processes the object and its reachable subgraph (i.e., objects that it points to, and so on recursively) locally. A GC thread enters *work-stealing*

mode when it runs out of local objects to process. Work-stealing mode focuses on parallelism and allows a GC thread to "steal" references to process from any other node, and to access such references itself, even if they are remote. A GC thread in work-stealing mode periodically re-enters local mode, in order to check whether some local reference has become available again, either sent by a remote thread, or discovered by scanning a stolen reference.

NumaGiC also includes a set of NUMA-friendly placement policies that prepare the heap for the mostly-distributed design. First, the policies aim at minimising the number of inter-node references. Because sending a reference is slightly more costly than remotely accessing a small object, sending a reference is only beneficial if, on average, the referenced object itself references many objects on its same node. In this case, the cost of sending the reference is amortised over the memory access locality of the receiving GC thread. Second, the policies aim to balance the amount of live objects across memory nodes, in order to equalise the work between GC threads. Indeed, if some nodes are underloaded, their GC threads quickly enter work-stealing mode and start accessing the overloaded nodes, which can potentially saturate their memory controller.

We implemented NumaGiC in the Hotspot Java Virtual Machine of OpenJDK 7. NumaGiC targets long-running computations that use large data sets, for which a throughput-oriented *stop-the-world* GC algorithm is suitable.[1] NumaGiC is based on Parallel Scavenge, the default throughput-oriented GC of Hotspot. Our experiments compare NumaGiC with (a slightly improved version of) Parallel Scavenge, and with an improved version of Parallel Scavenge designed for ccNUMA architectures called NAPS [10]. NAPS balances memory access among nodes and improves memory access locality for only newly-allocated objects, whereas NumaGiC also improves memory access locality for objects that survive collection and enforces memory access locality by using messages when GC threads have local memory to collect.

We evaluate NumaGiC against two widely-used big-data engines, Spark [23] and Neo4j [17], with Java heaps sized from 110 GB to 350 GB. We also evaluate two industry-standard benchmarks, SPECjbb2013 [25], and SPECjbb2005 [24], along with the DaCapo 9.12 and SPECjvm2008 benchmarks. We experiment both on a 48-core AMD Magny-Cours machine with 8 nodes and a total memory size of 256 GB, and on a 40-core Intel Xeon E7 hyper-threaded machine with two execution units per core, with 4 nodes and total memory size of 512 GB. Our evaluation shows that:

- On applications with large heap, NumaGiC always increases the overall performance of applications on the two machines. With heap size that provide the best possible application performance for the three evaluated GC, NumaGiC improves the overall performance of applications by up to 45% over NAPS, and by up to 94% over Parallel Scavenge. This result shows that a mostly distributed design increases substantially the performance of applications with large heap on ccNUMA machines, and that a mostly distributed design seems to improve performance independently of the architecture.
- On applications with large heap with the most efficient heap size for all the GC, NumaGiC increases collector throughput, *i.e.*, the number of live bytes processed per time unit, on the two machines by up to 3.6× and by up to 5.4×, compared to NAPS and Parallel Scavenge respectively.
- On applications with large heap, NumaGiC scales substantially better than NAPS, and Parallel Scavenge with increasing number of NUMA nodes. NumaGiC scales almost linearly in case of SPECjbb2005, and Spark. The only limiting factor in its scalability is the amount of inter-node shared objects, which depends on the application's execution behavior. The design of NumaGiC itself is scalable.
- On 33 applications from DaCapo 9.12 and SPECjbb2008 with smaller working set, NumaGiC improves substantially the performance of 7 applications by more than 5%, and only degrades the performance of a single application by more than 5% (by 8%). This result shows that a mostly distributed design is almost always beneficial and statistically beneficial for 20% of the applications with modest workload.

The paper is organized as follow. Section 2 presents Parallel Scavenge, the GC at the basis of NumaGiC. Section 3 presents the NUMA-friendly placement policies, and Section 4 the design of NumaGiC. Section 5 reports on our evaluation results and discusses their implications. Section 6 reviews related work, and Section 7 concludes the paper.

2. Background

A garbage collector (GC) observes the memory of some application process, called the mutator. Starting from some well-known roots, it records which objects are referenced, *scans* them for more references, and so on recursively. Reachable objects are *live*; unreachable ones are garbage and are deallocated.

Parallel Scavenge is the default GC of the Hotspot Java Virtual Machine, and forms the starting point for this work. Parallel Scavenge is a stop-the-world, parallel and generational collector [13]. This means that the application is suspended during a collection, that it spawns several parallel GC threads, and that it segregates objects into generations. Generations are justified by the observation that, in the common case, a young object becomes garbage more quickly than an older object [2, 14, 29].

[1] A stop-the-world GC algorithm suspends the application during collection, in order to avoid concurrent access to the heap by the application. A stop-the-world GC is opposed to a concurrent GC [13], which favours response time at the expense of throughput, because it requires fine-grain synchronisation between the application and the GC.

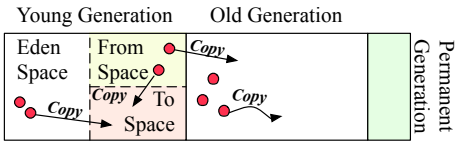


Figure 1. Memory Layout of Parallel Scavenge

Parallel Scavenge has three generations (see Figure 1): the *young generation* contains recently-allocated objects, the *old generation* contains older objects, and the *permanent generation* contains class definitions. Hereafter, we conflate the permanent with the old generation to simplify the presentation. Parallel Scavenge has two kinds of collections. A *young collection* collects only the young generation; a *full collection* collects both generations. Parallel Scavenge leverages the generational hypothesis by performing young collections more frequently than full collections.

In the remainder of this section, we describe the young and the full collectors, then discuss the parallelism of Parallel Scavenge, and finally discuss the memory placement of Parallel Scavenge.

2.1 Parallel Scavenge young collection

The Parallel Scavenge young collection is a copying algorithm. When it reaches a live object, it copies it elsewhere in order to compact the memory. This design fights fragmentation, and ensures that dead memory can be recycled in large chunks.

Parallel Scavenge promotes a young object to the old generation only after it has survived a few young collections. For this purpose, it divides the young generation into the so-called *eden space* and two *survivor spaces*, called *from-space* and *to-space*.

Parallel Scavenge allocates new objects in eden space. The from-space contains the young objects that have survived a few collections, and, before a collection, the to-space is empty.

Parallel Scavenge copies live objects of both the eden space and the from-space, either to the to-space or to the old generation, according to their age (see Figure 1). At the end of collection, the eden space and the from-space are empty and can be recycled, and to-space contains the surviving young objects. In the next cycle, the eden space again serves to allocate new objects, and the from- and to-spaces are swapped.

The roots for young collection are the global variables, the stacks of the mutator threads, and the old-to-young root objects, *i.e.*, the old-generation objects that reference objects in the young generation. Parallel Scavenge identifies the old-to-young root objects by instrumenting writes performed by the mutator. It divides the old generation into chunks called cards [21, 30]. Each card has a corresponding bit in a *card table*, indicating whether the card might contain a reference to the young generation. The GC traverses the card table, in order to identify the cards that potentially contain old-to-young root objects. Each time an old object is assigned with a reference towards the young generation, the Java Virtual Machine marks the corresponding card table entry "dirty," indicating the possible existence of a young-generation root.

2.2 Full collection

Full Collection re-uses many of the data structures and algorithms used by young collection. It uses a three-phase mark-compact algorithm, compacting the old generation by copying live objects to the beginning of the old generation memory area to avoid fragmentation. In the first phase, called the marking phase, the GC threads traverse the object graph in parallel and mark the live objects. In the second, summary phase, Parallel Scavenge calculates (sequentially) the destination address of the live objects; this mapping is organised in regions of 4 KB each. In the third, compacting phase, the GC threads process the regions in parallel and copy the live objects to the beginning of the old generation.

2.3 Parallelism in Parallel Scavenge

Parallel Scavenge uses a master-slave design. The master thread, called the VM Thread, queues some tasks, described next, in a single task queue. Then it starts the parallel phase, in which GC threads dequeue tasks and perform the corresponding work.

A **root task** records some root of the object graph. Root tasks are used both in the copying phase of the young collector and in the marking phase of the full collector. A GC thread that dequeues a root task traverses the object graph depth first. For this purpose, the thread maintains a so-called *pending queue* of references, initialized with the references indicated in the root task. When the thread dequeues a reference, if the referenced object was already processed, the GC thread skips it. Otherwise, it either copies (in the young collector) or marks (in the marking phase of the full collector) the object, and finally scans the object and appends the references it contains to its pending queue.[2] When the pending queue is empty, the root task is finished.

A **compacting task** is used by a GC thread to compact the heap during the compacting phase of the full collector, by copying the regions, according to the mapping computed during the summary phase. A GC thread copies the objects from (one or more) source regions to some empty destination region. The destination region may be originally empty, or may become empty because its objects have been copied elsewhere. This requires some synchronisation, which uses the pending queues. Initially, each GC thread adds to its pending queue a subset of the empty regions. For each destination region, the GC thread copies the corresponding objects from their source regions and updates the references contained in

[2] Note that the GC traverses each live reference exactly once, and hence, the number of reference accesses is equal to the number of references.

the destination region. When a source region becomes empty, it is added to the GC thread's pending queue, and so on recursively.

The **steal task** is the last task on the queue. When a GC thread reaches the steal task, this means that it has no other work to do, and should now steal references from randomly-chosen pending queues.

When, after several random steal attempts, the GC thread does not find any work, it synchronises with the other GC threads with a termination protocol. It increments a global counter, which counts the number of GC threads in the termination protocol, and waits for the other GC threads. While it waits, it regularly checks all the pending queues of all the other GC threads, and leaves the termination protocol by restarting the steal task if it finds work to perform.

2.4 Parallel Scavenge Baseline

The original Parallel Scavenge suffers from memory access imbalance on NUMA machines, which drastically degrades its performance [10]. Furthermore, it suffers from a synchronisation design flaw, unrelated to our study of memory placement [10]. Since it is a rather simple fix, and to ensure a fair comparison, we hereafter consider a slightly modified version of Parallel Scavenge, which we shall call Parallel Scavenge Baseline or PSB. Borrowing from NAPS [10], PSB corrects the synchronisation flaw, and uses an *interleaved* memory placement policy, in which pages are mapped from different physical nodes in round-robin. This ensures that memory is approximately uniformly allocated from all the nodes, and hence, memory access is expected to be uniformly and randomly distributed among all the nodes.

3. NUMA-friendly placement

Before turning to the GC algorithm itself, we present NUMA-friendly policies designed to optimise the memory placement for the mostly-distributed design. As stated in the introduction, since sending an inter-node reference is slightly more expensive than remotely accessing a small object, we aim to minimise the number of inter-node references, improving what we call the *spatial NUMA locality* property. Furthermore, also as explained in the introduction, *memory allocation balance* is important, to avoid saturating some memory nodes.

Placement occurs, either when the mutator first allocates an object, or when the GC copies an object to a new location. To be efficient, the placement decision should take less time than the benefit of improved memory placement. For instance, we found that the aggressive graph partitioning heuristics of Stanton and Kliot [26], although highly effective to minimise the number of inter-node references and to balance the memory allocation among the nodes, are much too costly for our purpose.

Therefore, we focus on simple policies that require only local information, are computed quickly, and have a small memory overhead. Placement may be somewhat suboptimal, but should remain satisfactory relative to the low computation overhead. In order to identify interesting policies, we first analyse the natural object graph topology created by several applications. Then, based on this analysis, we propose NUMA-friendly placement policies that leverage this topology.

3.1 Object graph analysis

In order to improve the spatial NUMA locality and the memory allocation balance, we first analyse the object graph of five applications, chosen for their diverse allocation patterns: Spark, a multi-threaded map-reduce engine [23]; Neo4j, an embedded, disk-based, fully transanctional Java persistence engine that manages graph data [17]; SPECjbb2013, a business logic service-side application benchmark that defines groups of threads with different behaviours [25]; SPECjbb2005, another business logic service-side application benchmark where all the threads have the same behaviour [24]; and H2, an in-memory database from the DaCapo 9.12 benchmark [4, 11].

We analyse the topology created by the applications in HotSpot 7 running 48 GC threads on an AMD Magny-Cours machine with 48 cores and eight memory nodes. We use a customised version of PSB, which ensures that the objects allocated by a mutator thread running on node i always stay on node i.[3] In this experiment, as in all the experiments presented in the paper, we let the Linux scheduler places the mutator threads on the cores. Table 1 reports the following metrics:

- The *heap size* that we set for the experiment.
- The *proportion of clustered references*, defined as the following ratio: the number of references between objects allocated by the mutator threads of the same node, divided by the total number of references. We report separately the proportion of clustered references between young objects, from old to young objects, and among all the objects. The first two are traversed during a young collection, while all the references are traversed during a full collection. In order to gather representative numbers, we ignore the first few collections to avoid warm-up effects, and thus measure the proportion of clustered references on a snapshot of memory taken during the 8^{th} full collection for Spark, 5^{th} for Neo4j, SPECjbb2005 and SPECjbb2013, and 3^{th} for H2.
- The *thread allocation imbalance*, defined as the standard deviation over the average number of objects allocated by the mutator threads running on each node.

We observe that the proportion of clustered references is always high, especially between young objects and from old to young objects. Indeed, our experiment reports a proportion between 77% and 100% between young objects, between

[3] This version of PSB is exactly the pure distributed algorithm described in Section 5.5.

	Heap size	Proportion of clustered references			Thread allocation imbalance
		Young	Old-to-young	All	
Spark	32GB	0.99	0.91	0.53	0.10
Neo4j	32GB	0.99	0.72	0.27	0.21
SpecJBB2013	24GB	0.99	0.99	0.51	0.07
SpecJBB2005	4GB	1.00	0.18	0.55	0.09
H2	2GB	0.77	0.16	0.50	0.40

Table 1. Analysis of the object graph

16% and 99% from old to young objects and between 27% and 55% for the other references, whereas, if clustering were random, we would expect a proportion of clustered reference equal to $1/8 = 12.5\%$ with eight nodes. This shows that the applications considered have a natural tendency to cluster their objects.

Furthermore, we observe that memory allocation imbalance varies between the applications, from highly balanced for Spark, SPECjbb2013 and SPECjbb2005, to imbalanced for H2. This confirms the need for placement policies to counter-balance the latter case.

3.2 NUMA-friendly placement

We have designed our placement policies based on the above observations. We can leverage the observation that the mutator threads of the same node tend to naturally cluster the objects that they allocate to the objective of spatial NUMA locality: all we need to do is to ensure that an object allocated by a mutator thread is placed on the same node as the mutator thread.

In addition to improving locality for the GC, this memory placement policy also has the interesting side-effect of improving application locality, because a mutator thread tends to access mostly the objects that it has allocated itself.

We also observed an imbalanced allocation pattern in some applications. Therefore, always placing all objects allocated by a mutator thread on the same node would be detrimental to memory allocation balance. Therefore, we should also design policies to alleviate this imbalance, by migrating objects from overloaded memory nodes to underloaded memory nodes during collection.

With this in mind, we designed four different and complementary policies:

- *Node-Local Allocation* places the object allocated by a mutator thread on the same node where the mutator thread is running.
- The *Node-Local Root* policy ensures that the roots of a GC thread are chosen to be located mostly on its running node.
- During a young collection, *Node-Local Copy* policy copies a live object to the node where the GC thread is running.
- During the compacting phase of a full collection, the *Node-Local Compact* policy ensures that an object being compacted remains on the same node where it was previously located.

The first policy ensures that a mutator thread allocates to its own node initially. The other three avoid object migration, so that the object stays where it was allocated. The middle two avoid object migration during a young collection because a GC thread on node i mainly processes objects of node i and copies them on node i. The node-local compact policy simply prevents object migration during a full collection.

For applications with an imbalanced allocation pattern, the Node-Local Copy policy, in conjunction with stealing, also tends to re-balance the load. Consider the case where Node A hosts more objects than Node B. Consequently, processing on Node B will finish sooner than Node A. As presented in Section 2, rather than remaining idle, the GC threads on Node B will start "stealing" i.e., to process remaining objects of Node A. The Node-Local Copy policy ensures that GC threads on B will copy the objects to B, restoring the balance.

3.3 Implementation details

In order to be able to map addresses to nodes, we make use of the *fragmented spaces* of NAPS [10]. A fragmented space is partitioned into *logical segments*, where the virtual address range of each segment is physically allocated on a different node. In contrast to an interleaved space (which maps pages randomly from all nodes, see Section 2.4), a fragmented space allows a GC thread to retrieve the segment and NUMA location of an object from its address. The GC thread can also place an object onto a specific NUMA node by using a segment mapped to the desired node.

NAPS implemented fragmented space for the young generation, with the goal of memory allocation balance. We extend NAPS implementation to the old generation, by adding a per-segment compacting algorithm, which applies the Node-Local Compacting policy of not migrating objects between nodes. Beside the goal of enforcing the spatial NUMA-locality and the memory allocation balance, we have also optimised the memory access locality during the compacting phase by ensuring that a GC thread selects the regions of its node before stealing the regions of the other nodes.

In order to implement the Node-Local Root policy, we partition the card table into segments. Each segment of the card table corresponds to a segment of the old generation, and hence identifies old-to-young roots from a given node.

Also for the Node-Local Root policy, we ensure that a GC thread processes the stack of a mutator thread running on its node, which mainly contains references allocated by the mutator thread, thus allocated on the same node, thanks to the Node-Local Allocation policy.

4. NumaGiC

NumaGiC focuses on improving memory access locality without degrading parallelism of the GC. A GC thread normally runs in *local mode*, in which it collects its own node's

memory only. It switches to *work-stealing mode* when parallelism degrades. In work-stealing mode, a GC thread can steal work from other threads, and is allowed to collect the memory of other nodes remotely.

We present the two modes and the conditions for switching between them.

4.1 Local mode

In local mode, a GC thread collects its local memory only. When processing a reference, the GC thread checks for the home node of the referenced object, *i.e.*, the node that hosts the physical memory mapped at the virtual address of the object. If the home-node is the node of the GC thread, then the GC thread processes[4] the object itself. Otherwise, it sends the reference to the home node of the object, and a GC thread attached to the remote home node will receive and process the reference. Checking the home-node of an object is fast, as it consists of looking up a virtual address in a map of segments of the fragmented spaces (see Section 3.3).

Moreover, when a GC thread idles in local mode, it may steal work from the pending queues (described in Section 2.3) of the other GC threads of its node, but not from remote nodes.

Communication infrastructure. Our initial design used a single channel per node, where other nodes would post messages; experience shows that this design suffers high contention between senders. Therefore, NumaGiC uses a communication channel per each pair of nodes, implemented with an array-based lock-free bounded queue [12].

Because multiple threads can send on the same queue, sending a message is synchronised thanks to an atomic compare-and-swap, a relatively costly operation. In order to mitigate this cost, references are buffered and sent in bulk. The evaluation of Section 5.4 shows that buffering 16 references for the full collections, and 128 references for young collections gives satisfactory performance.

A GC thread checks the receiving end of its message channels regularly, in order to receive messages: *(i)* when it starts to execute a GC task, and *(ii)* regularly while stealing from other threads of the same node.

4.2 Work-stealing mode

In work-stealing mode, a GC thread may steal from any node, and may access both local or remote memory. In work-stealing mode, a GC thread steals references from three groups of locations; when it finds a reference to steal in one of these groups, it continues to steal from the same group as long as possible, in order to avoid unsuccessful steal attempts. The first group consists of its own transmit buffers, cancelling the messages it sent that were not yet delivered. The second group consists of the receive side of other nodes' communication channels. The third group consists of the pending queues of other GC threads.

When a GC thread does not find references to steal, it waits for termination. Classically, asynchronous communication creates a termination problem [5]. For instance, even though a GC thread might observe that its pending queues and its receive channel are all empty, this does not suffice to terminate it, because a message might actually be in flight from another GC thread.

To solve this problem, a GC thread does not enter the termination protocol of the parallel phase (described in Section 2.3) unless it observes that all of the messages that it has sent have been delivered. For this purpose, before entering the termination protocol, it verifies that the remote receive side of every of its communication channels is empty, by reading remote memory.

After having observed that all its sent messages are received, a GC thread waits for termination, by incrementing a shared counter and regularly checking all termination.

4.3 Switching between local and work-stealing modes

A GC thread enters work-stealing mode when its does not find local work: when its local pending queue is empty, when its steal attempts from the pending queues of the GC threads of its node have failed, and when its receive channels are empty.

Once a GC thread is in work-stealing mode, it adapts to current conditions by regularly re-entering local mode. The rationale for this adaptive behaviour is twofold. First, local work can become available again if a stolen object or one of its reachable objects is local; in this case, it makes sense to switch back to the local mode, because the received reference will often open up a significant sub-graph, thanks to spatial NUMA locality. Second, re-entering local mode and back to work-stealing mode ensures that the GC thread will retry to steal from all the groups, regularly checking all sources of work again, especially its receive channels.

We tuned the frequency of re-entering local mode, finding that switching back to local-mode every 1024 stolen objects gives satisfactory performance in all our experiments.

5. Evaluation of NumaGiC

This section studies the performance of NumaGiC, using both standard benchmarks and widely-used, industrially-relevant applications. We first describe the hardware and software settings, followed by the applications used in the evaluation. Then, we study the impact of the policies presented in Section 3, and the impact of some design parameters and trade-offs on a small set of experiments. Finally, we undertake a full-scale evaluation of the impact of NumaGiC followed by an analysis of its scalability.

5.1 Hardware and software

We run our experiments with the Hotspot Java Virtual Machine of OpenJDK 7, on two different machines.

[4] Processing an object means copying it during young collection, and marking it live during full collection.

The first one, called *Amd48* hereafter, is a Magny-Cours machine with four AMD Opteron 6172 sockets, each consisting of two nodes. Each node has six cores (2.1 GHz clock rate) and 32 GB of RAM. In total there are 48 cores and 256 GB of RAM on eight nodes. The nodes are interconnected by HyperTransport 3.0 links, with a maximum distance of two hops. The system runs a Linux 3.9.7 64-bit kernel and gcc 4.6.1. NumaGiC is configured with 48 GC threads on Amd48.

The second machine, called *Intel80* hereafter, is a an Intel server with four Xeon E7-2860 processors, each consisting of a single node. Each node has 10 cores (2.27 GHz clock rate) and 128 GB of RAM. Each core carries two hyper threads. In total, there are 40 cores (80 hardware threads) and 512 GB RAM on four nodes. The nodes are interconnected by QuickPath Interconnect links, with a maximum distance of two hops. The system runs a Linux 3.8.0 64-bit kernel and gcc 4.7.2. NumaGiC is configured with 80 GC threads on Intel80.

5.2 Applications and Benchmarks

Our evaluation uses two big-data analytics engines, two industry-standard benchmarks, and two benchmark suites.

Spark Spark is an in-memory map-reduce engine for large-scale data processing [23]. We use Spark 0.9.0 to run a page-rank computation on a dataset consisting of a subset of a graph of 1.8 billion edges taken from the Friendster social network [8].

On Amd48, we use two configurations, one with 100 million edges, and the other with 1 billion edges. For the first one, we set the heap size to 32 GB. The computation triggers approximately 175 young and 15 full collections and lasts for roughly 22 minutes with PSB (Parallel Scavenge Baseline), which is sufficiently short to run a large number of experiments. This is also the configuration used for the evaluation of the memory-placement policies of Section 5.3. The second configuration is much more demanding; we run it with heap sizes increasing from 110 GB to 160 GB, in steps of 25 GB. On Intel80, we measure only the 1.8-billion edge configuration, with heap sizes ranging from 250 GB to 350 GB, in steps of 50 GB.

The database itself is stored on disk. Following the Spark developers' advice, we use the remaining RAM to mount an in-memory file system `tmpfs` for Spark's intermediate results. Thus, these experiments make use of all the available RAM. Furthermore, they run on all the cores of the machine.

Neo4j Neo4j is an embedded, disk-based, fully transactional Java persistence engine that manages graph data [17]. Neo4j caches nodes, relationships and properties in the Java heap, to accelerate computation. We implemented a driver program that uses Neo4j 2.1.2 as a library package, and queries the database for the shortest path between a given source node and 100,000 randomly-chosen destination nodes. We use the native shortest-path algorithm of Neo4j, and execute it in parallel using the fork/join infrastructure provided by the Java library. The database used for this experiment is also created from the Friendster dataset.

On Amd48, we use the first billion edges, and on Intel80, we use all the 1.8 billion edges. We run it with heap sizes ranging from 110 GB to 160 GB, in steps of 25 GB on Amd48, and from 250 GB to 350 GB, in steps of 50 GB, on Intel80. We follow the advice of the Neo4j developers, to leave the rest of the RAM for use by the file-system cache. This experiment also makes use of all the available RAM and all the cores.

SPECjbb2005 and SPECjbb2013 SPECjbb2005 and SPECjbb2013 are two business logic service-side application benchmarks [24, 25] that model supermarket companies. SPECjbb2005 only models the middle tier server, using a set of identical threads, each one modelling a warehouse; each warehouse runs in complete isolation. SPECjbb2013 models all the components of the company, using a larger number threads with different behaviours, interacting together.

For SPECjbb2005, we run a warm-up round of 30 seconds, then a measurement round of eight minutes. On Amd48 (resp. Intel80), we evaluate different heap sizes, from 4 GB to 8 GB, in steps of 2 GB (resp. from 8 GB to 12 GB, in steps of 2 GB). For SPECjbb2013, we let the benchmark compute the maximal workload that can be executed efficiently on the machines, which ensures that all mutator threads are working. On both Amd48 and Intel80, we evaluate different heap sizes, from 24 GB to 40 GB, in steps of 8 GB.

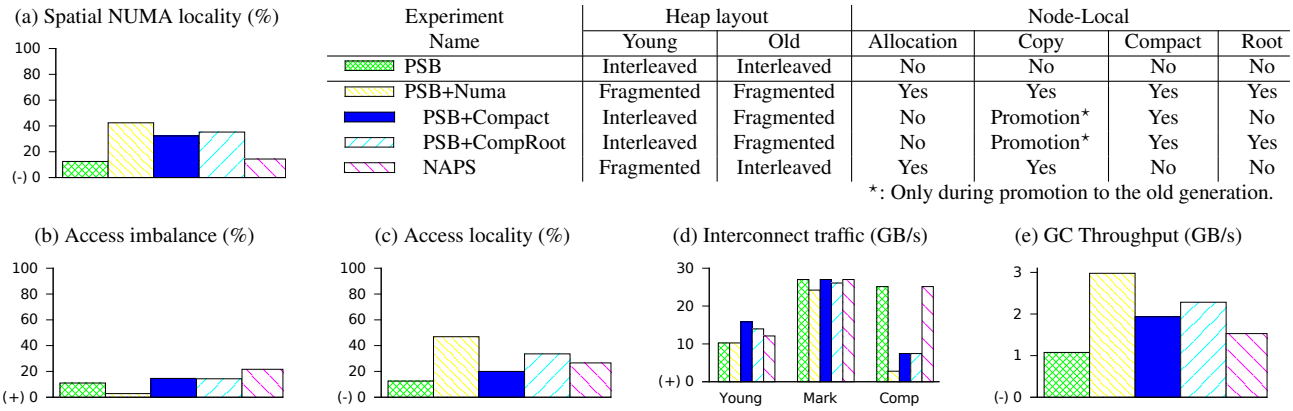
DaCapo 9.12 and SPECjvm2008 The DaCapo 9.12 and SPECjvm2008 benchmarks are widely used to measure the performance of Java virtual machines. They include 52 real applications with synthetic workloads; we retained all 33 that are multi-threaded.

For the DaCapo applications, we selected the largest workload. For the SPECjvm2008 applications, we have fixed the number of operations to 40 because this value ensures that all the mutator threads are working. We do not configure a specific heap size, instead, relying on Hotspot's default resizing policy. The experiments execute with one mutator thread and one GC thread per core.

The resulting heap sizes are the smallest of our evaluation and; therefore, we expect that the impact of GC on overall performance will be small. These benchmarks are also not representative of the big-data applications targeted by the NumaGiC design. Nonetheless, we include them in our evaluation, both for completeness, and to evaluate the performance impact of NumaGiC's multiple queues, which should be most apparent in such applications with a small memory footprint.

5.3 Evaluation of the policies

This section presents an evaluation of the placement policies discussed in Section 3.2. For this analysis, we focus on Spark with a 32 GB heap size running on the AMD Magny-Cours



Experiment Name	Heap layout		Node-Local			
	Young	Old	Allocation	Copy	Compact	Root
PSB	Interleaved	Interleaved	No	No	No	No
PSB+Numa	Fragmented	Fragmented	Yes	Yes	Yes	Yes
PSB+Compact	Interleaved	Fragmented	No	Promotion⋆	Yes	No
PSB+CompRoot	Interleaved	Fragmented	No	Promotion⋆	Yes	Yes
NAPS	Fragmented	Interleaved	Yes	Yes	No	No

⋆: Only during promotion to the old generation.

Figure 2. Evaluation of the NUMA-friendly placement policies. (+) = lower is better, (-) = higher is better.

with 48 cores and 8 nodes. The results are similar for the other evaluated applications and on both machines.

Figure 2 reports the following metrics from the experiment:

- *Spatial NUMA locality*: the ratio of number of local references (i.e., between objects on the same node) in the object graph to the total number of references. As for the graph analysis (see Section 3.1), we measure the Spatial NUMA locality on the snapshot of the object graph taken during the 8^{th} full collection. Higher is better.
- *Memory access imbalance*: for each GC thread, we compute the standard deviation of the ratio of number of accesses to each node over the total number of accesses, and then report the average. Lower is better.
- *Memory access locality*: the ratio of local memory accesses (read or write) performed by the GC threads, over the total number of memory accesses. Higher is better.
- *Interconnect traffic*: the average number of GB/s transmitted on the interconnect by a node during collection. We report separately traffic during the young collections, during the marking phases of the full collections, and during the compacting phases of the full collections. Lower is better.
- *GC throughput*: the number of gigabytes of live data scanned per second by the collector. Higher is better.

To characterise the effect of each policy, we compare different variants. The baseline is PSB (see Section 2.4). As indicated in the table of the figure, PSB+Numa extends PSB with fragmented spaces and the four NUMA-friendly placement policies described in Section 3.2. We observe that the policies improves spatial NUMA locality, as the PSB+Numa algorithm increases the fraction of local references from 12% to 42% (see Figure 2.a). We also observe that memory access balance is slightly better with PSB+Numa than with PSB, with a standard deviation of 2.8% vs. 11%, which is already low (see Figure 2.b).

In the rest of this section, the variants PSB+Compact, PSB+CompRoot and NAPS [10] enable a finer-grain comparison. PSB+Compact turns on fragmented spaces only in the old generation, does Node-Local Compacting during a full collection, and does Node-Local Copy when promoting from young to old generation, but not when copying from young to young generation. PSB+CompRoot is the same as PSB+Compact, with the addition of Node-Local Root policy. NAPS uses fragmented spaces in the young generation only, and enables the Node-Local Allocation and Node-Local Copy policies when copying from the young to the young generation, but not when promoting from the young to the old generation.

Thanks to the rebalancing effect of the Node-Local Copy policy used in conjunction with stealing, we observe that memory access balance remains good in all the experiments, with a 20% standard deviation in the worst case (see Figure 2.b). Consequently, we focus only on spatial NUMA locality in what follows.

Spatial NUMA locality Observe that the Node-Local Compact policy, in conjunction with the Node-Local Copy policy during object promotion, yields the highest single improvement of spatial NUMA locality, from 12% with PSB to 32% with PSB+Compact (see Figure 2.a). When a GC thread promotes an object from the young to the old generation, the Node-Local Copy policy copies a root object and its reachable sub-graph to the node of the GC thread, thus avoiding remote references in the object graph. Thereafter, the Node-Local Compact policy preserves this placement during full collections by preventing object migration. We have measured that 75% of the references are between objects of the old generation. For this reason, improving spatial NUMA locality in the old generation has a large impact on the overall spatial NUMA locality, since it concerns a large part of the heap.

Comparing PSB+CompRoot with PSB+Compact, we observe that the Node-Local Root policy alone does not have a significant effect on spatial NUMA locality. In PSB+CompRoot, the young space is interleaved. Memory pages in the

heap are allocated in round robin from every nodes, thus, old-to-young root references and young-young references are randomised between the nodes, exactly as in PSB+Compact.

Comparing PSB with NAPS, observe that the Node-Local Copy and Allocation policies alone do not have a significant effect on spatial NUMA locality. NAPS improves it inside the young generation, but old-to-young references are still random because the old space is interleaved. We have measured that only 6% of the references are inside the young generation in Spark; thus NAPS has only a marginal effect on spatial NUMA locality. NAPS uses these two policies, not to improve spatial NUMA locality, but only to improve memory access balance, because the Node-Local Copy policy, in conjunction with stealing, re-balances the load (as explained in Section 3.2).

Comparing PSB+Compact with PSB+Numa, observe that the Node-Local Copy and Allocation policies, when used in conjunction with the two other policies, improve spatial NUMA locality from 32% to 42%. In this case, old-to-young and young-to-old references, which concern 19% of the references, have a better spatial NUMA locality (roughly 50% of these references are local). This is because the conjunction of all the placement policies ensures that objects allocated by some application thread are allocated on its same node, and remain there when they are copied.

To summarise, this study shows that, individually, each of the policy does not have a drastic effect on spatial NUMA locality, but that, when they are used in conjunction, they multiply the spatial NUMA locality by 3.5×.

GC throughput improvements It is interesting to observe that our NUMA-friendly placement policies have the side-effect of improving GC throughput (Figure 2.e). This is because they also improve memory access locality (Figure 2.c).

Node-Local Compaction improves memory access locality, from 13% with PSB to 20% with PSB+Compact. As explained in Section 3.3, a GC thread first processes the regions of its node before stealing, which avoids many remote access during the compacting phase (Figure 2.c). This improved memory access locality drastically reduces interconnect traffic during the compacting phase (7.5 GB/s instead of 25.1 GB/s, see Figure 2.d).

Node-Local Root alone also improves memory access locality substantially, with 33% local access in PSB+CompRoot, compared to 20% for PSB+Compact (Figure 2.c). This is because a GC thread selects old-to-young root objects from its own node during a young collection. Moreover, when combined with the other policies, the Node-Local Copy and Node-Local Allocation policies ensure that a root object generally references objects on the same node. As a result, memory access locality improves substantially, from 33% with PSB+CompRoot, to 47% with PSB+Numa (Figure 2.c).

Overall, the improved memory access locality translates to better GC throughput, reaching 3.0 GB/s with PSB+Numa, compared to 1.0 GB/s for PSB (Figure 2.e).

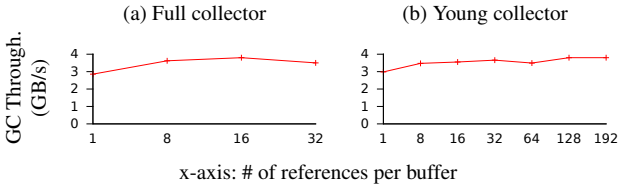


Figure 3. GC throughput of NumaGiC with varying transmit buffer sizes (higher is better).

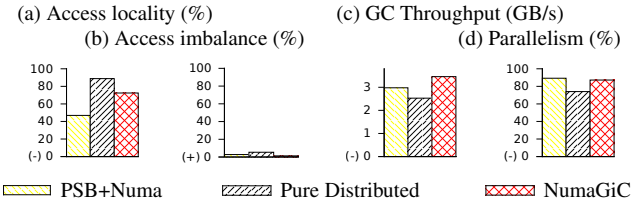


Figure 4. Memory access locality versus parallelism (-) = higher is better, (+) = lower is better

5.4 Impact of transmit buffer size

One of the internal design parameters of NumaGiC is transmit buffer size (see Section 4.1); we study its impact on performance. For this experiment, we run Spark with the small workload on Amd48. Figure 3 reports the GC throughput of NumaGiC, i.e., the number of gigabytes of live data collected per second, with varying transmit buffer sizes. Figure 3.a varies the buffer size used in the full collector, and sets a fixed size of 128 references for the young collector. Figure 3.b varies the buffer size used by the young collector, and sets the full-collector's buffer size to 16 references.

We observe that performance increases quickly with buffer size. However, the increase ceases around 16 references (resp. 128) for the full (resp. young) collector. Experiments with other applications, not reported here, confirm that these values give good performance, on both Amd48 and on Intel80. Therefore, we retain these values in the rest of our experiments.

5.5 Memory access locality versus parallelism

As stated in the introduction, enforcing memory access locality can be detrimental to parallelism. To highlight this issue, we run Spark with the small workload on Amd48 with three different GC set-ups: PSB+Numa, which adds all the NUMA-friendly placement policies of Section 3.2 to Parallel Scavenge Baseline (PSB), Pure Distributed, a version of NumaGiC in which a GC thread always stay in local mode, and full NumaGiC.

Figure 4 reports three metrics defined in Section 5.3, the memory access locality (a), the memory access imbalance (b) and the GC throughput (c). It also reports the parallelism of the GC (d), defined as the fraction of time where GC threads are not idle. We consider idle time to be the time in the

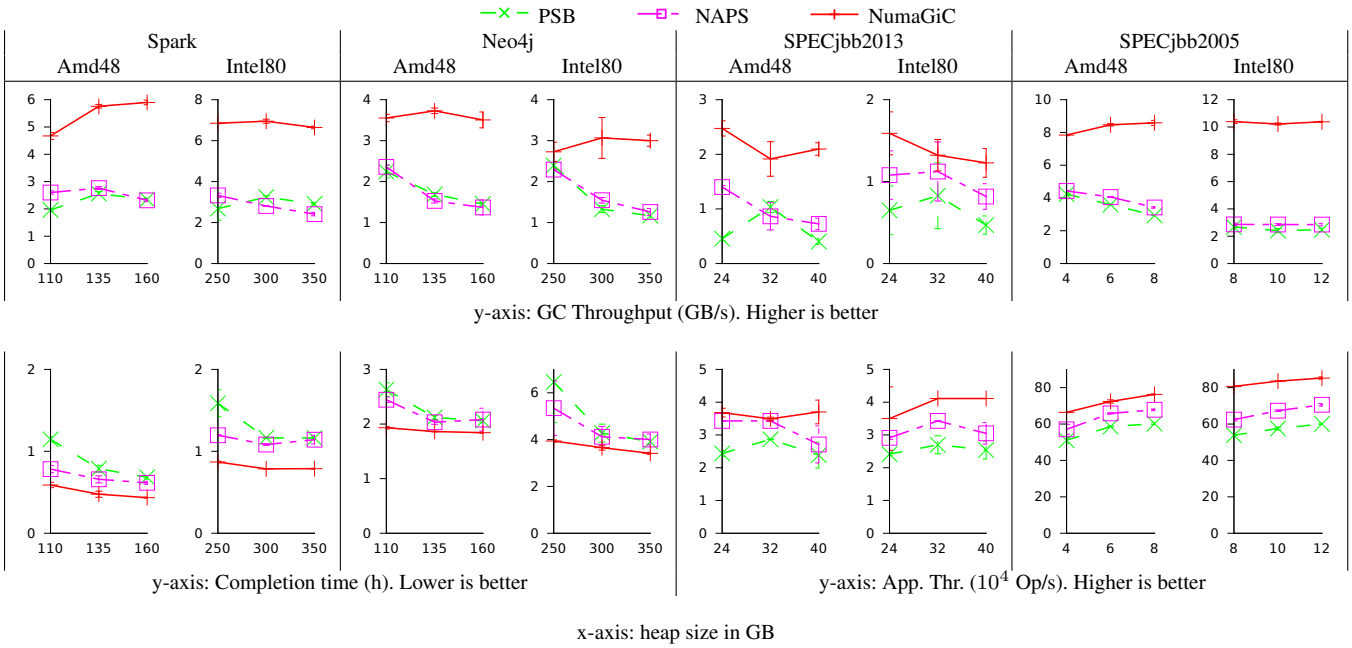


Figure 5. Evaluation of the applications with large heap.

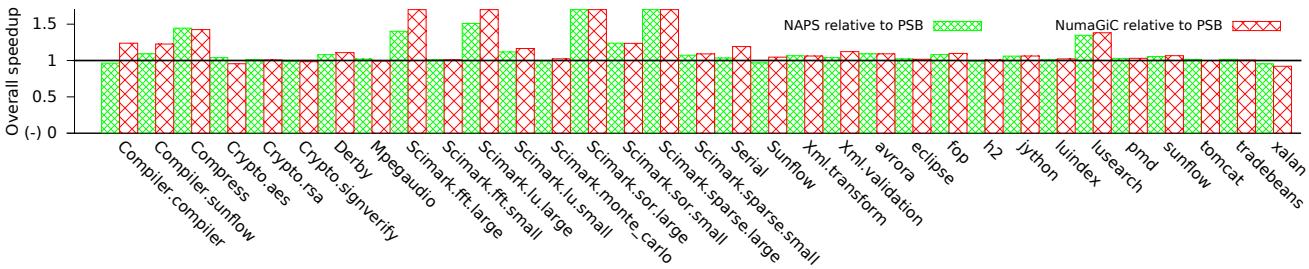


Figure 6. Evaluation of applications with small heap (DaCapo and SPECjvm2008). Higher is better.

termination protocol, *i.e.*, where it synchronises to terminate the parallel phase (see Section 2.3).

We can observe that memory access balance remains good with Pure Distributed (Figure 4.b). However, this observation is not significant. For example, memory allocation, and thus access, is imbalanced with H2, as shown in Table 1. Indeed, by construction, Pure Distributed algorithm does not migrate objects, and therefore, memory access balance by the GC is directly correlated to the allocation pattern of the mutator.

Moreover, for Spark, although Pure Distributed improves memory access locality substantially over PSB+Numa (Figure 4.a), throughput decreases, from 3.0 GB/s down to 2.5 GB/s (Figure 4.c). This is caused by degraded parallelism, because GC threads are idle 26% of the time in the Pure Distributed algorithm, against only 11% in PSB+Numa (Figure 4.d).

Analysing this problem, we observe a sort of a convoy effect, in which nodes take turns, working one after the other. Because the NUMA-friendly policies reinforce spatial NUMA locality, the object graph tends to structure itself into connected components, each one allocated on a single node, and linked to one another across nodes. We observe furthermore that the out-degree of object is small, 2.4 on average, and that the proportion of roots in the full object graph is around 0.008% (roughly 15,000 objects), also quite small. As a result, the number of connected components reached by the full collector at a given point in time tends to be relatively small, and sometimes even restricted to a subset of the nodes. In this case, some GC threads do not have any local work. Since, by design, the pure distributed algorithm prevents remote accesses, the GC threads remain idle, which degrades parallelism.

Observe that NumaGiC corrects this problem by providing the best of the PSB+Numa and the Pure Distributed algorithms. The parallelism of NumaGiC remains at the level of PSB+Numa (11% of idle time for PSB+Numa against 13% for NumaGiC, see Figure 4.d). At the same time, NumaGiC

degrades memory access locality slightly, compared to Pure Distributed (from 88% to 72% of local accesses, see Figure 4.a), but it remains largely improved over PSB+Numa (47% of local accesses). Thanks to improved memory access locality GC throughput increases from 3 GB/s in PSB+Numa, to 3.5 GB/s in NumaGiC (Figure 4.c).

5.6 Performance analysis

This sections studies the impact of NumaGiC, both on GC performance (throughput and duration of the GC phase), and on overall performance (how end-to-end application performance is impacted by NumaGiC).

Figure 5 reports GC throughput and overall speedup of Spark (under large workload on Amd48), Neo4j, SPECjbb-2005 and SPECjbb2013 on Amd48 and Intel80. We report the average and standard deviation over 3 runs.

As explained earlier, we vary the heap size. We were unable to identify the minimal heap size for Spark and Neo4j because the runs last too long. For example, on Amd48, the computation of Neo4j lasts for 2h37 with PSB and a heap size of 110 GB, but does not complete even in 12 hours with a heap size of 100 GB. For this reason, for Spark and Neo4j, we use the smallest heap size that ensures that the application terminates in less than 8 hours.

Observe that NumaGiC always improves the GC throughput over NAPS, up to 2.9× on Amd48 and up to 3.6× on Intel80, which translates into an overall improvement (application + GC) up to 42% on Amd48 and up to 45% on Intel80. With heap sizes that provide the best overall performance for all the GC (160 GB on Amd48 and 350 GB on Intel80 for Spark and Neo4j, 8 GB and 12 GB for SPECjbb2005, 40 GB and 40 GB for SPECjbb2013), the overall improvement ranges between 12% and 42% on Amd48 and between 12% and 45% on Intel80.

Figure 6 reports the GC throughput and the speedup, in terms of completion time, of the DaCapo and SPECjvm2008 benchmarks on Amd48. Since these applications have small workloads, we expect the performance impact of GC to be modest. Nonetheless, observe that, compared to NAPS, NumaGiC improves the overall performance of seven of the applications by more than 5% (up to 32% for Scimark.fft.large), does not change it by more than 5% for 25 of them, and degrades performance by more than 5% in a single application (Crypto.AES, degraded by 8%). For this application, GC throughput actually improves, but, as we let Hotspot uses its default resizing policy, this modifies the heap resizing behaviour, causing the number of collections to double. In summary, even for small heaps, 20% of the applications see significant overall improvement thanks to the the better memory access locality of NumaGiC, and with the sole exception of Crypto.AES, NumaGiC never degrades performance relative to NAPS.

To summarise, big-data applications with a large heap benefit substantially from NumaGiC. It significantly improves garbage collector throughput, which translates into a significant overall application improvement. With small heap sizes, NumaGiC is either neutral (most cases), rarely degrades performance somewhat (one application), and improves overall performance in 20% of the evaluated applications. Furthermore, the fact that the improvements remain similar on two different hardware architectures tends to show that our mostly-distributed design is independent from a specific hardware design.

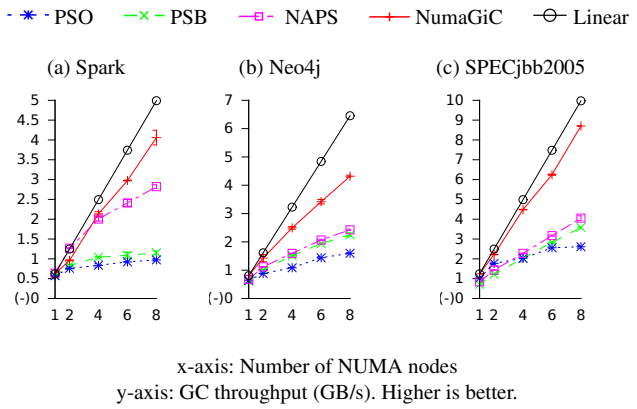


Figure 7. Scalability for three applications with large heap

5.7 Scalability

This section studies the scalability of the garbage collectors discussed in this paper, as the number of NUMA nodes increases. The experiments were conducted on Spark (40GB), Neo4j (32GB), and SPECjbb2005 (6GB). SPECjbb2013 is not included, as its performance exhibits a large standard deviation at low core count, and hence does not provide stable results.

Figure 7 shows the scalability of GC throughput of the original Parallel Scavenge distributed with Hotspot (PSO), our improved version of Parallel Scavenge (PSB), NAPS, and NumaGiC. For comparison, we also plot linear throughput relative to the 1-node performance of NumaGiC.

As shown in the figure, NumaGiC scales substantially better than all the other three GCs. NumaGiC scales almost linearly with SPECjbb2005, slightly less with Spark, and again less with Neo4j. This variation of scalability is directly related to the proportion of clustered objects of each application, as shown in Table 1. An application such as SPECjbb2005 has a high proportion of clustered objects, or in other words low inter-node sharing, which leads to higher scalability. In contrast, Neo4j has a lower proportion of clustered objects, hence lower scalability. In this case, more inter-node shared objects cause more inter-node messages in NumaGiC, which in turn degrades performance.

6. State of the Art

Our work is inspired by multi-kernels [3, 18] and clustered operating systems [22] for multi-core machines. These systems partition memory between cores or NUMA nodes, and

communicate via messages. The partitioning avoids remote memory accesses and ensures memory access locality. Our starting point is similar: NumaGiC partitions a Java Virtual Machine's address space between nodes, and isolates nodes during collection, in order to avoid remote memory accesses. However, we showed in Section 5.5 that strict partitioning degrades parallelism, and hence performance. Therefore, we take a pragmatic approach; we relax the local access constraints to avoid leaving GC threads idle, and leverage remote access when beneficial to performance.

The best throughput-oriented stop-the-world GC in the standard HotSpot JVM is Parallel Scavenge, which scales poorly on large-scale multiprocessors [9]. An improved version of Parallel Scavenge called NAPS was proposed by Gidra et al. [10]. NAPS improves memory access balance on a ccNUMA machine. However, as our evaluation shows, the memory access locality of NAPS is poor. Thanks to its adaptive distributed design, NumaGiC has better memory access locality, which translates to a better performance.

Zhou and Demsky [31] designed a parallel NUMA-aware mark-and-compact collector on their proprietary Java Virtual Machine, targeting the TILE-Gx microprocessor family. Zhou and Demsky target relatively small heap sizes, between 1.5 GB and 4 GB, whereas we consider big-data applications with heap sizes of more than 150 GB. Similarly to NumaGiC, in their collector, each core collects its own local memory, and sends messages when it finds remote references. The study of Zhou and Demsky considers only a mark-and-compact algorithm, similar to the one used in the old generation of NumaGiC. In contrast, NumaGiC is a generational GC with two generations with multiple spaces, which complicates the design, especially at the boundary between the spaces. Moreover, Zhou and Demsky focus on the locality of the application and the GC. They do not consider balancing memory access among the nodes, but recent studies showed that unbalanced memory access has a dramatic impact [6, 10]. They do not consider improving spatial NUMA locality, whereas we observe that a distributed design is beneficial only if the number of remote references is low. Finally, although Zhou and Demsky state that restricting GC threads to accessing only local memory can degrade parallelism, they do not observe it in their evaluation, and thus do not propose a solution. Our own evaluation shows that this issue is central to performance.

Ogasawara [19] and Tikir and Hollingsworth [28] study how NUMA effects impact performance of the language runtime. They copy an object to the memory node where it is accessed the most. This aims to improve application locality, but does not address GC locality, as we have done with NumaGiC.

A widely-used technique to improve concurrency is the thread-local heap [1, 7, 16, 20, 27]. With this technique, when first allocated, an object is placed in a heap local to the creating mutator thread. Later, it migrates to a shared global heap only if it becomes referenced from the shared heap. This heap layout supports collecting a thread-local heap concurrently with the mutator threads on the other cores. The partitioned eden space in NumaGiC is similar because it allows a mutator thread to allocate its memory locally, but it has a radically different goal. However, our partitioned eden space could probably be used as a building block towards NUMA thread-local heaps collected currently.

In order to improve memory access balance and locality for native applications, some recent work proposes to migrate, or even to replicate, whole memory pages by observing the memory access pattern of the application at the operating system kernel level [6, 15]. NumaGiC does not use this technique, because, in order to avoid fragmentation, a copying GC continuously moves objects around in memory, obfuscating the memory access patterns observed by the operating system kernel.

7. Conclusion

We presented the rationale and the detailed design of NumaGiC, a mostly-distributed, adaptive GC for ccNUMA architectures. NumaGiC is designed for high throughput, to be used for long-running applications with a large memory footprint, typical of big-data computations.

NumaGiC's distributed design avoids the costly remote accesses of non-NUMA-aware GCs. However, NumaGiC pragmatically uses remote access when it is beneficial to performance, and switches between the local mode (avoiding remote access) and the work-stealing mode (maximising parallelism), according to the current workload. Thanks to this design, NumaGiC demonstrates high memory access locality without loss of parallelism, which translates to a drastic reduction of interconnect traffic, and to a substantial improvement of GC throughput, which translates to an overall speedup between 12% and 45% over NAPS on the two ccNUMA machines for applications with large heaps.

As future work, we plan to study a concurrent GC for ccNUMA architecture, where each node could collect its memory in isolation. Indeed, in applications where responsiveness is important, suspending the application for long GC pauses is not tolerable. These kinds of applications require a concurrent collector, which runs concurrently with the application. Because they need to synchronise at a fine grain with the mutator (to detect changes to the object graph), concurrent collectors are considerably more complex. HotSpot comes with a choice of collectors, including concurrent ones. We observe that they all suffer from poor locality, as they access remote memory nodes roughly 80% of the time, on the 48-core machine. The techniques used in this study to avoid remote access are most likely applicable to a concurrent collector as well. Indeed, they appear unrelated to the concurrent collector's synchronisation with application, since they concern mainly the mapping of the heap on the different nodes and the communication between the GC threads.

References

[1] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM '10*, pages 21–30. ACM, 2010.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *SP&E*, 19(2):171–183, 1989.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09*, pages 29–44. ACM, 2009.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190. ACM, 2006.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 3(1):63–75, 1985.

[6] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *ASPLOS '13*, pages 381–394. ACM, 2013.

[7] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93*, pages 113–123. ACM, 1993.

[8] Friendster. SNAP: network datasets: Friendster social network. `http://snap.stanford.edu/data/com-Friendster.html`, 2014.

[9] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *SOSP Workshop on Programming Languages and Operating Systems, PLOS '11*, pages 1–5. ACM, 2011.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *ASPLOS '13*, pages 229–240. ACM, 2013.

[11] H2. H2 database engine. `http://www.h2database.com/`, 2014.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[13] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 1st edition, 2011.

[14] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–429, 1983.

[15] LinuxMemPolicy. What is Linux memory policy? `http://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt`, 2014.

[16] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *ISMM '11*, pages 21–32. ACM, 2011.

[17] Neo4j. Neo4j – the world's leading graph database. `http://www.neo4j.org`, 2014.

[18] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP '09*, pages 221–234. ACM, 2009.

[19] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA '09*, pages 377–390. ACM, 2009.

[20] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. Eliminating read barriers through procrastination and cleanliness. In *ISMM '12*, pages 49–60. ACM, 2012.

[21] P. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. Technical report, Cambridge, MA, USA, 1988.

[22] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *EuroSys '11*, pages 61–76. ACM, 2011.

[23] Spark. Apache Spark– lightning-fast cluster computing. `http://spark.apache.org`, 2014.

[24] SPECjbb2005. SPECjbb2005 home page. `http://www.spec.org/jbb2005/`, 2014.

[25] SPECjbb2013. SPECjbb2013 home page. `http://www.spec.org/jbb2013/`, 2014.

[26] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD '12*, pages 1222–1230. ACM, 2012.

[27] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM '00*, pages 18–24. ACM, 2000.

[28] M. M. Tikir and J. K. Hollingsworth. NUMA-aware Java heaps for server applications. In *IPDPS '05*, pages 108–117. IEEE Computer Society, 2005.

[29] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE '84*, pages 157–167. ACM, 1984.

[30] P. R. Wilson and T. G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Notice*, 24 (5):87–92, 1989.

[31] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In *ISMM '12*, pages 3–14. ACM, 2012.

This research was supported in part by ANR (France) projects ConcoRDanT (ANR-10-BLAN 0208), Infra-JVM (ANR-11-INFR-008-01) and STREAMS (ANR-2010-SEGI-010-02), and by COST Action IC1001 Euro-TM.