THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Lokesh GIDRA

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Garbage Collector for memory intensive applications on NUMA architectures

soutenue le 28 septembre 2015

M. Gaël THOMAS	Directeur de thèse
M. Marc SHAPIRO	Directeur de thèse
M. Vivien QUEMA	Rapporteur
M. Luis VEIGA	Rapporteur
M. Albert COHEN	Examinateur
M. Tim HARRIS	Examinateur

To my loving father

Acknowledgements

First and foremost, I would like to express my gratitude to my advisors Gaël Thomas, Marc Shapiro, and Julien Sopena, for their essential guidance and insight throughout my graduate career. Their support and encouragement throughout the thesis have been very helpful, and without our many fruitful discussions the result would surely not have been as good.

I would also like to thank my thesis jury for spending their valuable time to review and improve the thesis quality. I am really thankful to them.

My sincere and warm thanks goes to all my colleagues at Regal group in LIP6 Lab for their inspirations and helpful comments during my thesis. I especially thank Florian David, Brice Berna, Suman Saha, Pierpaolo Cincilla, Gauthier Voron, Maxime Lorrillere, Thomas Preud'Homme, and Sergey Legtchenko for devoting some of their precious time to help me on numerous occasions during my work.

Special thanks to my wife, Vidhi for being encouraging and patient at every stage during my doctoral studies. I kept her from so many holiday trips due to my thesis.

Needless to say, my family deserves a great deal of credit for my accomplishments. I thank my parents and sisters for all their love and support. I especially thank my father for his innumerable sacrifices to ensure my continuous professional growth.

I would like to thank again Gaël Thomas and Florian David. Without their support, I could not have completed my PhD.

Finally, I would like to thank God for giving me the ability to do this work.

Abstract

Large-scale multicore architectures create new challenges for garbage collectors (GCs). 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. In this thesis, 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 and increase memory access imbalance, by allowing threads to steal from other nodes when they are idle. NumaGiC strives to find a perfect balance between local access, memory access balance, and parallelism.

In this work, we compare NumaGiC with Parallel Scavenge and some of its incrementally improved variants 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 94% over Parallel Scavenge, and increases the performance of the collector itself by up to 5.4× over Parallel Scavenge. In terms of scalability of GC throughput with increasing number of NUMA nodes, NumaGiC scales substantially better than Parallel Scavenge for all the applications. In fact in case of SPECjbb2005, where inter-node object references are the least among all, NumaGiC scales almost linearly.

Contents

Chapter 1

Introduction

Large applications like data analytics engines [42], databases [32], scientific applications, application servers and web servers have huge hardware resource requirements. The amount of data to process and/or the number of user requests to serve, pushes these applications to easily consume all the CPU, memory and I/O bandwidth available on a server machine, even requiring multiple such machines in some cases.

Managed Runtime Environments (MREs) such as Java and C# are being increasingly preferred for development of such applications due to the speed, portability and safety of development that they provide. Improvements in techniques like Just In Time (JIT) compilation and automatic garbage collection (GC) have made them competitive with other programming languages. Therefore, as with any system software, for these applications to perform efficiently, it is essential for the MRE and its components to be efficient.

One of the performance critical component of MREs is garbage collector (GC). A garbage collector reclaims memory allocated to dead (unreachable) objects so that it can be used for new allocations. To identify the dead objects, it traverses through the entire object graph, starting from global and stack variables, which are called *roots* in GC context. Objects which are unreachable during the graph traversal would also be unreachable to the application. Therefore, they are considered dead and thus their memory is safely recycled. The garbage collector is also responsible for compacting the heap in order to defragment the memory. It does so by relocating all the live objects to a separate heap location.

The cost of a garbage collection cycle[1] depends on the amount of live data it processes, which in turn depends on the memory requirement of the application. Furthermore, since the majority of large applications are memory intensive, a considerable amount of time is spent in doing GC which makes the GC performance even more critical. This thesis studies performance behavior of GC for large applications on contemporary hardware architectures.

Unlike Symmetric Multi-Processors (SMP), which used to have a *flat* architecture with uniform access latenices from one hardware location to any other location, the contemporary multi-core architectures, called cache-coherent Non-Uniform Memory Access (cc-NUMA) architectures, have a

[1] A collection cycle is one iteration of garbage collection out of all the iterations that are triggered during the entire execution of the application.

hierarchical design. At the first level there are various nodes, each consisting of multiple cores, a last level cache, at least one memory controller which is responsible for a set of memory modules, and finally a system bus which connects all these components. At higher level there is a network which inter-connects these nodes to facilitate inter-node communication. This way the hardware hides the physical distribution of memory from the software, which is necessary to support legacy multi-threaded applications which assume a shared memory architecture. On such architectures, memory access latency to access a node-local memory location is much smaller as compared to accessing a memory location on a different node.

There exists many parallel garbage collectors to exploit multiplicity of processors [12, 15, 16, 35, 48, 52]. However, they were developed in the era of SMPs. Therefore, it is unclear how the design shift in hardware from SMPs to cc-NUMA architectures will impact the GC's performance. In this thesis, I intend to find answer to this question. More precisely, I want to answer the following questions:

1. What is the impact on GC's performance when utilizing the existing parallel garbage collectors on cc-NUMA architectures?

2. If the existing GCs fail to scale on contemporary hardware, then what are the bottlenecks?

3. Can these existing garbage collectors be redesigned to fix these bottlenecks? What would be an ideal GC design for NUMA architecture?

A wide spectrum of real applications and benchmarks are evaluated to determine whether the garbage collectors continue to improve their performance with every advancement in computer hardware. The garbage collectors of the widely used Java virtual machine, called HotSpot, which is part of OpenJDK Java Development Kit were chosen. The experiments were conducted on a 48-core 8-node AMD machine. However, the evolution of hardware is emulated by running experiments on varying number of CPU cores, starting from 1 to 48, with a fixed size increment at every step. The results showed that after a certain number of cores, the GC throughput, which is the amount of live data processed per unit of time, kept decreasing with increasing number of cores. Two sources of inefficiencies have been experimentally identified: idling CPU cores, and excessive inter-node traffic.

Idling CPU cores. Since the existing parallel GCs were developed in the era of SMPs, their internal data structures use synchronization primitives which scale well up to few threads, but drastically hurts the scalability when a lot of GC threads are used, which is the case on contemporary hardware with many cores. Today, the GC threads spend embarrassingly large amount of time in synchronizing and very little time in doing the operation on the data structure. The issue was resolved by replacing the GC-internal data structures with efficient lock-free versions. Using lock-free data structure improved the scalability during the parallel execution of collection. Another synchronization related problem is with wake-up/suspension of GC threads at start/end of the parallel phase of GC. To resolve this issue, the thread management code is simplified to ensure that thread wake-up/suspension is done with minimal synchronization. Together these two solutions have completely removed the explicit waiting of GC threads during the parallel phase of the garbage collector, making the phase lock-free [20].

Excessive inter-node traffic. Experiments revealed that a substantially huge amount of inter-node remote access is done during garbage collection. The following two reasons are identified as the main culprits:

1. **Non-local memory access:** A cc-NUMA hardware 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, it silently traverses inter-node references and thus processes objects on any memory node. Consequently, GC threads often access remote memory, which causes a huge amount of inter-node traffic and slows down memory access.

2. **Unbalanced memory access:** When the physical memory associated with the object heap is allocated from few nodes, but is accessed by threads running on all of them, it saturates the memory controller(s) of these particular node(s), severely hurting scalability. This problem is faced by GC threads as they are deployed on all the nodes. This problem arises due to a very common application behaviour, where the applications have a *serial* initialization phase. This phase forces a majority of physical pages (corresponding to the heap) to be allocated from only those few nodes, where the initialization thread(s) run. Furthermore, since the same heap is reused after every GC cycle, the problem is faced during every collection.

To overcome both problems, we propose NumaGiC [21]. First, NumaGiC solves the problem of non-local memory access by using a mostly-distributed design. At the basis of NumaGiC, we observe that the problem of non-local memory access can be solved by ensuring that a GC thread only processes objects located on its own memory node. When a GC thread discovers an object located on a different node, it could notify a GC thread located on the object's home node, which continues the scan locally. However, strictly applying this solution, unfortunately, degrades parallelism, because a GC thread remains idle when it does not have local objects to collect, waiting for GC threads on other nodes to reach some object(s) located on its own node. NumaGiC is thus designed to aim for memory access locality during the GC, without degrading GC parallelism, by letting each GC thread switch between two modes of execution. A GC thread starts with *local mode*, in which the GC focuses on memory access locality. In this mode, the GC threads strictly follow the above described constraint. When a GC thread runs out of local objects to process, it enters *thief mode*. This mode focuses on parallelism and allows a GC thread to "steal" references to process from other nodes, and to access such references itself, even if they are remote. A GC thread in thief mode regularly 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.

Second, to solve the issue of unbalanced memory access, and also to prepare the heap for the mostly-distributed design, NumaGiC includes a set of NUMA-friendly placement policies. The policies first ensure that the GC balances the memory among all the nodes, which solves the problem of unbalanced memory accesses. This ensures that every node contributes in memory utilization, and also to equalise the collection work between all GC threads. The policies also aim at minimising the number of inter-node references. Since sending a reference is slightly more costly than remotely accessing the associated object, sending a reference is beneficial only if, on average, the referenced object itself references many objects on its own node. In this case, the cost of sending the reference gets amortised over the memory access locality of the receiving GC thread.

NumaGiC is implemented in the Hotspot Java Virtual Machine of OpenJDK 7. It targets long-running computations that use large data sets, for which a throughput-oriented *stop-the-world* GC algorithm is suitable[2]. NumaGiC is based on Parallel Scavenge, the default throughput-oriented GC of Hotspot. Experiments compare NumaGiC with Parallel Scavenge. NumaGiC is evaluated against two widely-used big-data engines, Spark [42] and Neo4j [32], with Java heaps sized from 110 GB to 350 GB. We also evaluate two industry-standard benchmarks, SPECjbb2013 [44], and SPECjbb2005 [43], along with the DaCapo 9.12 and SPECjvm2008 benchmarks. The experiments were conducted on a 48-core 8 node AMD machine, having 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. The 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 two evaluated GC, NumaGiC improves the overall performance of applications by 12% to 62% over Parallel Scavenge. This result shows that a mostly distributed design increases substantially the performance of applications with large heap on NUMA machines, and that a mostly distributed design seems to improve performance independently of the architecture.

- NumaGiC scales well with the number of nodes. At constant heap size, the GC throughput, *i.e.*, the number of live bytes processes per time unit, increases with the number of nodes. As compared to Parallel Scavenge, NumaGiC never degrades performance.

- On applications with large heap with the most efficient heap size for all the GC, NumaGiC increases collector throughput on the two machines by 2.2–5.2×, compared to Parallel Scavenge.

- On 33 applications from DaCapo 9.12 and SPECjbb2008 with smaller working set, NumaGiC improves substantially the performance of 19 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 more than half of the applications with modest workload.

Organization of the document. The thesis is structured as follows:

- Chapter 2 gives background information required for understanding this research. It will cover detailed explanations of the contemporary NUMA architectures and Parallel Scavenge garbage collector, which is the baseline of our work. The chapter also presents State-of-the-art in the field of this research.

- Chapter 3 establishes our problem statement. We present our throughput evaluation of all the garbage collectors of OpenJdk7. The chapter then does a scalability analysis of Parallel Scavenge, our baseline garbage collector.

- Chapter 4 presents our solution to the problem of idling CPU cores during garbage collection. This chapter evaluates our solution on real benchmarks.

- Chapter 5 presents NumaGiC, a distributed garbage collector that we have developed to counter the problem of excessive inter-node traffic. It also compares NumaGiC's performance with the baseline GC on a wide range of applications.

[2] 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 [25], which favours response time at the expense of throughput, because it requires fine-grain synchronisation between the application and the GC.

- Finally, Chapter 6 concludes the thesis and discusses future research directions.

Chapter 2

Background and Related work

2.1 Parallel Scavenge GC

A garbage collector (GC) observes the memory of an 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 [25]. This means that it suspends the application during a collection, spawns parallel GC threads, and segregates objects into generations, under the observation that "most objects die young", *i.e.*, that in many programs, most objects become garbage within a few GC cycles after their creation [3, 27, 50].

2.1.1 Heap Layout

A generational collector segregates the heap into multiple generations, and by collects the young generation more frequently than the older one(s). As presented in Figure 2.1, Parallel Scavenge defines three generations: a small *young* generation; a larger *old* generation; and a *permanent* generation, similar to the old one, but much smaller.

Parallel Scavenge has two kinds of collections. A *young collection* collects only the young generation; a *full collection* collects all the generations. Parallel Scavenge leverages the generational hypothesis by performing young collections much more frequently than full collections.

The young generation of Parallel Scavenge is divided into three spaces, an *eden space* and two *survivor spaces*, called *from-space* and *to-space*. When a mutator, i.e., an application thread, allocates an object, it resides initially in the eden space. Later, depending on *age* of the object, it is either retained in young generation by copying it in the survivor spaces, or is promoted to the old generation. This decision is taken during young collection.

The old and the permanent generations consist each of a single space. The old space contains objects that were promoted from the young generation, and the permanent space contains the Java class definitions. Hereafter, we conflate the permanent with the old generation to simplify the presentation.

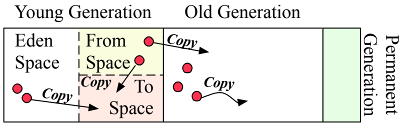


Figure 2.1 – Memory Layout of Parallel Scavenge

2.1.2 Object Allocation

To avoid synchronisation between the mutators at each object allocation, a mutator first allocates, for its exclusive use, a large memory chunk from the eden space called a Thread-Local Allocation Buffer (TLAB). Thereafter, it allocates objects from its TLAB, without synchronisation, using a bump pointer. Similarly, promoting an object to the survivor space or to the old generation allocates from Promotion-Local Allocation Buffers (PLAB), which are allocated from the to-space and the old generation.

2.1.3 Parallelism in Parallel Scavenge

Parallel Scavenge uses a master-slave design. The master thread, called the VM Thread, queues some GC tasks, in a shared task queue. Then it starts the parallel phase, in which GC threads dequeue tasks and perform the corresponding work. The various GC-tasks related to different parallel phases of Parallel Scavenge are described below. However, the discussion of thread management is left for chapter 4 to keep the explanation simple.

2.1.4 Young Collection

A young-generation collection is triggered when an allocation fails both to find space in the current TLAB and to get a new TLAB from the eden space. 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.

When a young-generation collection begins, the eden space contains those objects which are allocated since last collection; the from-space contains those that survived a few collections, and 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 2.1). 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.

2.1.4.1 GC Tasks

In order to parallelize the collection, Parallel Scavenge divides the collection work into multiple *GC tasks*, which can be executed in parallel by multiple GC threads. In the case of young-generation collection, there are three kind of tasks: root tasks, steal tasks and a single final task. Root tasks contain the entry points of the object graph. Steal tasks are used to balance the load between the GC threads. They are ordered in the queue after the root tasks, and are fetched by GC threads once they have fetched all the root tasks. Final task will be discussed in chapter 4 where we will discuss the GC thread management of Parallel Scavenge.

Root tasks. A root task provides an entry point into the graph of live objects. 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 [40, 51]. 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.

A GC thread performs a depth-first traversal (DFT) of the graph of live objects, starting from the addresses provided by the root tasks. For each object that a GC thread reaches, it creates a copy in the PLAB appropriate for the object's age, and installs a forwarding pointer to the new object in the old object's header. To avoid copying the same object concurrently with another thread, the GC thread uses an atomic compare-and-swap (CAS) instruction to install the forwarding pointer. Then, the GC thread pushes all the references contained in the object into a local *DFT queue*, which is implemented as a lock-free bounded queue, backed by an overflow stack. After this, the GC thread repeatedly pops a reference from its DFT queue, which it processes in the same way. When its DFT queue is empty, the GC thread picks either another root task, or eventually a steal task.

Steal tasks. As the sub-graph reachable from a root task depends on mutator activity, the load of the GC threads could be unbalanced, resulting in some GC threads remaining idle, while others still having a large amount of work. To better balance the load, an idle GC thread "steals" a reference from a randomly-chosen DFT queue and processes the corresponding sub-graph.[1] To avoid conflicting concurrent accesses to a DFT queue, the queue owner pushes and pops from the queue head, whereas other GC threads pop (steal) from the tail.

A GC thread may be unable to steal, either because the parallel graph traversal is terminated, or because its random choice never picked an overloaded GC thread even though one exists. To handle the second case, the GC thread does not directly leave the steal task. Instead, after a number of unsuccessful steal attempts, it enters a *termination protocol*. It first atomically increments a global thread counter in order to indicate to other GC threads that it is in the termination protocol. Then, the GC thread actively polls the global thread counter in a bounded loop. If the counter reaches the number of GC threads, this means that all BFT queues are empty and therefore that the parallel phase is terminated. In this case, the GC thread leaves the termination protocol. Otherwise, if the counter has not reached the number of GC threads in the bounded loop, the GC thread "peeks" into the other BFT queues. If all queues are empty, the GC thread also leaves the termination protocol, otherwise, it atomically decrements the global thread counter and continues to steal.

[1] Stealing is done from the queue only. The overflow stack is accessed solely by the owning thread.

2.1.5 Full Collection

If an object promotion fails at any stage of a young-generation collection because the old space is full, this indicates that the old generation needs to be collected. The GC thread that suffers the promotion failure sets a flag and continues the parallel phase. After the parallel phase, if the flag is set, the VM thread starts an old-generation 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.

2.1.5.1 Marking Phase

Marking phase's functionality is very similar to the young collection. Like young collection, marking also involves traversing through the object graph, visiting each live object. However, rather than copying every visited object, marking involves atomically setting a bit in the mark bitmap indicating that the object is alive, and updating summary data of the 4KB region which contains the object. These data structures are used by the following two phases. Furthermore, the old-to-young reference root set is not used in marking, as it is done on all the live objects, unlike young collection where only young objects are supposed to be traversed.

2.1.5.2 Summary Phase

Summary phase is a sequential phase executed by VM thread. It further updates the summary data of every region with additional information which is required for parallel compaction. Further details of this additional information are not discussed as they are not required for this research work, and would uneccessarily complicate the description.

The generational design of Parallel Scavenge ensures that the objects that are promoted to old generation have a high life expectancy. This phenomenon, along with compaction ensures that long living objects get accummulated towards the beginning of old generation. Therefore, progressively with every full collection, a portion of old generation gets densely populated with live objects, and hence the utility of compacting this dense portion diminishes accordingly. To overcome this, summary phase computes a *dense-prefix* of old generation. This dense-prefix indicates the boundary between the dense portion and the rest of the old generation, so that the portions can be handled differently in an efficient manner.

2.1.5.3 Compact Phase

Compact phase moves all the live objects in the entire heap towards the beginning of old generation, leaving a large contigous free space eliminating external fragmentation. This phase also updates all the references to the new location in all the live objects. The following GC-tasks are used by compact phase to parallelize the whole process:

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

Dense-prefix task As described in the summary phase, the utility of compacting the dense portion in the beginning of old generation is very little. Therefore, the objects in this dense portion are not moved, only their references are updated. This job of updating references of dense prefix objects is performed by *dense-prefix task*. Multiple tasks, each updating a sub-portion of the entire dense portion are utlized to parallelize this operation.

Steal task serves exactly the same purpose as in the case of young collection and marking phase, to balance the workload among GC threads. However, since compaction is performed on region basis rather than object, the regions (and not objects) are stolen from other GC threads' pending queues. Apart from this difference, the entire functionality is identical to that of young collection's steal task.

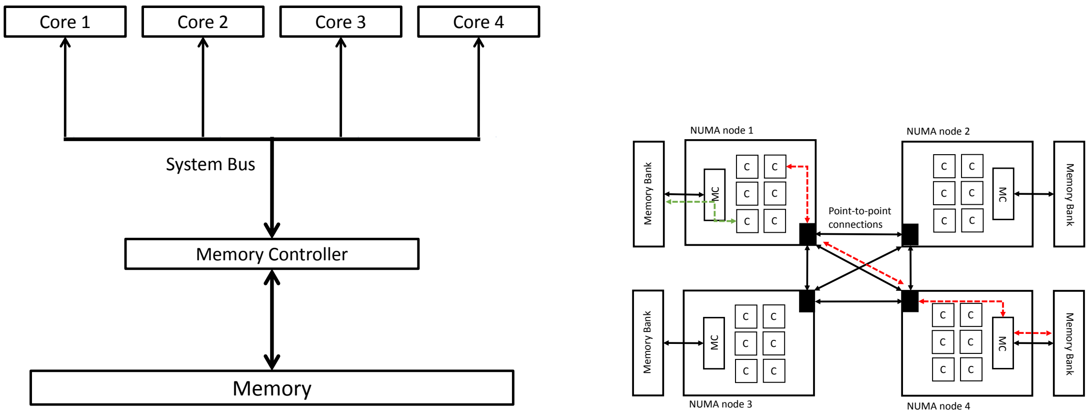
A sequential **post-compact** phase follows the compact phase, which is executed by the VM thread. Firstly, this phase updates references in all those objects which overlap with region boundaries. Such overlapping regions are not updated during compaction to avoid synchronization issues. Secondly, this phase updates the card-table. If the entire young generation is empty after the full collection, then the card-table is reset. Otherwise, all its bits are set. Finally, this phase clears the summary data of all regions and mark bitmap so that they can be used in the next full collection.

2.2 ccNUMA architecture

This section describes the characterstics of ccNUMA architecture which is commonplace in contemporary multicore architectures. Section 2.2.1 compares the legacy *Uniform Memory Access* architectures with *Non Uniform Memory Access* architectures. Section 2.2.2 describes AMD-48, the machine used for evaluation in this work. Section 2.2.3 discusses the impact of different memory placement techniques on application performance.

2.2.1 UMA vs. NUMA

Originally, when multicore machines appeared in the market, they were typically called *Symmetric multiprocessors* (SMP). In SMPs, a shared system bus, called front-side bus, served as the immediate link between the CPU, which contained all the cores, and all other devices in the system, including main memory. In this design, since a single shared system bus links the cores with memory, the cost of accessing any memory address from any CPU core is uniform. Therefore, SMPs had what is typically called *Uniform Memory Access* or simply UMA architectures. Figure 2.2a shows a simple block diagram of a SMP machine.



(a) UMA Architecture

(b) NUMA Architecture

Figure 2.2 – UMA and NUMA archs

The main issue with UMA architectures is that with every additional core, the traffic load on the shared bus, and main memory would keep increasing linearly. Therefore, after a certain core count, it is not possible for fixed bandwidth system bus and memory to serve all the cores as per their demand, and thereby stals the cores more often, wasting CPU cycles. This is how the shared hardware resources become bottlenecks and stop scaling with the increasing core count.

This problem with UMA architectures led to design of more complex architectures called *Non-Uniform Memory Acess* architectures. In NUMA architectures, the CPU cores are partitioned into multiple subsets, and each subset being grouped with an on-chip memory controller which controls a dedicated memory bank, into a single *NUMA node*. Finally, all the NUMA nodes are connected to each other and I/O controller using multiple point-to-point connections like AMD's HyperTransport and Intel's QuickPath Interconnect (QPI). This way the contentious shared system bus is replaced with multiple point-to-point connections, and main memory is partitioned multiplying the total memory bandwidth n times with n partitions in place. Block diagram of a NUMA architecture is depicted in figure 2.2b. Since a given CPU core has different distances to different memory banks, the memory access latencies are accordingly different, which is why these architectures are called Non-Uniform Memory Access. Accessing a local node memory bank is much faster as compared to accessing other nodes' memory banks.

In order to ensure backward compatibility for software, NUMA architectures often provide a shared memory model with system-wide cache coherency. This is why these architectures are also referred to as ccNUMA (*cache-coherent Non-Uniform Memory Access*). The hardware handles mapping memory addresses to the right NUMA nodes: typically, the range of memory addresses is split into as many contiguous chunks as there are nodes, and addresses from chunk n are all mapped to the n th NUMA node. If a hardware thread needs to access a memory address that is located in a remote NUMA bank, then requests have to be sent on the interconnect to the node that owns that NUMA bank, possibly with several hops if there is no direct interconnect link between the two nodes. This indirection increases latency: local accesses are faster than remote accesses.

In figure 2.2b, each die is a NUMA node that is connected to its local memory bank. An example of local NUMA access is shown in green: when a hardware thread reads data from memory, that data is copied into its L2 and L1 cache from which it can access it. When a hardware thread needs to access

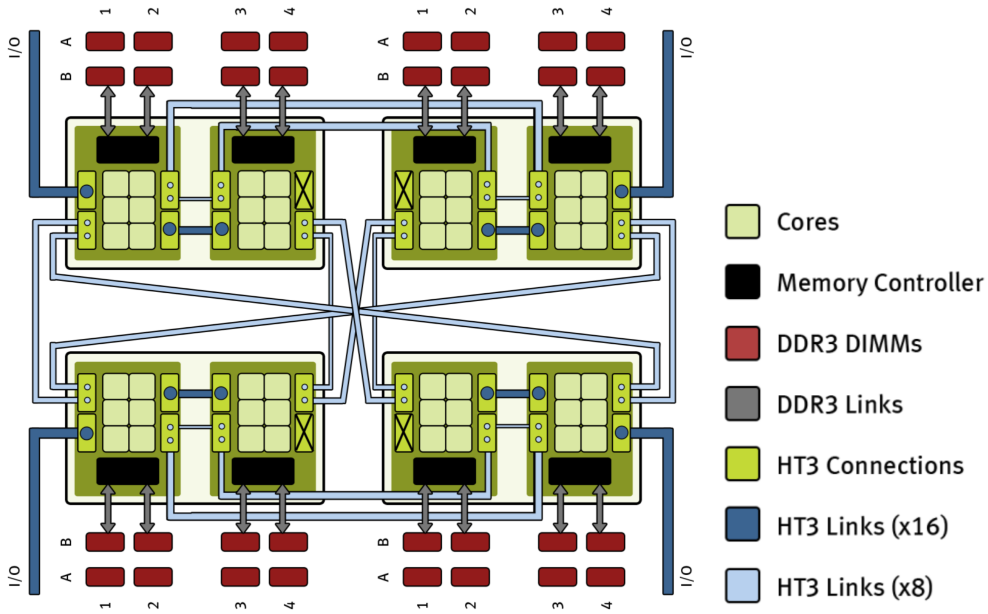
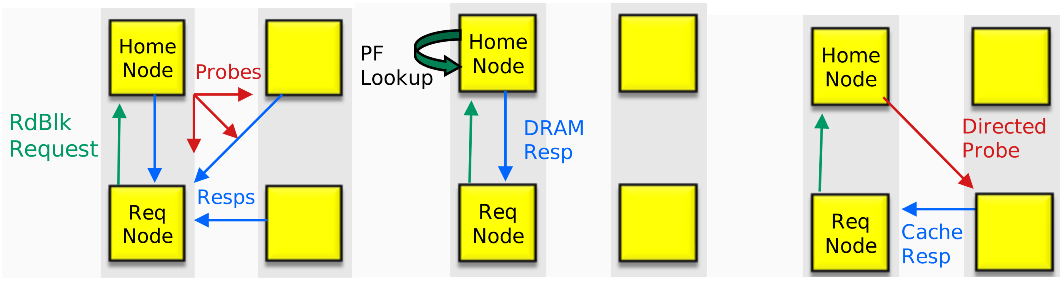


Figure 2.3 – Source: siliconmechanics.com

remote data, however, it must request it to the die whose NUMA bank holds the data. In the figure, any two dies have a direct interconnect link that connect them, therefore, one hop is sufficient. This is not always the case: requests sometimes have to be forwarded across several nodes in more complex NUMA architectures, which comes at an increased latency cost.

2.2.2 Machine used in the evaluation: AMD-48

AMD-48 is an x86 machine with four AMD Opteron 6172 CPUs (the Opteron 6100 series is codenamed “Magny-cours") [10]. The CPUs' clock speed is 2.100GHz. Each of the CPUs has twelve cores split across two dies: AMD-48 features 48 cores in total. AMD-48 provides one hardware thread per core as there is no hardware multithreading. Each core has a local L1 and L2 cache, while the L3 cache is shared among all six cores on the die. Each core has two dedicated 2-way set associative 64KB L1 caches, one for instructions and one for data, for a total of 128KB of L1 cache memory. Each core also has a 16-way set associative 512KB cache that contains both instructions and data. The six cores on each die share a 48-way set associative 6MB L3 cache. However, effectively only 5MB can be used by software as 1MB is dedicated for *probe filter* (explained in section 2.2.2.1). All caches have a 64-Byte cache line size. Each die is a NUMA node, therefore, AMD-48 has eight NUMA banks. Each bank handles 32GB of 1.333GHz U/RDDR3 memory, for a total of 256GB quad-channel main memory. The interconnect links between the eight dies do not form a complete graph: each die is only connected to the other die on the same CPU and to three remote dies. Therefore, the diameter of the interconnect graph is two: inter-core communications (fetching cache lines from remote caches, or NUMA accesses, for instance) require at most two hops. The bandwidth of the interconnect links between intra-CPU dies is three times that of links between inter-CPU dies. The structure of the interconnect graph can be seen in figure 2.3, along with the rest the architecture of AMD-48. The interconnect uses HyperTransport 3.0 links with a theoretical peak bandwidth of 25.6GB/s at 6.4GT/s (GigaTransfers per second).



(a) Broadcast (Snooping) (b) HTAssist (Clean data) (c) HTAssist (Dirty data)

Figure 2.4 – Snooping vs. Directory-based inter-node cache coherency mechanisms

2.2.2.1 HyperTransport Assist

Traditionally, multicore architectures use broadcast messages when a core requests for a cacheline which is not found in its local caches and last-level cache. The requesting core broadcasts the request on the shared bus, and waits for cacheline as response from all the cores and memory controller. In case of a NUMA architecture, when a thread t_1 in NUMA node n_1 needs to access data from a cache line that is stored in the caches of a core in NUMA node n_2 , with the data being allocated in the RAM of node n_3, t_1 sends a directed request to the home node of the requested cacheline, which is node n_3. Thereafter, node n_3 broadbcasts probe request to all the nodes in the system. Meanwhile, node n_1 waits until it receives responses from all the nodes. This scenario is depicted in figure 2.4a. Since this scenario is fairly common, the resulting probe requests increase the load on caches and interconnect links, which may lead to non-negligible overhead. To prevent this, AMD Opteron CPUs use an optimization known as HyperTransport Assist [10] (a.k.a. HT Assist). With HT Assist, part of the highest level cache of die d_3 , whose memory controller handles node n_3 , holds a *probe filter* (or *cache directory*) which maps every cached line which belongs to node n_3 with nodes that hold the line. Thanks to the probe filter, n_3 on receiving the request from t_1 looks up in its probe filter for the corresponding entry. Since the entry stores n_2 as the node containing the requested line, node n_3 sends a directed probe to node n_2 instead of broadcast, which then responses back to node n_1 directly. This ensures that all communication is point-to-point, which is more efficient than broadcasting requests to all caches in the hope that one of them will respond with the cache line. This scenario is shown in figure 2.4.

This probe filter is fully inclusive i.e., there must be an entry for every line from that node which is cached in the system. If the probe filter is full, then some previous entry must be replaced (causing a potential writeback, plus invalidation of the previous entry's data from all caches) to accommodate a new request. On AMD-48, the size of probe filter is 1MB, and it can only cover 16MB of cache. This means that the amount of data that can be cached from a particular memory controller is limited to 16MB, and hence unbalanced memory access beyond the 16MB limit may cause cache invalidations due to probe filter's capacity misses. To verify this, we conducted an experiment wherein multiple threads read sequentially from thread-local buffers of 512KB size. The size is carefully chosen so that the buffer fits within the core-local L2 cache. We compare three different memory mappings for buffers to be allocated: (i) on node 0, (ii) on the same node as the accessing thread, and (iii) with pages from all nodes in round robin.

In theory, there should not be any impact of buffers' memory location, because the buffer should fit entirely in the L2 cache. Therefore, when experimented with 15 threads, in all the three cases, operation finished in the same amount of time. This is becasue 15 buffers of 512KB each is lessar than 16MB, the limit of probe filter's capacity. The theory is proved true even when experimented with 45 threads for (ii) and (iii). However, in case of (i), the operation took almost 10× more time to finish than (ii) and (iii). This is because 45 buffers of 512KB each is larger than 16MB, and hence the probe filter invalidations started taking place due to capacity constraints.

2.2.2.2 Access latencies on AMD-48

NUMA architectures offer non-uniform access latencies from any CPU core to different locations in memory hierarchy. Boyd-Wickizer et al. use a set of tools in their paper about Corey [7] to measure various metrics about their hardware. In particular, they provide an application named Memal that loads cache lines in the requested level in the memory hierarchy of a given core c_1, and reads them from another given core c_2, in order to measure the cost of a load operation. Many cache lines are read and the benchmark returns the average access time. We used Memal to identify different access latencies on AMD-48.

Access latencies from a core to its local L1 and L2 caches are 3 and 12 cycles respectively. Furthermore, accessing node-local L3 cache and memory banks costs approximately 40 cycles and 140 cycles respectively. Moreover, when a core reads a line which is cached in the L1 or L2 cache of another core in its own node, the access latency is same as that of L3 cache access, i.e. 40 cycles.

Inter-node access latencies are expected to be more than intra-node ones. Rightly so, inter-node L3 cache and memory access costs approximately 214 cycles and 240 cycles respectively for nodes that are 1-hop away, and 297 cycles and 330 cycles respectively for nodes that are 2-hop away. Interestingly, the difference between L3 cache and memory access latencies is only 20 cycles in case of inter-node access. Whereas, in case of intra-node access, this difference increases substantially to 100 cycles.

Please note that the inter-node L3 cache access latencies discussed in previous paragraph are for the cases where the cached line is located on the L3 cache of the line's home node. This is not always the case. The cache lines which contain shared data can be cached on a different node than its home node. In such cases, the HT Assist's 3-way communication on the AMD-48 machine can lead to 5-hop response in worst case. This translates to latency of approximately 375 cycles.

2.2.3 Influence of memory placement

In order to understand the influence of different memory placement on performance, we performed an experiment on AMD-48. This experiment measures the impact of memory access imbalance and locality, the main factors that impact performance [11].

Figure 2.5 summarises this evaluation. It contains one curve per placement algorithm, plotting the speedup in completion time of a fixed number of operations executed by a varying number of threads. Each operation consists of sequentially reading from a thread-local buffer that does not fit in the CPU caches, triggering a physical memory access on every read. Threads are pinned on the nodes with a round-robin policy. The placement algorithms are as follows:

- **Unbalanced-Remote.** All the buffers are allocated on Node 0. Access is (i) remote (except for the threads running on Node 0), and (ii) unbalanced.

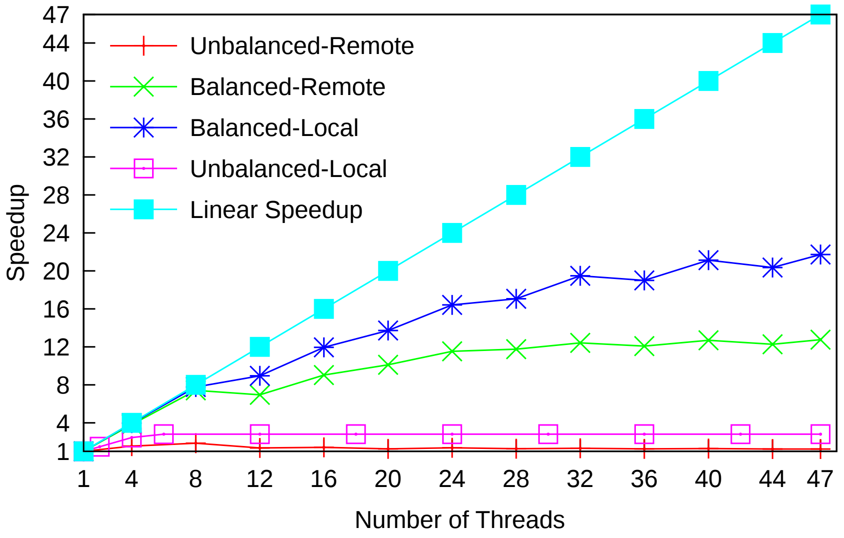


Figure 2.5 – Micro-benchmark showing performance impact of different memory placements.

- **Balanced-Remote:** The pages of all the buffers are allocated on all the nodes in round robin. Access is (i) mostly remote, and (ii) perfectly balanced.

- **Balanced-Local:** Every buffer is allocated on the same node as its corresponding thread. Access is (i) purely local, and (ii) perfectly balanced.

- **Unbalanced-Local.** The threads are first pinned on Node 0, and only these ones participate. The curve is extended beyond Node 0 by adding idle threads. Access is (i) local only, and (ii) unbalanced.

As shown in Figure 2.5, the performance of *Unbalanced-Remote* improves until 8 threads only, whereas *Balanced-Remote* continues to improve up to 32 threads. We can conclude from this that when load is unbalanced, fewer threads suffice to saturate the machine (either the memory controller(s) and/or the interconnect between the nodes). This experiment shows that balancing the load has a huge impact on scalability. Comparing *Unbalanced-Remote* with *Unbalanced-Local* shows that balancing load is a higher priority than improving locality.

Furthermore, *Balanced-Remote* stops improving beyond 32 threads, whereas *Balanced-Local* continues to improve up to 40 threads, and has significantly better results. We conclude that improving memory locality constitutes a secondary scalability objective, once balancing access among nodes has been taken care of.

Balanced-Local also shows that even with a perfect balance and a perfect locality, the memory controller eventually saturates, and that we can only expect to scale up to the 48 cores if the memory accesses to the memory nodes are not too frequent. As the only difference between the two experiments is the use of the interconnect in *Balanced-Remote*, we can also conclude that *Balanced-Remote* stops scaling before *Balance-Local* because of the interconnect saturation.

2.3 Applications/Benchmarks and JVM used for evaluation

2.3.1 HotSpot Java Virtual Machine

HotSpot Java Virtual Marchine is part of OpenJDK, mostly widely used Java Development Kit. We run all our experiments on OpenJDK7.

2.3.2 Spark

Spark is an in-memory map-reduce engine for large-scale data processing [42]. We use Spark 0.9.0 to run a page-rank computation on a dataset consisting of a graph of 1.8 billion edges taken from the Friendster social network [18]. 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.

For the scalability tests, we run with 40GB heap with first 100 million edges of the friendster dataset.

2.3.3 Neo4j

Neo4j is an embedded, disk-based, fully transactional Java persistence engine that manages graph data [32]. 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. For the scalability tests, we use 32GB heap with first 100 million edges of the friendster dataset.

2.3.4 SPECjbb2005

SPECjbb2005 is a business logic service-side application benchmark [43] 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. We run a warm-up round of 30 seconds, then a measurement round of eight minutes. For the scalability tests, we use 6GB heap size.

2.3.5 DaCapo

The DaCapo 9.12 benchmarks are widely used to measure the performance of Java virtual machines. We selected the largest workload for all the DaCapo applications and all the applications are run with 100MB heap size.

This section described those applications/benchmarks which are evaluated in all the following chapters. Furthermore, the heap size and dataset size mentioned above is for the scalability tests in the following chapters. For any other tests, the configuration details will be given in that chapter. Moreover, wherever any additional application/benchmark is used for evaluation, it will also be described in the corresponding chapter.

2.4 Related work

The advent of multicores have greatly changed the way we program software. From OS kernels to application software, every layer of the software stack has been impacted. In this chapter we discuss this change in software development as a consquence of the hardware evolution. Furthermore, since the goal of our work is to improve the performance of garbage collectors on contemporary multicore architectures, we will discuss the existing work in this field in great detail.

2.4.1 Kernel and Operating Systems

Majority of computers today are built using multicore processors, and future core counts, due to moore's law, are bound to further increase [6]. However, increase in core count alone cannot provide the speedup that we would expect to get. The other hardware components like on-die network and memory bandwidth also need to evolve. This neccessity has led to complicated architectures like ccNUMA at present, and maybe something even more complicated in future. This trend demands the systems research community to rethink if we need new OS techniques for future multicore hardware, or if we simply need to apply existing techniques used in large multiprocessor systems in legacy systems.

Before diving into the specific research contributions, it is worthwhile to broadly discuss different aspects of contemporary and future multicore architectures that influence systems research. These architectures encourage *shared access* of resources like system bus, memory controllers, program variables etc. These shared resources become points of *contention* hindering system's scalability. Hence, avoiding contention is very critical and therefore majority of research projects in this domain have tackled this issue. *Remote access* and *access imbalance* discussed in section 2.2.3 are two causes which render hardware resources like interconnect and memory controllers contentious.

Another aspect to consider is of *heterogeneous architectures*. It has been shown that to effectively exploit the increasing transistor count on the chips, instead of deploying multiple homogenous cores, we should deplay few fast, and possibly complex, cores with a lot of slow, and possbily simple, cores on a single chip. These different kinds of cores on the same chip may even not have the same instruction set. Furthermore, special hardware components like Network interface card (NIC) and GPUs are becoming more and more programmable. This allows, for instance, to run a network-bound application to directly run on the NIC, eliminating the entire cost of bringing network packets close to the general-purpose cores. These heterogeneous architectures opens up areas of research on how to effectively exploit this heterogeneity.

Furthermore, it is believed that the increasing number of cores and the use of specialized co-processors, like GPUs and programmable NICs, will make it impossible to have a system-wide cache-coherency that is maintained by the hardware. The Intel Single-Chip Cloud Computer (SCC) [30] has demonstrated how a non-cache-coherent system will look-like. This opens up another challenge for systems software to deal with such a scenario.

Many new research operating systems have been proposed for future multicore hardware, mostly focusing on eliminating *contention*. K42 [2] and Tornado [19] are designed to reduce it and improve locality for cache-coherent NUMA architectures. These systems introduced clustered objects (each cluster being an OS service), which optimize access to shared data through the use of partitioning and replication techniques. However, it relies on cache-coherency provided by the hardware for communication among different replicas. Neither of the two systems deal with the heterogeneity issue mentioned above. Similarly, Corey [7] advocates reducing contention within the OS by allowing applications to specify sharing requirements for OS data, effectively relaxing the consistency of specific objects. It does so by exposing APIs to applications to avoid unnecessary sharing of kernel state. Its abstractions (address range, kernel core, and share) ensure that each kernel data structure is used by only one core by default, while giving applications the ability to specify when sharing of kernel data is necessary.

Unlike K42, Tornado, and Corey, which rely on hardware's cache-coherency for inter-core communication, Barrelfish [4] is a shared-nothing distributed operating system. It uses a multikernel model, which distributes replicated kernels on every core and uses message-passing instead of shared-memory to maintain their consistency. Furthermore, Barrelfish strives to attain fine-grained understanding of application requirements for good scheduling decisions. Furthermore, the authors have demonstrated that the multikernel design is easily adaptable to heterogeneous architectures.

Similar to Barrelfish, Helios [33], also aims at bridging the heterogeneity of different processing units in a platform by using satellite kernels, which provide the same abstractions across different processor architectures. However, unlike all the above systems, Helios does not try to reduce contention. Instead it strives to ensure ease of developing, deploying and tuning applications on heterogeneous platforms. Interestingly, even without trying to eliminate performance hurdles like contention, Helios gains substantial performance improvements by effectively exploiting programmable hardware components like NICs and GPUs.

All the research efforts that we have discussed so far has built the systems from scratch. Boyd-Wickizer et al., however, argue that there is no scalability reason to give up a traditional operating systems yet [8]. They analyze Linux kernel on a 48-core cache-coherent NUMA machine with a set of applications that are designed for parallel execution and stress kernel services. The authors show that with standard parallel programming optimizations in the kernel and applications, most of the kernel bottlenecks which are stressed by the applications can be removed.

In Cerberus paper [41], the authors make a bridge between two different approaches (i.e., designing new OSes and refining comodity OSes) of scaling operating systems. Like Helios and Barrelfish, Cerberus makes use of replicated kernels and state. However, it differs from existing work mainly in that it aims at improving performance scalability of existing applications by using a backward-compatible technique called OS clustering. Under this technique, Cerberus runs multiple commodity OS instances on top of a virtual machine monitor to host one application. Furthermore, it provides the POSIX programming interface to shared-memory applications. This way, the existing multi-threaded applications require little or no porting effort.

Unlike all the above discussed systems which intend to improve kernel scalability (which certainly in-turn improves application performance too), Carrefour proposes a new memory management algorithm for cache-coherent NUMA systems, implemented inside Linux kernel, which directly impacts applications' performance by improving memory access [11]. The authors argue that member access locality has been the design focus of existing operating systems that target NUMA architectures. However, modern NUMA hardware has much shorter remote access delays. On the other hand, congestion on memory controllers and interconnects caused by memory traffic of memory-intensive applications can have much larger impact. Furthermore, the authors showed that memory access imbalance has considerably larger performance impact than remote access. To fix these issues, Carrefour's algorithm uses techniques like migration and replication of memory pages among NUMA nodes to avoid memory traffic hotspots to ensure congestion free use of memory controllers and internconnect links.

We completely support Carrefour's argument about congestion of memory controllers and inter-

Papers	Characterstics			Description
	Eliminates Contention	Handles Non Cache-Coherency	Handles H/W Heterogeneity	
K42, Tornado, and Corey [2, 7, 19]	Yes	No	No	Avoid sharing by partitioning/replicating data
Barrelfish [4]	Yes	Yes	Yes	Avoid sharing using share-nothing model
Helios [33]	No	Yes	Yes	Ensures ease of development and deployment on heterogeneous arch
Boyd-Wickizer et al. [8]	Yes	No	No	Linux made to scale using standard parallel programming techniques
Cerberus [41]	Yes	No	No	Reduce sharing by deploying multiple comodity OSes on a VMM
Carrefour [11]	Yes	No	No	Memory management algo providing good local access without compromising balance

Table 2.1 – Summary of research papers on operating systems for contemporay multi-core hardware.

connect being more severe problem than access locality. That is why NumaGiC has a *mostly* distributed design instead of a *purely* distributed, prioritizing congestion freedom of memory controllers and interconnect links, over remote access. However, we identified that a third factor, *parallelism*, also plays a more important role than locality, and there is always a trade-off between locality, and parallelism. Therefore, NumaGiC does not hesitate from doing occassional remote accesses, for which it pays slightly higher cost of remote access, to ensure balanced memory access among NUMA nodes, and also enough work for all GC threads sustain good parallelism.

Table 2.1 summarises all the research work that has been discussed in this section.

2.4.2 Garbage Collectors

2.4.2.1 Parallel Garbage Collectors

Being a very resource intensive operation, garbage collection must be very efficient and must finish as quickly as possible. On multi-core architectures, one straight-forward way of finishing GC quickly is to parallelize the operation. A *parallel garbage collector* generally, when the process runs out of free heap space, stops all the application threads, then launches multiple GC threads, which then finish the GC operation at the earliest possible, and then the application threads are resumed. Because of this reason, they are also known as *stop-the-world* garbage collectors. Parallel garbage collectors have been researched since a long time. Jones et al. discuss various techniques in great detail [25]. ParallelScavenge, the baseline GC of our work, is a parallel garbage collector which originates from [16].

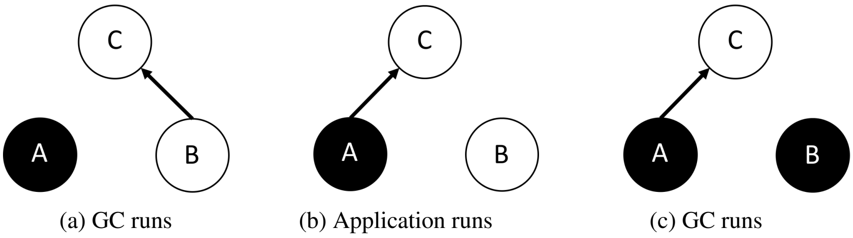
2.4.2.2 Concurrent Garbage Collector

Although parallel garbage collectors manages to reduce the GC time considerable by parallelizing the process, but time-critical applications sometimes have even more strict time constraints. Specially for applications which has large working set size, GC pauses incurred by parallel GCs may be completely unacceptable. To deal with this issue, *concurrent garbage collectors* were invented to avoid (or at least reduce) the time when application threads are stopped for garbage collection. This is done by making GC thread(s) work *concurrently* with the application threads for majority (if not entire) of the garbage collection time[2].

There has been much research on concurrent garbage collection, offering varying degree of concurrency. Concurrent garbage collectors has to following two functions:

2.4.2.3 Concurrent marking.

In order for marking to work concurrently with the application threads, it is necessary that the garbage collector marks all object(s) which are supposed to be live by the end of GC. However, the concurrency introduces a race condition, which may cause GC to miss some live objects [14, 46]. For example, in the scenario presented in Figure 2.6, C will be freed even through it is still used by the application. To tackle this race condition, concurrent GCs use *write barriers* which intercept all updates to references. For example, in Figure 2.6, when the reference to C is written in A, the write barrier informs the collector that C must be scanned.



(a) GC runs (b) Application runs (c) GC runs

Figure 2.6 – Black means scanned. The GC scans A in 2.6a, then the application modifies the graph in 2.6b, finally, the GC scans B but does not reach C in 2.6c because A is already scanned.

2.4.2.4 Heap Compaction.

While concurrent marking is quite straight-forward, compacting the heap concurrently is very complex. The complexity arises due to the fact that compaction involves *moving* the live objects. While moving an object, there is a time window during which there exists two copies of the same object. During this time window, the application thread might modify the old copy, thereby making the new copy inconsistent with the old one. Furthermore, compaction also involves updating the references to live objects from old location to the new one. This means that during the reference update operation, for a live object, there could be some references pointing to the old copy, and the rest to the new copy. This could also potentially cause the two copies to be inconsistent. To deal with the problem of concurrent compaction, there are following three ways:

Stop-the-world compaction. The simplest solution to the problem of compaction is to defer it as much as possible, and do it when required in a stop-the-world fashion. This completely eliminates the issue of inconsistency among dual copies of the live objects, as described in the previous paragraph.

[2] While some concurrent GCs stop all the application threads for very short time for collecting entry points into object-graph, there exist so called *on-the-fly* GCs which never stop all the application threads at the same time.

To defer the need of compaction as much as possible, the collector keeps track of all unallocated holes in the heap [35]. This list of unallocated holes gets updated after every concurrent marking phase is over. This ensures that the need for compaction only arises when an allocation request cannot be met within the available free holes. At this point, all the applicaiton threads are stopped and compaction performed to defragment the heap. This stop-the-world compaction brings back the issue of long pauses during the compaction. To deal with which, Detlefs et al. [12] propose *Garbage First*, which although stops the world for compaction, but logically partitions the heap into fixed-size regions, which can be then independently compacted. The regions with maximum garbage are chosen first for compaction, that is why the name ”Garbage First”.

Fragmented allocation. The need for compaction can be completely eliminated if objects are allocated in fixed-size fragments. This approach is used in *Schism* garbage collector by Pizlo et al. [37]. Objects larger than the fragment size span multiple fragments which may not be contiguous. This means that different fields of an object may require different number of address derefences to reach the fields. However, since this garbage collector can be successfully used in a real-time systems as eliminating need of compaction ensures that worst-case execution time can be predicted accurately, as demonstrated in Schism.

Concurrent compaction. The two techniques of compaction discussed above are basically not concurrent. Although they are important components of garbage collectors which are concurrent as a whole, but the objects are not moved, for the purpose of compaction, while the mutators are also running. Sapphire was the first concurrent copying collector [24]. It logically partitions the healp into two semi-spaces, so that during collection, live objects can be replicated from one space to the other. At the end of a collection cycle, all the memory in the old space is reclaimed. While the object is being replicated, the consistency of two copies of an object are ensured using write barriers. However, Sapphire does not support simultaneous updates by several application threads to a single object, while it is being replicated. It requires the threads to block on memory writes to an object that is being copied. Furthermore, it effectively uses only half of the given heap, because the other half is reserved for collection purposed.

Compressor, another concurrent compaction algorithm, exploits hardware traps for performing compaction concurrently [26]. It performs compaction on page-by-page basis. This way it does not have to reserve more than a page for compaction purpose. Furthermore, to ensure concurrent access by application threads, it creates another *to-virtual heap* with the reserved page as its first page, and without any read/write permission on it. Then, it stops the world to quickly update the roots so that they start pointing to new location of their referents in the *to-virtual heap*. This ensures that now application threads can only access *to-virtual heap*, and in absence read/write permission would get system trap. This trap is serviced by moving the corresponding page from old space to *to-virtual heap* and then enabling that page's read/write permission. The only disadvantage of Compressor is that it initially, right after beginning the compaction activity, the application threads suffer from a trap storm yielding very low processor utilization for few milliseconds.

Another concurrent compaction algorithm, called STOPLESS, targets real-time applications, uses a concept of *wide* objects while replicating objects [36]. A wide object represents an intermediate representation of an object being copied. In other words, when an object is being copied, there are three, instead of two copies of it, the third being the wide object. The wide objects are the same as the actual object being copied, but it appends every field with a status field. Objects are copied field-by-field where every field is updated using a double-word compare-and-swap instruction so that the status field appended to each field can also be updated along with the field.

So far all the concurrent garbage collectors that we have discussed offer a single space concurrent collection. Even if the the heap is laid out in generational manner, only the full heap collections are concurrent, not the intra-generational onces. The arguement in favour of this approach is that younger generations are small and also create garbage more often and hence can be collected using some stop the world garbage collector with short pauses. However, since full heap collections can be really longe, they are done using concurrent garbage collector to avoid long pauses. On the other hand, Tene et al. argue that on multi-gigabyte heaps, even young generation collections cannot be done in stop-the-world fashion as the large generation size leads to intorlerably long pauses [48]. Therefore, they propose C4, a *Continuously Concurrent Compacting Collector*, which supports simultaneous-generational concurrent collections. It means that different generations in a generational heap setup are simultanously collected, each using a concurrent garbage collector. C4 uses read barriers and hardware traps to collect young and old generation simultaneously and concurrently, without compromising on data consistency. The read barriers are used to update references after compaction, and hardware traps (by protecting memory access to heap regions), like in the case of Compressor, are used to trap application threads while accessing a live object which has not been copied yet.

While concurrent garbage collectors ensure short response time, they incur extra cost of synchronization with application threads using read/write barriers. Latest processors have started supporting *Hardware Transactional Memory*. In [38], Ritson et al. explore use of HTMs for different synchronization needs of garbage collectors. They explored three different garbage collection scenarios: parallel copying, concurrent copying, and bitmap marking. For parallel copying, HTM was used in place of using a compare-and-swap to establish a forwarding pointer in the old objects' header word. For concurrent copying, Sapphire collector was used to experiment this scenario [24]. Sapphire's collector thread uses compare-and-swap to update every field of objects in the to-space. HTM replaced these compare-and-swap operations. Similarly, in parallel bitmap marking, collector threads use compare-and-swap operations to update bits in the bitmap. HTM, in this context too, was used to replace the expensive compare-and-swap operations. The authors identified that HTM is most applicable to concurrent collectors as the other two scenarios did not provide any considerable improvement. The concurrent copying could gain substantially because it provided sufficient work to amortize the cost of transaction setup.

2.4.2.5 Thread-local heaps

Another popular technique to improve concurrency in garbage collected languages is the use of thread-local heap [1, 15, 29, 39, 47]. In this technique, every application threads has a its own private heap. All allocations from a thread are done in its private heap. There is a shared heap too, which contains all such objects which are shared among all the application threads. The main idea is that as long as all objects in a thread-local heap are only accessible to its associated thread, the thread-local heap can be collected independently of the rest of the heap, which contains other thread-local heaps and the shared heap. To maintain this invariant, write barriers are used. Whenever a reference field in an object belonging either to the shared heap or any thread-local heap i is updated to a referent in a thread-local heap j, where $i \neq j$, then the referent object and entire sub-graph reachable from it are moved to the shared heap. This allows a thread to collect its private heap while the other application threads continue to execute on their own private heaps and the shared heap. For the shared heap, any of the garbage collection techniques discussed above can be utilized.

Papers	Technique Used			Description
	Marking	Compaction	NUMA-aware	
Flood et al. [16]	STW	STW	N/A	Generational parallel GC. Early version of Parallel Scavenge
CMS and G1 [12, 35]	Co: WB	STW	N/A	Compaction in small regions to avoid long pauses
Schism [37]	N/A	Fragmented Allocation	N/A	Eliminates need for compaction
Sapphire [24]	Co: WB	Co: WB	N/A	Exploits Java memory model, does not support non-blocking shared modification
Compressor [26]	N/A	Co: MT	N/A	Region-by-region compaction using memory protection
STOPLESS [36]	Co: WB	Co: Wide obj	N/A	Uses Double-word CAS to move objects field-by-field
C4 [48]	Co: RB	Co: MT	N/A	Generational GC providing concurrent collection in all generation simultaneously
Ritson et al. [38]	Co: WB	Co: HTM	N/A	Evaluates use of HTM for concurrency control during various GC operations
Doligez and Leroy [15]	Co: TLH	Co: TLH	N/A	Seggregates objects into shared and thread-private ones to provide concurrenct garbage collection
Ogasawara [34]	STW	STW	Biased Lock	Moves objects to the node where they are likely to be accessed by mutator next
Tikir and Hollingsworth [49]	STW	STW	H/W counter	Uses h/w counters to decide where to move objects to improve application locality
Zhou and Demsky [52]	STW	STW	Yes	Targets only locality and not balance

Table 2.2 – Summary of research papers on various Garbage Collection Techniques (**STW**: Stop-The-World, **Co**: Concurrent, **WB**: Write Barrier, **RB**: Read Barrier, **MT**: Memory Traps, **HTM**: Hardware Transactional Memory, **TLH**: Thread-Local Heap).

2.4.2.6 NUMA-awareness in Garbage Collectors

There is not much research done on incorporating NUMA-awareness in garbage collection. Ogasawara [34] and Tikir and Hollingsworth [49] proposes to migrate objects to the node where the thread which accesses that object most of the time is scheduled to run. This migration is done during garbage collection. For this, Ogasawara uses *dominant thread* information stored in objects' header. An object which is locked or reserved using a biased lock [13] by a thread is migrated, along with the whole sub-graph originating from that object, to the node where the thread was last dispatched before GC. Tikir and Hollingsworth take a different approach, in which they use hardware performance counters to determine the *most-accessed-from* node of every object during the application execution. And then this information is used during GC to migrate objects. Both of these research work intend to use GC as a tool to improve object locality for application threads on NUMA architectures. Whereas, in NumaGiC we intend to improve the GC's performance on NUMA architectures. Furthermore, in NumaGiC we ensure that both object access locality as well balance are maintained, which is not the case with the above described two approaches as they only target locality, while it has been shown by inventors of Carrefour that balance is much more important than locality [11].

The work that is closest to NumaGiC is by Zhou and Demsky [52]. The authors designed a parallel NUMA-aware garbage collector on their proprietary Java Virtual Machine, targeting the TILE-Gx microprocessor family, which allows software to deactivate cache-coherency on demand. They use this feature and disable cache-coherency during garbage collections (not for the applications for backward compatibility). In the collector, each core collects its own local memory, and sends messages when it finds remote references. Zhou and Demsky focus on the locality of the application and the GC. They do not consider balancing memory access among the nodes, but as discussed above, Carrefour [11] show that memory access balance has a dramatic impact. They also do not consider reducing inter-node references, 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.

Table 2.2 summarises all the research work that has been discussed in this section.

Chapter 3

Evaluation of Existing Garbage Collectors on NUMA architectures

Parallel garbage collectors came into existence when multiprocessors were simpler in design. However contemporary hardware is much more complex. Therefore, it is time to re-evaluate their scalability. Hence, we conducted experiments to evaluate (1) GCs effect on application scalability, and (2) GC scalability. This evaluation was conducted on AMD48 machine in Hotspot Java virtual machine of OpenJDK7, described in chapter 2, section 2.2.3.

Hotspot provides three parallel/concurrent GCs: (1) The default GC of HotSpot, Parallel Scavenge (please refer to chapter 2, section 2.1), (2) a generational concurrent mark-sweep GC (ConcMS), consisting of a parallel copying young generation and a concurrent mark-sweep old generation [35]; and (3) Garbage First (G1) [12], which is a concurrent, incremental, parallel garbage collector.

3.1 Selection of applications and garbage collector

For this evaluation we wanted to use applications/benchmarks which cover diverse memory usage/access patterns. While it is critical to have include memory-intensive applications for this excercise, inclusion of cpu-intensive small memory footprint applications is also vital to cover the entire spectrum. Therefore, our set of applications/benchmarks include:

1. Spark and Neo4j, two memory-intensive real applications,
2. SPECjbb2005, a SPEC benchmark with large heap-size requirements, and
3. applications from DaCapo benchmark suite to cover smaller heap-size cases

Along with their diverse memory requirements, considering that our evaluation involves scalability comparisions, it is important for the chosen applications/benchmarks to have good application scalability. Furthermore, performing scalability study of all the applications of DaCapo benchmark suite and the other three large heap-size applications on all the three garbage collectors of HotSpot for the entire research work would take enormous amount of time. Therefore, we performed a throughput

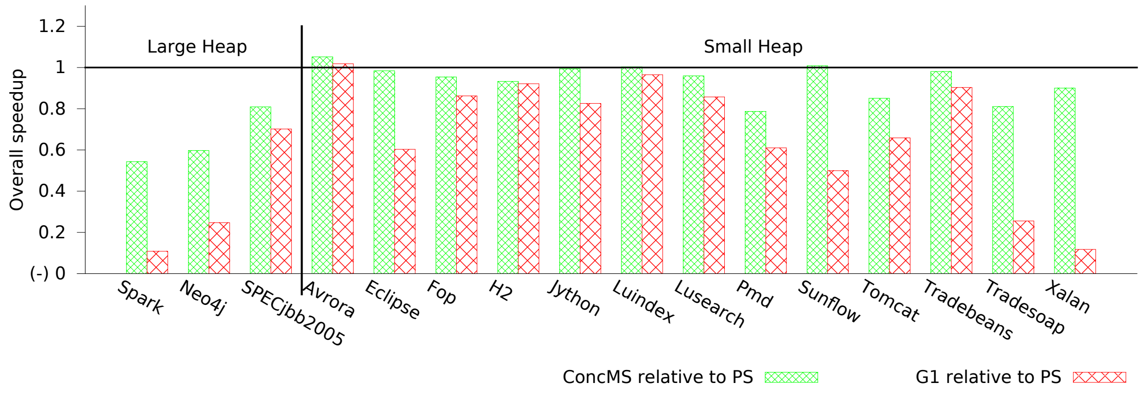


Figure 3.1 – Evaluation of various applications with different heap size requirements with the three GCs of HotSpot. Higher is better

experiment on all the DaCapo benchmark applications, and the other three applications using all the three GCs of HotSpot. Then we chose the best performing garbage collector, and three most scalable DaCapo applications, in addition to the three large heap-size applications, for all other evaluations in our research.

Figure 3.1 shows the throughput of all the Dacapo applications and Spark, Neo4j, and SPECjbb2005. All the DaCapo application are run using 100MB heap size, except for H2, Tradebeans, Tradesoap, and Eclipse, which are run using 500MB heap size. SPECjbb2005 was run using 6GB heap size, Neo4j with 32GB, and Spark with 130GB heap. The figures are relative to Parallel Scavenge. As figure 3.1 shows, among all the applications, only Avrora runs approximately 5% faster with ConcMS, and G1 as compared to Parallel Scavenge. On all the other applications, Parallel Scavenge performs either better, or comparable to ConcMS and G1. Particularly, on applications which has large heap sizes, ConcMS and G1 are substantially slower as compared to Parallel Scavenge.

Therefore, considering this result we decided to use Parallel Scavenge as the base garbage collector for this research work. Among all the applications, six applications are chosen for the scalability analysis of this section (and all the scalability studies in the following chapters). These six applications include the three larger heap size applications i.e. SPECjbb2005, Neo4j, and Spark. Furthermore, among all the DaCapo benchmark applications, considering the result of figure 3.2, the three most scalable applications are utilized, i.e. Sunflow, Lusearch, and Tomcat. Figure 3.2 shows the comparison of speedup, i.e., the execution time with one thread compared to the execution time with n threads, of all Dacapo benchmarks. This experiment was performed with a (relatively) very huge initial heap size (26GB) to avoid triggering GC as much as possible. Furthermore, in order to emulate a hardware with same number of cores as application threads in the experiment, we set CPU-affinity for each run such that all the threads are packed in minimum number of required nodes. The experiments were repeated three times and the average execution time is reported.

3.2 Scalability analysis

In this section we analyse the scalability of the six applications using Parallel Scavenge. We intend to analyze:

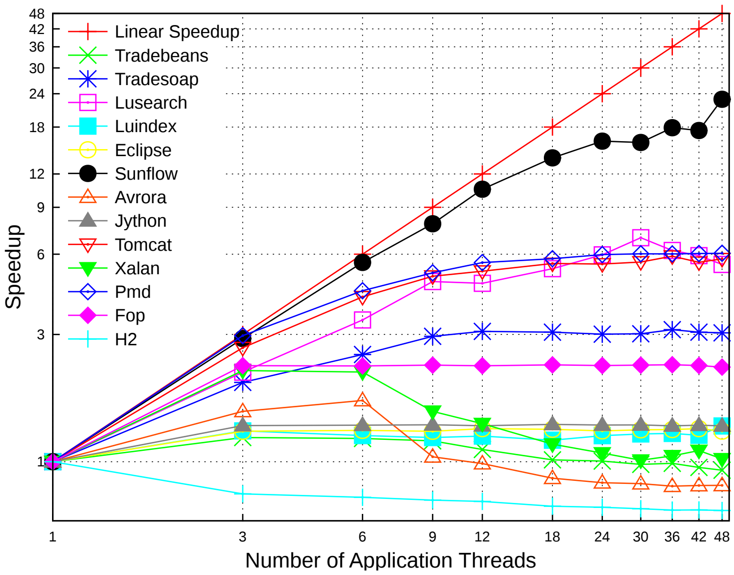


Figure 3.2 – Scalability comparison of DaCapo benchmarks with large heap size.

1. Scalability of application time/throughput with varying number of cores, and
2. Scalability of GC throughput with varying number of NUMA nodes.

3.2.1 Application scalability

This sub-section analyses the application scalability. We vary the number of cores used to run the application. The cores are activated such that the minimum required nodes are used. This is done to emulate a hardware with the specified number of cores. Figure 3.3 shows the completion time of five applications: Spark, Neo4j, Sunflow, Lusearch, and Tomcat [1]. The completion time consists of pause time, which is the time spent in performing garbage collection, and application time, which is the time taken by application threads.

Figure 3.3 shows that in case of Lusearch, Sunflow, and Tomcat the pause time starts increasing after a certain number of cores. The increase in pause time compensates any improvement in application time due to better application scalability, obstructing the overall scalability. In the case of Spark, the pause time does not increase with increasing number of cores. On the other hand, the application time continues to decrease. This trend leads to continuous increase in the proportion of time spent in garbage collection, thereby increasing the dominance of garbage collector's scalability in the overall scalability. At 48 cores, the proportion of pause time in total completion time is above 50%. This reflects the importance of garbage collector's scalability. Furthermore, it reflects that, had the garbage collector be scalable, in which case we must have observed a decreasing amount of pause time, the overall scalability would have been even better. Finally, in the case of Neo4j, neither the pause time increases, nor the pause time's proportion in total completion time. This reflects that the application itself lacks good scalability. However, since the proportion of pause time is large, a scalable garbage

[1] SPECjbb2005 could not be included in figure 3.3 as it is a fixed-time benchmark. Completion and pause time remain constant irrespective of the number of cores used.

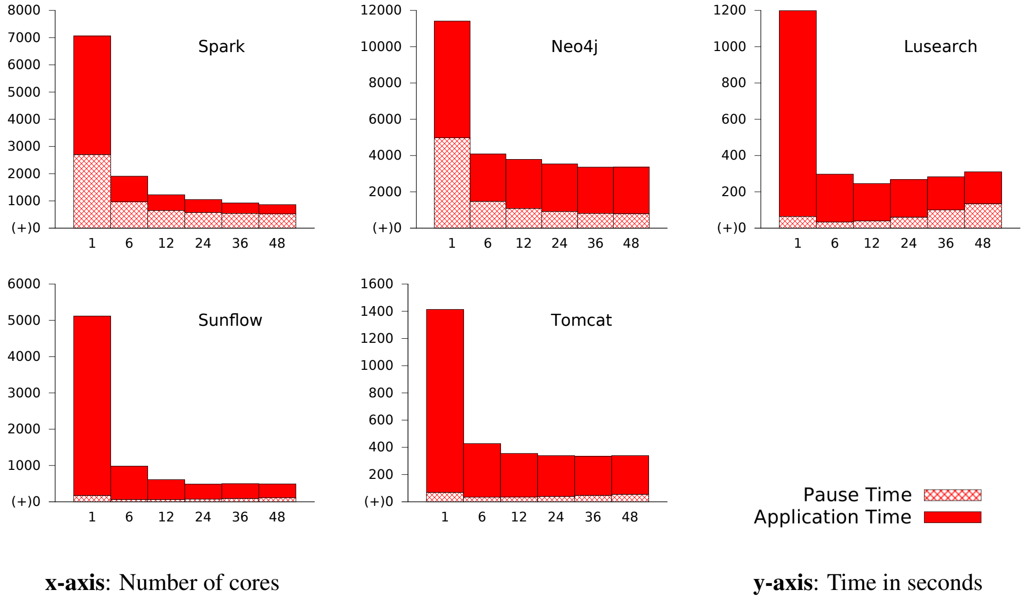


Figure 3.3 – Total completion time of the application and the time spent in GC.

collector could have positively affected the overall scalability, which is not happening with Parallel Scavenge.

It is clear from the above result that garbage collector's scalability plays a very significant role in the overall scalability. To analyze the application scalability from a different perspective, figure 3.4 shows the completion time/application throughput curve of all the six applications (including SPECjbb2005) with increasing number of cores and compares it with the linear improvement over the 1-core performance of Parallel Scavenge. It is clear from the graphs that all the six applications are far from optimal completion time (Application throughput in case of SPECjbb2005). Certainly, it is impossible for some applications to attain the optimal completion time due to their inherent suboptimal scalability. However, there is definitely some room for improvement in application scalability by improving the garbage collector's scalability.

3.2.2 GC scalability

Figure 3.3 clearly indicates that the garbage collector's scalability can play a vital role in the overall scalability of the applications. The fact that pause time's proportion in total completion time increases with number of cores indicates that memory access pattern does get affected by the NUMA archicture. Please recall that the cores are enabled such that minimum number of nodes are active to include the active cores. This means that the NUMA effects become increasingly prominent with increasing core count and hence start showing its impact.

To further analyse this effect, we study GC scalability in terms of NUMA nodes as well. This scalability study is conducted with the help of GC throughput. GC throughput is computed as the amount of live data processed during garbage collection per unit of time. GC scalability is analysed with increasing number of NUMA node. The NUMA nodes are activated in steps, starting with one active NUMA node. Therefore, the application threads as well as the GC threads are restricted to the active nodes.

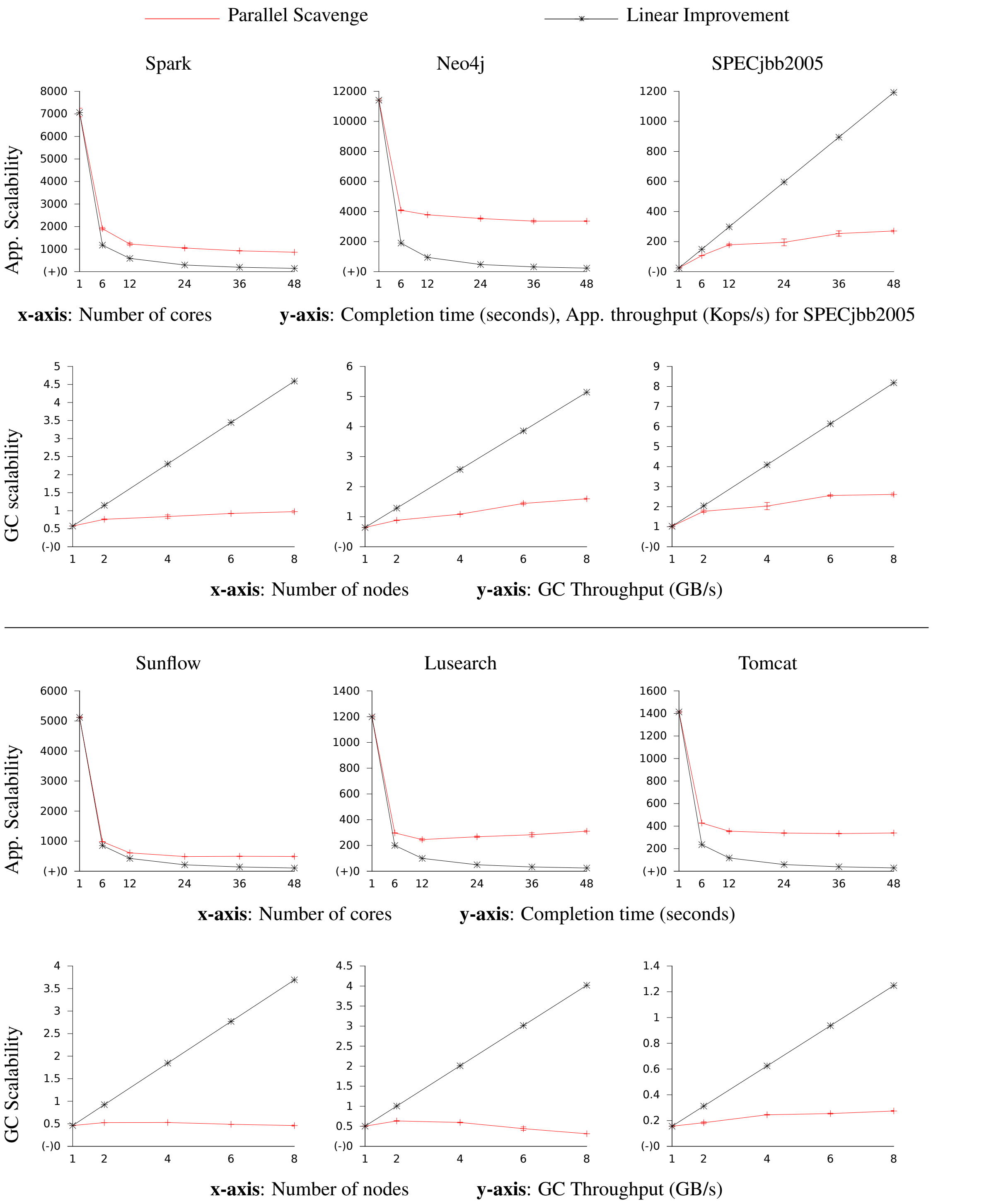


Figure 3.4 – Application and GC scalability for the six applications. (+) = lower is better, (-) = higher is better.

Figure 3.4 shows the GC scalability curves of the six applications with increasing number of NUMA nodes. As shown in the figure, the GC throughput shows poor scalability. Lusearch and Sunflow show negative scalability after a certain number of NUMA node is reached. In case of the other four applications, the GC throughput continues to improve with every increase in NUMA nodes. However, it is substantially far from linear speedup. Furthermore, the fact that negative effects of NUMA architecture, does not hide away the negative effects of shared access contention on synchronisation primitives. In fact, in case of small heap-size applications Lusearch, Tomcat, and Sunflow, there is a possibility that the poor scalability is not due to NUMA-oblivous memory access, instead is due to contention on some synchronisation primitive used inside the garbage collector. We will go into the details of these possibilites in the following chapters.

3.3 Conclusion

In this chapter, we presented some preliminary results, obtained by experimenting GCs implemented in OpenJDK7 on a 48-core NUMA machine. We chose the best performing garbage collector among the three provided by Hotspot and conducted scalability analysis. The results suggest that GC acts as a bottleneck for applications and does not scale. Rather than pause time decreasing, we observed an increase as the number of cores increases. This establishes a genuine requirement to re-look at the design of existing garbage collectors, which were designed before the NUMA architecture came into existence, so that they can scale better on contemporary hardware.

Chapter 4

Synchronisation Primitives in Parallel Scavenge

In this chapter the mechanism for shared access synchronisation in Parallel Scavenge is discussed. The performance issues related to synchronisation that are experienced by Parallel Scavenge on large multicores are described in detail. Furthermore, the solution that we implement is also explained. Finally we present an experimental evaluation where we compare GC and application scalability of this improved Parallel Scavenge with the baseline one.

4.1 Background

Chapter 2 described in detail the heap layout and algorithm of Parallel Scavenge. In this section thread management of Parallel Scavenge is discussed. Parallel Scavenge consists of several phases. Figure 4.1 shows all the phases involved in young collection. First, mutators are paused at the stop-the-world barrier. During the initialisation phase, a single thread, called the VM thread, prepares GC tasks, which are executed in the parallel phase by the GC threads. A *Termination Protocol* detects that all GC threads have terminated the parallel phase; after which a barrier suspends the GC threads and wakes up the VM thread. In the final phase, the VM thread mainly resizes the heap and wakes up the mutators.

The same sequence of phases as shown in figure 4.1 is used during marking and compaction steps of the old collection (please refer to chapter 2). Figure 4.2 shows the entire sequence of phases when a full garbage collection is performed. All the phases inside the *Stop-the-World pause* of figure 4.1 are repeated twice in sequence, once for the marking step, and then for the compaction step. The *final phase* of marking step, and *init phase* of compaction are merged into one phase as both of them are performed by VM thread. Furthermore, the summary phase is also executed during this merged phase.

4.1.1 Synchronisation mechanism

As synchronisation between threads is a common performance bottleneck on a multicore machine [28], we describe how it is implemented and used in Parallel Scavenge. Parallel Scavenge uses mon-

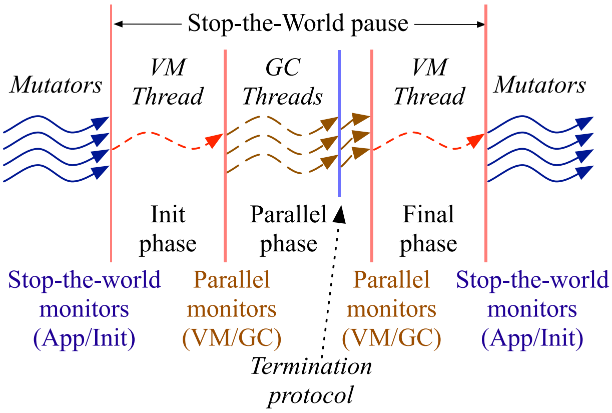


Figure 4.1 – Phases of Young Collection in Parallel Scavenge

itors for this purpose. A monitor contains both a condition variable, to suspend and wake up threads, and a lock to prevent concurrent accesses to shared data. Furthermore, since the condition variable of a monitor is a shared variable, it is protected by the associated lock.

Parallel Scavenge uses two monitor pairs: a *stop-the-world pair* and a *parallel pair*. The *stop-the-world pair* suspends the mutators and wakes up the VM thread at the beginning of the collection, and vice-versa at the end. It consists of an App monitor, to synchronise the mutators, and an Init monitor, to synchronise the VM thread. The *parallel pair* suspends the VM thread and wakes up the GC threads at the beginning of the parallel phase, and vice-versa at the end. It consists of a VM monitor to synchronise the VM thread; and a GC monitor to synchronise the GC threads, and to protect data shared between the GC threads.

4.1.2 Initialisation phase

Being a stop-the-world collector, Parallel Scavenge has to ensure that all mutators are paused at the beginning of a collection. The stop-the-world monitor pair coupled with a counter to know the number of suspended mutators serves this purpose. As soon as all the mutators are suspended, the VM thread initialises the queue of GC tasks by adding all the GC tasks (refer to chapter 2) which can be executed in parallel by the GC threads in the parallel phase. Then, the VM thread synchronises with the GC threads: it wakes up all the GC threads that are waiting on the GC monitor, and suspends itself by waiting on the VM monitor.

4.1.3 Parallel Phase

In this phase, all the GC threads parallely execute the GC tasks added by the VM thread to the GC-task queue in the initialisation phase. In order to dequeue a GC task for the shared GC-task queue, GC threads need to synchornize, for which they use the lock inside the GC monitor.

It is critical that all the GC-tasks are executed completely, before moving to the next phase. Therefore, a barrier is required at the end of the parallel phase which ensures that all the GC threads reach that barrier before moving forward. In Parallel Scavenge, this barrier mechanism is implemented inside *termination protocol*.

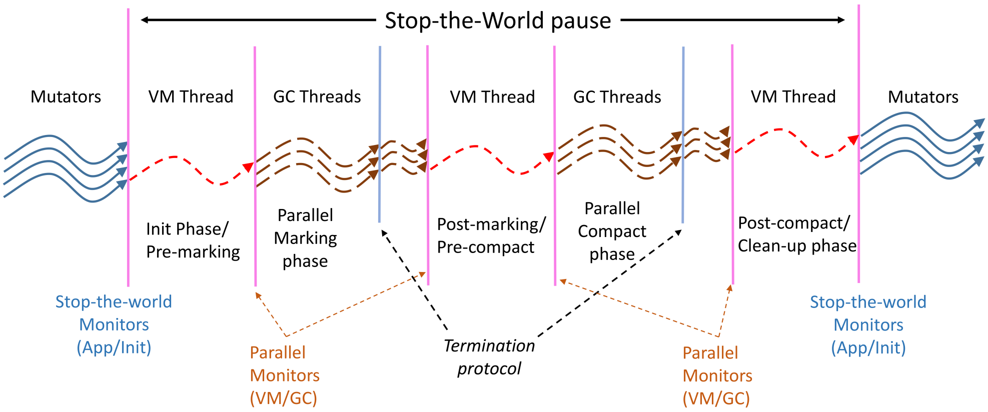


Figure 4.2 – Phases of Full Collection in Parallel Scavenge

4.1.3.1 Termination protocol

One of the similarities between young collection, marking, and compaction is that all of them make use of *steal tasks*. These tasks have the responsibility of balancing the workload among all the GC threads. The steal tasks also implement a termination detection protocol (a barrier in other words). This termination protocol allows every GC thread to offer termination after a configures number of steal attempts fail to fetch any work for the GC thread. When all the GC threads offer to terminate, it indicates that the work is done, and hence all the threads have reached the barrier. Otherwise, after waiting for other GC threads to join it in offering termination for a fixed amount of time, it checks sequentially if any GC thread has work to do. If found one, the GC thread retracts its offer, and tries to steal again.

4.1.3.2 Final task

A special task is added at the very end of the GC-task queue, called the *final task*. This task is suppose to be picked up only after ensuring that the operations performed in the parallel phase are safely finished. The final task aims to ensure that no GC thread is modifying the heap when the VM thread restarts. Once a GC thread has left the termination protocol, it again attempts to pop a task from the task queue. Since there is a single final task, only one GC thread succeeds. It thereby becomes the leader, and coordinates with other GC threads using the parallel monitors (VM and GC). We call it the final thread.

The other GC threads find the task queue empty. They increment a global thread counter protected by the GC monitor. Then, they wake up the final thread and suspend themselves on the same monitor. Conversely, the final thread waits on the GC monitor, until the global thread counter reaches the number of GC threads. Once this is done, the final thread wakes up the VM thread using the VM monitor. This constitutes entry into the final phase, where the VM thread is the only one running.

Note that, in the normal case studied here, this final synchronisation is redundant with the termination protocol because both ensure that all the GC threads have terminated their steal tasks. It is required only in specific configurations (out of scope here) where there are no steal tasks.

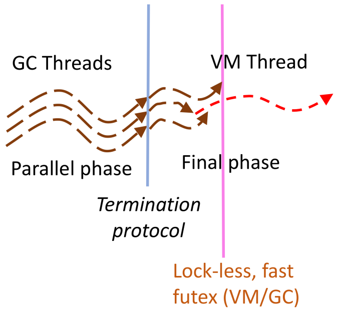


Figure 4.3 – Lazy GC thread parking after the parallel phases in Parallel Scavenge

4.1.4 Final synchronisation phase

The main purpose of the final phase varies between young collection, marking step, and the compaction step. In case of young collection, this phase adapts the sizes of the spaces and the young generation. The sophisticated resizing policy of Parallel Scavenge is essential for performance, because it adapts the heap size to the needs of the application, based on factors such as space usage and/or time taken by the collection cycle. In Parallel Scavenge, resizing is cheap, as it consists of simply adjusting a set of pointers.

In case of marking step, the final synchronisation phase gets merged with the init phase of compaction step. This merged phase performs summary step, compaction of permanent generation, and updating the task queue of GC tasks with GC tasks of compaction step.

In case of compaction step, the VM thread serially performs some post-compact operations in the final phase. It updates the references of the objects which overlap more than one region (refer to chapter 2). After that, it clears the mark bit-maps and summary data array so that these data structures can be reused in the next full colleciton. Finally, it (re)sets the entire card table, depending on whether the young generation contains live data after compaction or not.

Once the post parallel phase operations are done, the VM thread, in case of young collection and compaction step, resumes the mutators using the stop-the-world monitor pair. It wakes up the mutators with the App monitor, and then sleeps, awaiting the next collection with the Init monitor.

4.2 Synchronisation optimisations

Parallel Scavenge uses locks to synchronise access to internal shared data structures, such as the GC-task queue and the condition variables. To implement lock acquisition, Parallel Scavenge first attempts a fast-path atomic compare-and-swap (CAS) instruction, spinning for some number of iterations. If this fails, the mutator falls back to a slow-path using Posix synchronisation primitives. CAS works fine for a small number of threads, as the fast path generally succeeds. However, as observed by Lozi et al. [28], on a large multicore, lock performance collapses under contention; this is particularly severe when spinning on CAS. To avoid this issue, we study the code to find the most contended locks, which we fix as explained next.

4.2.1 Lock-free GC task queue

Issue. Parallel Scavenge synchronises access to the GC-task queue by using the GC monitor's lock. At the beginning of the parallel phase, all the GC threads access the task queue at the same time. The lock becomes contended, and its performance degrades drastically. Many GC threads wait for a long duration, preventing them from participating in the parallel phase, sometimes for the whole duration of a collection.

Solution. The task queue has First-In-First-Out semantics. In Parallel Scavenge, it is implemented as a singly-linked list. To avoid the performance collapse caused by lock acquisition, we implement the task queue with a Michael-Scott lock-free queue [31]. Recall that the VM thread enqueues tasks during the initialisation phase, which are dequeued during the parallel phase. Thus, there is no concurrency between enqueue and dequeue operations, but only between dequeues. Therefore, the VM thread enqueues without synchronisation, and GC threads dequeue using atomic CAS operations.

4.2.2 Lazy GC parking

Issue. When a GC thread executes the final task, all GC threads request the GC monitor's lock in order to synchronise the end of the parallel phase. However, this lock request immediately follows the termination protocol, and therefore all the GC threads reach this point at roughly the same time. As a consequence, the lock gets contended and its performance collapses.

Solution. To avoid this issue, we remove the GC monitor's lock. The lock was used for three purposes: first, to protect the task queue; second, to protect the global thread counter which is used for the barrier at the end of the parallel phase; and third, to protect the associated condition variable of the GC monitor, which is used to suspend the GC threads.

The first use is already taken care by the lock-free task queue. For the second case, we remove the redundant synchronisation in the final task (see Section 4.1.3.2). Instead of waiting for the other GC threads, the final thread simply wakes up the VM thread, and then suspends itself. Other GC threads suspend themselves without any synchronisation. After this change, the global thread counter is not required anymore.

For the last case, we replace the condition variables of VM and GC monitors with Linux's `futex_wait` calls [17]. A `futex_wait` has the semantics of an atomic compare-and-sleep, and does not require acquiring a lock. Figure 4.3 depicts our solution.

However, our modifications potentially introduce a new race condition if a GC thread gets preempted after the termination protocol, but before suspending itself. In this scenario, the VM thread may wake up the mutator threads, a new collection may begin and the VM thread may wake up the GC threads to begin the new parallel phase. If the delayed GC thread suspends itself on the futex at this step, it will never be woken up by the VM thread, leading to a deadlock. This issue is depicted in figure 4.4.

This was not a problem in Parallel Scavenge because, during the whole execution of the final task, the GC threads own the lock of the GC monitor. Acquiring this lock is required to wake up the GC threads with the GC monitor to begin the new parallel phase. Therefore, the GC threads inevitably suspend themselves before receiving the wake up notification. However, this condition does not hold with our optimisations as the GC monitor does not exist anymore.

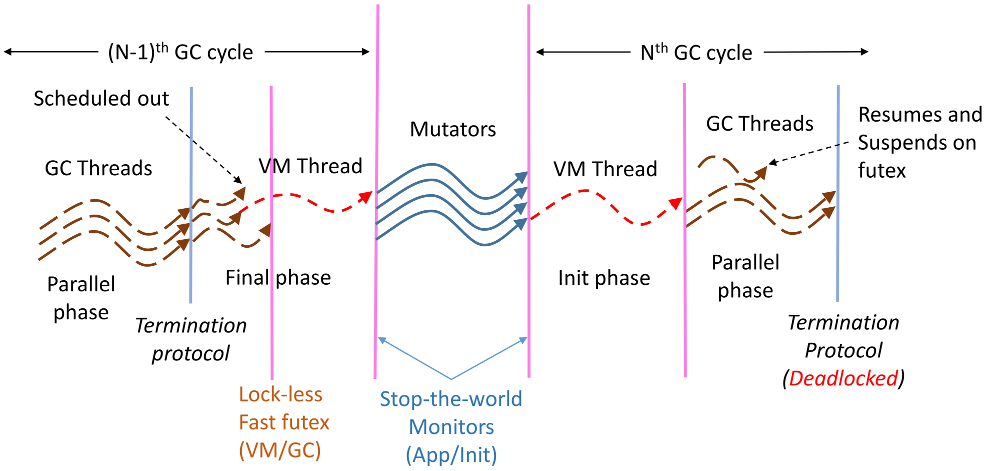


Figure 4.4 – Deadlock arising out of lazy GC thread parking

To solve this problem, we use a timestamp, which is incremented atomically by the VM thread before the parallel phase. To suspend itself, a GC thread uses the futex to atomically check whether the timestamp has been modified before sleeping. If it was not modified, the futex call suspends the GC thread. Otherwise, it returns, and the GC thread directly enters the new parallel phase, thus avoiding the deadlock.

4.3 Post-compact optimisations

The three post-compact operations: updating references of overlapping objects, clearing data structures, and (re)setting card table, can very easily be parallelised. Since all these operations are performed on list-type data structures, we simply partition these data structures into multiple slices, one per GC thread, and then deploy all the available GC threads to work on these slices in parallel.

4.4 Evaluation

Taking forward the scalability evaluation introduced in section 3.2 of chapter 3, in this section we compare the implications of the improvements explained in this chapter with the baseline Parallel Scavenge GC. Figure 4.5 shows the result of application scalability, and GC scalability compared in terms of NUMA nodes. For simplicity, the modified ParallelScavenge GC which contains the improvements explained in this chapter is called SynchroPS. In terms of GC scalability, SynchroPS scales better than Parallel Scavenge for all the applications, except SPECjbb2005. However, the scalability of SynchroPS is substantially less than linear scale.

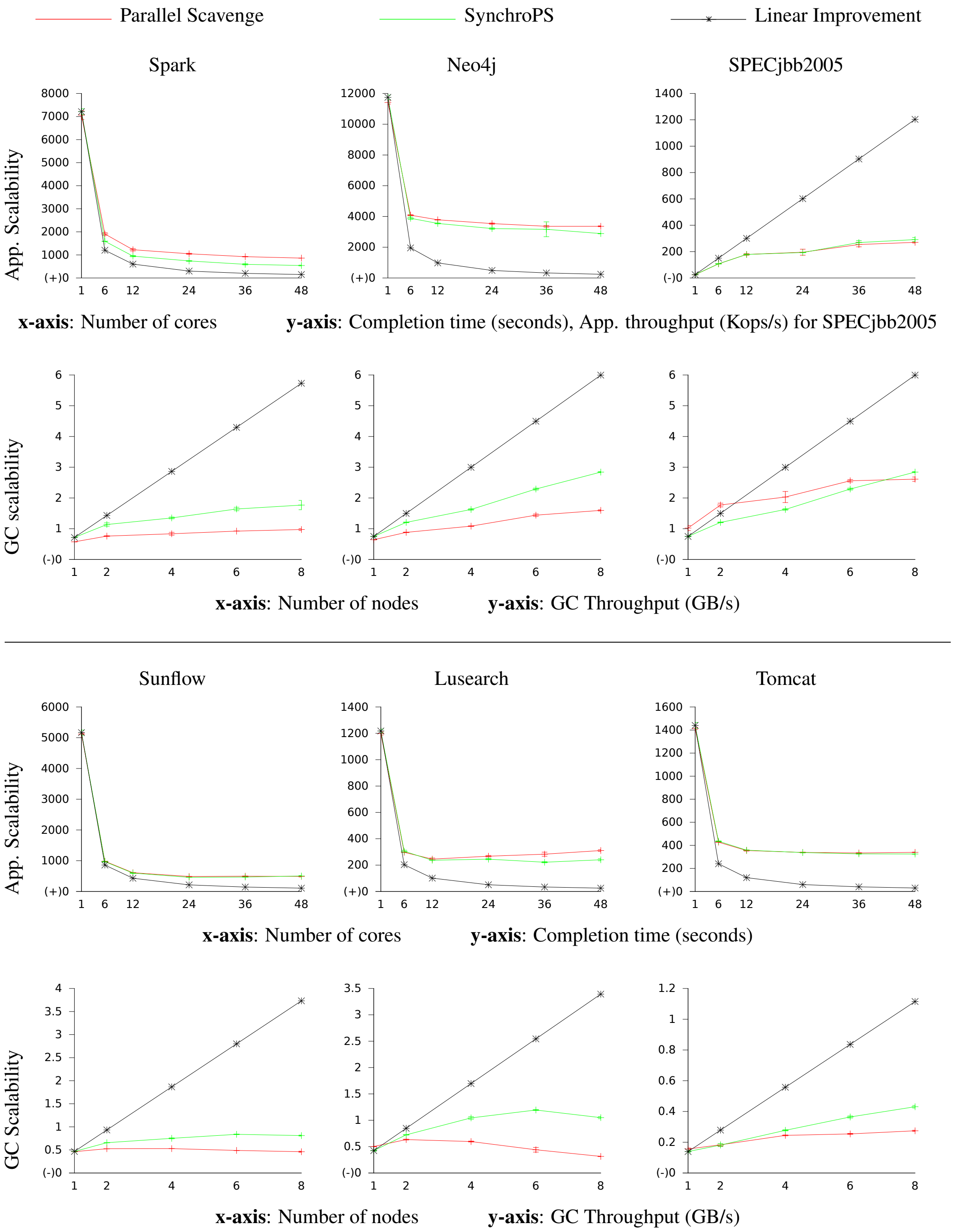


Figure 4.5 – Application and GC scalability for the six applications. (+) = lower is better, (-) = higher is better.

The case of SPECjbb2005 is special because of the memory allocation pattern of this application. SPECjbb2005 has an initilization phase which is executed by a single thread. This initilization phase allocates enough objects to consume almost the entire eden space. Considering this scenario, and the fact that kernel allocates pages using first-touch policy wherein pages are taken from the same NUMA node where the faulting thread is executing. This results in a major portion of the eden space being mapped on a single NUMA node. Furthermore, the generational design of GC ensures that the same eden space mapping is reused after every young collection as all the live objects are moved to either the survivor space or old generation. This leads to increasing pressure on the node's memory controller with increasing number of GC threads, eventually saturating the memory controller after a certain number of threads are reached.

This allocation pattern should affect both Parallel Scavenge and SynchroPS equally. Then why does SynchroPS unexpectedly perform worse than Parallel Scavenge? The contention on mutex lock (please refer to subsection 4.2.1) plays a positive role in overcoming the memory controller saturation problem for Parallel Scavenge GC. As the contention on the mutex lock obstructs the GC threads to perform GC work (and hence memory access) all at the same time, the pressure on the memory controller gets reduced proportionately, leading to better GC throughput even with lessar parallelism than what SynchroPS provides. This phenomenon shows the importance of memory access balance.

For Application scalability, the quantum of its improvement, as shown in figure 4.5, in case of all the six applications depends on the proportion of pause time in the completion time of that application, and the improvement in GC scalability. Therefore, Spark and Lusearch has benefited the most as these two applicaitons have the highest proportion of pause time in completion time; Neo4j, Sunflow, and Tomcat benefit from better GC scalability, but not as much as Spark and Lusearch. Finally, in case of SPECjbb2005, the difference in application scalability of Parallel Scavenge and SynchroPS is negligible.

4.5 Conclusion

In this chapter we explained all the standard multi-core optimizations, which are not related to NUMA architectures, that we implemented in Paralellel Scavenge. We also compared the scalability of parallel Scavenge and its improved version, that we call SynchroPS. The results show that, except for SPECjbb2005, the result for the other five applications, as expected, was better with SynchroPS than Parallel Scavenge. However, the scale of GC improvement with SynchroPS is substantially less than the linear curve. As the parallel phase of SynchroPS is entirely lock-free, we do not think that SynchroPS can be further optimized with new synchronization optimizations. Thus, we suspect that the lack of scalability of the GC is caused by another phenomenon: the lack of NUMA awareness.

Chapter 5

NumaGiC: a Garbage Collector for NUMA architectures

This chapter analyses memory access pattern of Parallel Scavenge garbage collector using representative applications to identify NUMA issues. Using this analysis, we describe the design of our proposed garbage collector NumaGiC, a mostly-distributed GC. Finally, we evaluate NumaGiC using a wide range of real applications and benchmarks by comparing it with different variants of Parallel Scavenge.

5.1 Interleaved Parallel Scavenge

The original Parallel Scavenge, and the improved version SynchroPS (refer chapter 4) suffers from memory access imbalance on NUMA machines, which drastically degrades its performance [20]. Since it is a rather simple fix, and to ensure a fair comparison, we created a modified version of SynchroPS, which we shall call InterPS. InterPS 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.

To evaluate the effect of memory interleaving, we use the same scalability experiments as used in previous chapters (refer to chapters 3 and 4). Figure 5.1 shows very interesting results. Out of the six applications that we experimented, while three applications, namely Sunflow, Lusearch, and Tomcat showed no improvement over SynchroPS; Spark and Neo4j show negative performance by using InterPS as compared to SynchroPS; only SPECjbb2005 gained substantial performance improvement by interleaving the memory across NUMA nodes.

Sunflow, Tomcat, and Lusearch did not gain from memory interleaving due to their very small working set size. The three applications do not create enough memory pressure to show any visible impact of better memory balance due to memory interleaving.

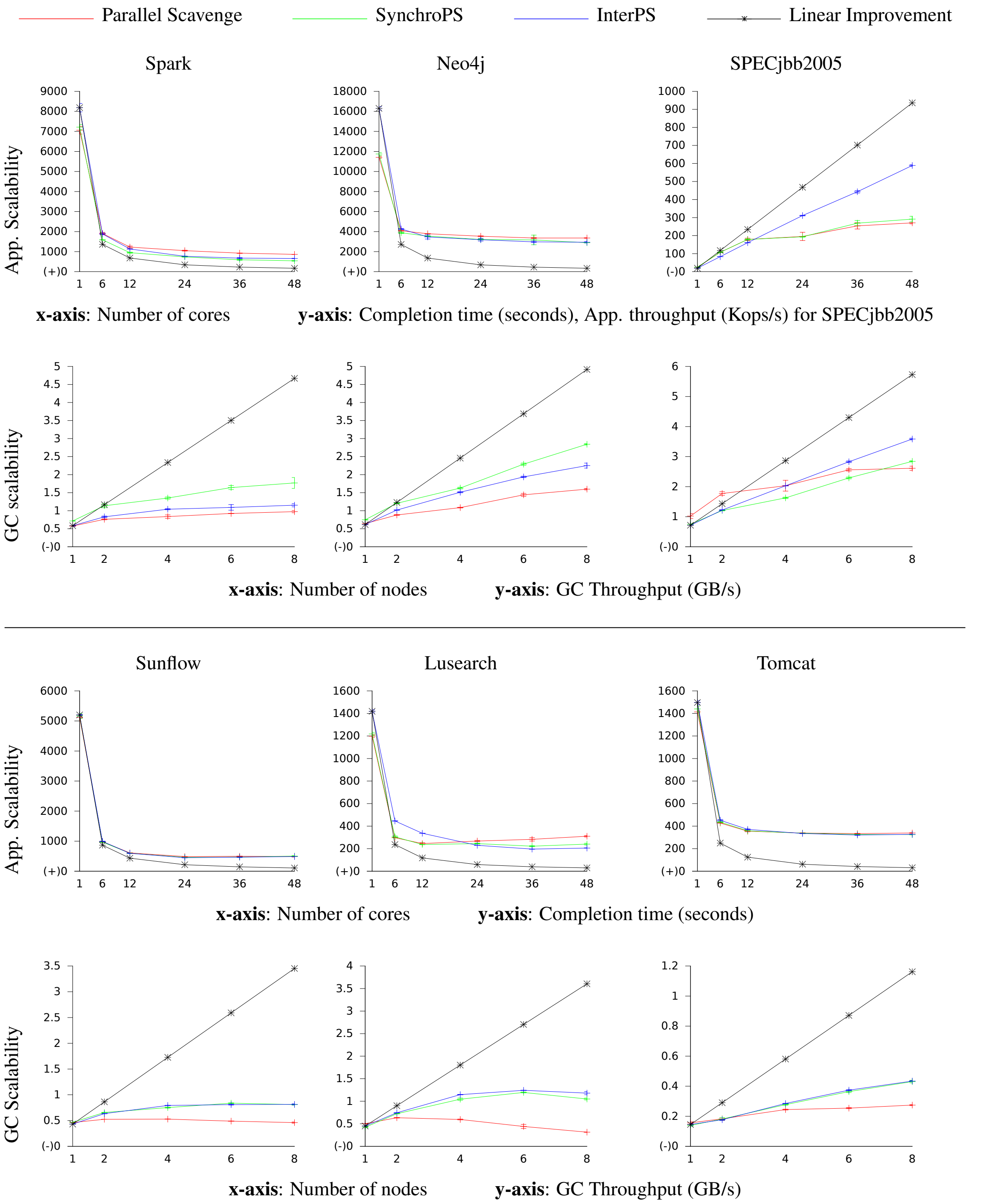


Figure 5.1 – Application and GC scalability for the six applications. (+) = lower is better, (-) = higher is better

In case of Spark and Neo4j, we saw a negative impact of memory interleaving. Both these applications are very memory intensive and therefore any variation in the memory access pattern would undoubtedly have an visible impact on the performance. However, as we see in figure 5.1, enforcing a perfect memory balance had a negative impact. Although memory interleaving provides perfect memory access balance, but it comes at a cost of very poor memory access locality. In fact, because of the round-robin interleaving, on average, out of every n accesses by a thread, $n - 1$ accesses are necessarily remote. These two applications possibly have a reasonably good memory access balance, as well as some natural access locality. However, interleaving the heap completely destroyed access locality to provide perfect memory access balance. Therefore, the marginal improvement in memory access balance, and complete destruction of access locality had such an adverse impact on these two applications that it degraded these applications' performance as compared to SynchroPS.

SPECjbb2005, on the other hand, substantially gained because of memory interleaving. This happens because with SynchroPS and Parallel Scavenge, the application gets majority of physical memory corresponding to the young generation mapped onto a single NUMA node. This happens due to a single-threaded initilisation phase of SPECjbb2015. This behavior saturates the memory controller on the particular node and hence impacting the performance. Memory interleaving of InterPS fixes this problem. This shows that memory interleaving is better than accessing memory only from few NUMA nodes (as with SPECjbb2005), even though interleaving destroys memory access locality (as with Spark and Neo4j). This experiment clearly shows that just using an interleaved space is not a satisfactory solution to better exploit NUMA topology, and that a better memory placement is required to avoid remote memory accesses.

5.2 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 and as shown in case of SPECjbb2005 (see figure 5.1), *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 [45], 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.

5.2.1 Object graph analysis

In order to improve the spatial NUMA locality and the memory allocation balance, we first analyse the object graphs of five applications, chosen for their diverse allocation patterns:

	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 5.1 – Analysis of the object graph

1. Spark, the multi-threaded map-reduce engine [42];

2. Neo4j, the embedded, disk-based, fully transanctional Java persistence engine that manages graph data [32];

3. SPECjbb2013, the business logic service-side application benchmark that defines groups of threads with different behaviours [44];

4. SPECjbb2005, another business logic service-side application benchmark where all the threads have the same behaviour [43]; and

5. H2, an in-memory database from the DaCapo 9.12 benchmark [5, 22].

Please note that Sunflow, Lusearch, and Tomcat are not included in this analysis. This is because, as seen in figure 5.1, these three applications have memory footprints that are to small to be effected by any memory access pattern modifications.

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 InterPS, which ensures that the objects allocated by a mutator thread running on node i always stay on node i.[1] In this experiment, as in all the experiments presented in the thesis, we let the Linux scheduler place the mutator threads on the cores. Table 5.1 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 randomly chosen and representative 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. Indeed, our experiment reports a proportion between 77% and 100% between young objects, between 16% and 99% from old to young objects and between 27% and 55% for all the references,

[1] This version of InterPS is exactly the pure distributed algorithm described in Section 5.4.5.

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.

5.2.2 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 which 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. This requires the garbage collector to move live objects while copying/compacting to the same node where they are originally allocated. This, in addition to the node-local clustering observation, will ensure that the garbage collector does not create any spurious inter-node references.

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 2.1, 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.

5.2.3 Implementation details

In order to be able to map virtual addresses to nodes, we make use of *fragmented spaces*. 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 5.1), a fragmented space allows a GC thread to retrieve the segment and NUMA location of an object from its virtual address. The GC thread can also place an object onto a specific NUMA node by using a segment mapped to the desired node. We implement a variant of InterPS which, for young generation, uses fragmented spaces to ensure node-local object allocation. We call this garbage collector NAPS (Node-local Allocating Parallel Scavenge). NAPS still uses an interleaved space for old generation as in the case of InterPS. NAPS implements the Node-Local Allocation policy fully, while Node-Local Copy policy is partially supported: copy of young objects from eden space to survivor space is node-local; however, due to an interleaved old generation, object promotions from young to old generation are not node-local.

In NumaGiC, we further 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. Besides 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.

5.3 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.

5.3.1 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[2] 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 5.2.3).

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

Moreover, when a GC thread idles in local mode, it may steal work from the pending queues (described in section 2.1) 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 [23].

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 5.4.4 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.

5.3.2 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 [9]. 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 2.1) 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.

5.3.3 Switching between local and work-stealing modes

A GC thread enters work-stealing mode when it 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 two-fold. 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 subgraph, thanks to spatial NUMA locality created by the heuristics. Second, re-entering local mode and

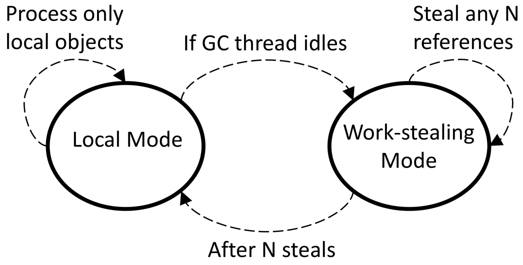


Figure 5.2 – State transition diagram depicting switching between local and work-stealing modes.

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.

Figure 5.2 shows the state transition between the two modes explained above. 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.

5.4 Evaluation of NumaGiC

This section studies the performance of NumaGiC, using both standard benchmarks (SPECjbb2013, SPECjbb2005, SPECjvm2008, and Dacapo) and widely-used, industrially-relevant applications (Spark, and Neo4j). Although most of the experimental setup is already described in chapter 2 section 2.3, for a more complete evaluation, we have added an additional NUMA machine, and two benchmarks. These are described in the following sub-section. The sub-section will also desribe the heap-size and workloads used in all the applications. Following that, we study the impact of the policies presented in Section 5.2, 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.

5.4.1 Hardware

In addition to the machine described in chapter 2, called *Amd48* hereafter, a second machine, called *Intel80* hereafter, is also used for evaluation. Intel80 is 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 48 GC threads on Amd48 and 80 GC threads on Intel80.

5.4.2 Applications and Benchmarks

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

Spark 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 InterPS (Interleaved Parallel Scavenge), 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 5.4.3. 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 experiments are run on all the cores of the machine.

Neo4j 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 [43, 44] that model supermarket companies. SPECjbb2013 models all the components of the company, using a larger number of threads with different behaviours, interacting together.

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.

5.4.3 Evaluation of the policies

This section presents an evaluation of the placement policies discussed in Section 5.2.2. For this analysis, we focus on Spark with a 32 GB heap size running on the AMD Magny-Cours with 48 cores and 8 nodes. The results are similar for the other evaluated applications and on both machines.

Figure 5.3 reports the following metrics from the experiment:

	Experiment	Heap layout		Node-Local			
	Name	Young	Old	Allocation	Copy	Compact	Root
▩	InterPS	Interleaved	Interleaved	No	No	No	No
▧	InterPS+Numa	Fragmented	Fragmented	Yes	Yes	Yes	Yes
■	InterPS+Compact	Interleaved	Fragmented	No	Promotion*	Yes	No
▨	InterPS+CompRoot	Interleaved	Fragmented	No	Promotion*	Yes	Yes
▤	NAPS	Fragmented	Interleaved	Yes	Intra-young gen$^\delta$	No	No

*: Only during promotion to the old generation.

$^\delta$: Only copies from eden to survivor space during young collection.

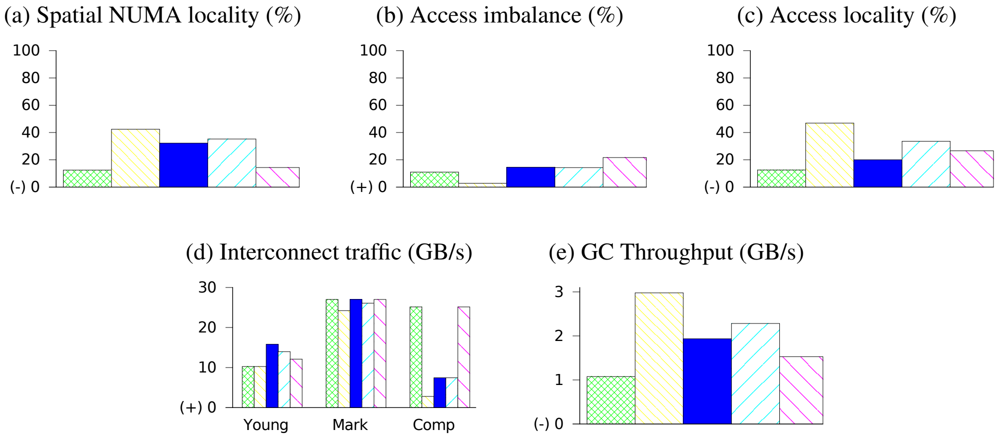


Figure 5.3 – Evaluation of the NUMA-friendly placement policies. (+) = lower is better, (-) = higher is better.

- *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 5.2.1), 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 InterPS (see Section 5.1). As indicated in the table of the figure, InterPS+Numa extends InterPS with fragmented spaces and the four NUMA-friendly placement policies described in Section 5.2.2. We observe that the policies improves spatial NUMA locality, as the InterPS+Numa algorithm increases the fraction of local references from 12% to 42% (see Figure 5.3.a). We also observe that memory access balance is slightly better with InterPS+Numa than with InterPS, with a standard deviation of 2.8%

vs. 11%, which is already low (see Figure 5.3.b).

In the rest of this section, the variants InterPS+Compact, InterPS+CompRoot and NAPS enable a finer-grain comparison. InterPS+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. InterPS+CompRoot is the same as InterPS+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 5.3.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 InterPS to 32% with InterPS+Compact (see Figure 5.3.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 InterPS+CompRoot with InterPS+Compact, we observe that the Node-Local Root policy alone does not have a significant effect on spatial NUMA locality. In InterPS+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 InterPS+Compact.

Comparing InterPS 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 5.2.2).

Comparing InterPS+Compact with InterPS+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×.

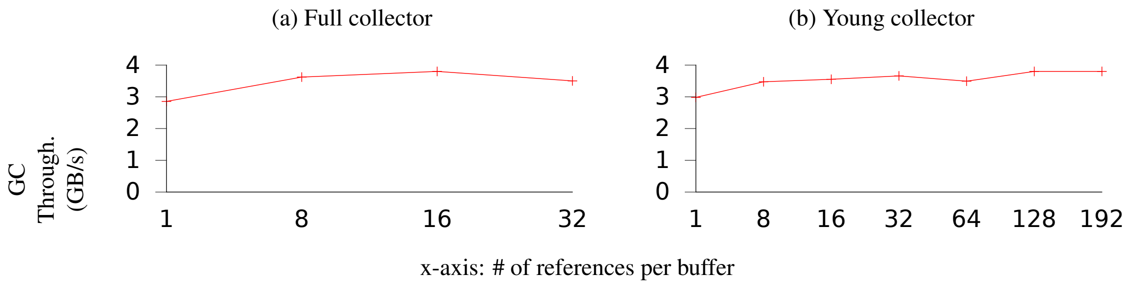


Figure 5.4 – GC throughput of NumaGiC with varying transmit buffer sizes (higher is better).

GC throughput improvements It is interesting to observe that our NUMA-friendly placement policies have the side-effect of improving GC throughput (Figure 5.3.e). This is because they also improve memory access locality (Figure 5.3.c).

Node-Local Compaction improves memory access locality, from 13% with InterPS to 20% with InterPS+Compact. As explained in Section 5.2.3, a GC thread first processes the regions of its node before stealing, which avoids many remote access during the compacting phase (Figure 5.3.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 5.3.d).

Node-Local Root alone also improves memory access locality substantially, with 33% local access in InterPS+CompRoot, compared to 20% for InterPS+Compact (Figure 5.3.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 InterPS+CompRoot, to 47% with InterPS+Numa (Figure 5.3.c).

Overall, the improved memory access locality translates to better GC throughput, reaching 3.0 GB/s with InterPS+Numa, compared to 1.0 GB/s for InterPS (Figure 5.3.e).

5.4.4 Impact of transmit buffer size

One of the internal design parameters of NumaGiC is transmit buffer size (see Section 5.3.1); we study its impact on performance. For this experiment, we run Spark with the small workload on Amd48. Figure 5.4 reports the GC throughput of NumaGiC, i.e., the number of gigabytes of live data collected per second, with varying transmit buffer sizes. Figure 5.4.a varies the buffer size used in the full collector, and sets a fixed size of 128 references for the young collector. Figure 5.4.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.

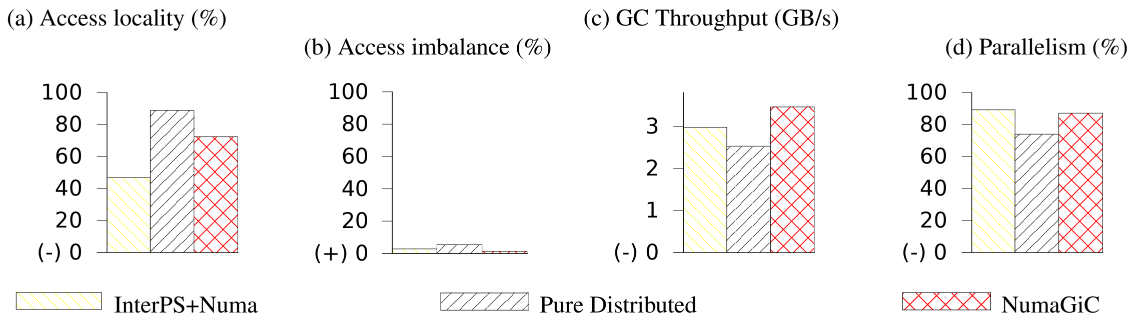


Figure 5.5 – Memory access locality versus parallelism. (-) = higher is better, (+) = lower is better

5.4.5 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: InterPS+Numa, which adds all the NUMA-friendly placement policies of Section 5.2.2 to Parallel Scavenge Baseline (InterPS), Pure Distributed, a version of NumaGiC in which a GC thread always stay in local mode, and full NumaGiC.

Figure 5.5 reports three metrics defined in Section 5.4.3, 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 termination protocol, *i.e.*, where it synchronises to terminate the parallel phase (see section 2.1).

We can observe that memory access balance remains good with Pure Distributed (Figure 5.5.b). However, this observation is not significant. For example, memory allocation, and thus access, is imbalanced with H2, as shown in Table 5.1. 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 InterPS+Numa (Figure 5.5.a), throughput decreases, from 3.0 GB/s down to 2.5 GB/s (Figure 5.5.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 InterPS+Numa (Figure 5.5.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.

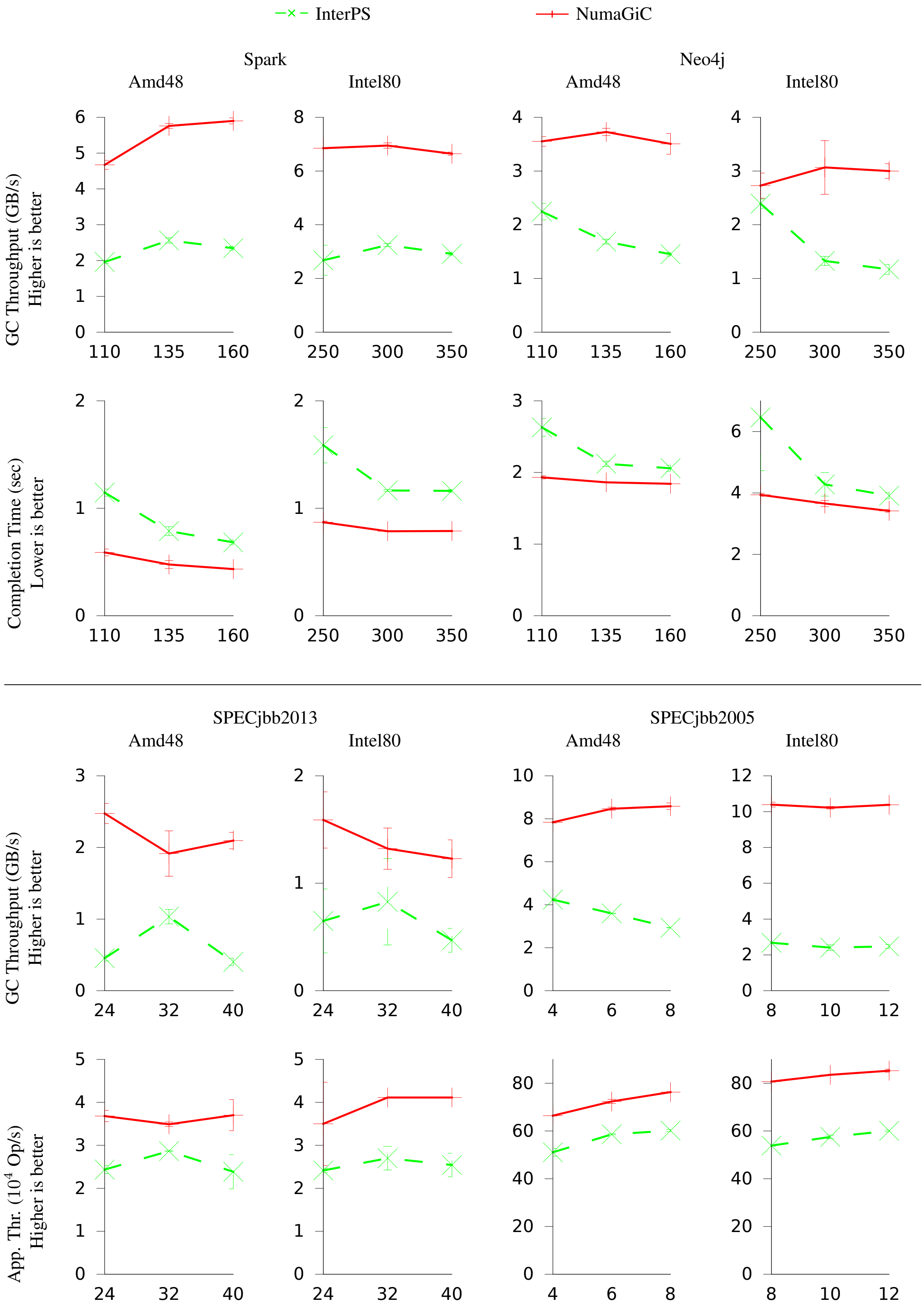


Figure 5.6 – Evaluation of the applications with large heap. **x-axis**: Heap size in GB.

Observe that NumaGiC corrects this problem by providing the best of the InterPS+Numa and the Pure Distributed algorithms. The parallelism of NumaGiC remains at the level of InterPS+Numa (11% of idle time for InterPS+Numa against 13% for NumaGiC, see Figure 5.5.d). At the same time, NumaGiC degrades memory access locality slightly, compared to Pure Distributed (from 88% to 72% of local accesses, see Figure 5.5.a), but it remains largely improved over InterPS+Numa (47% of local accesses). Thanks to improved memory access locality GC throughput increases from 3 GB/s in InterPS+Numa, to 3.5 GB/s in NumaGiC (Figure 5.5.c).

5.4.6 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 5.6 reports GC throughput and overall speedup of Spark , Neo4j, SPECjbb2005 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 InterPS 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 InterPS, up to 5.4× on Amd48 and up to 4.2× on Intel80, which translates into an overall improvement (application + GC) up to 94% on Amd48 and up to 82% 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 57% on Amd48 and between 14% and 62% on Intel80.

Figure 5.7 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 InterPS, NumaGiC improves the overall performance of 19 applications by more than 5% (up to 143% for Scimark.sparse.large), does not change it by more than 5% for 13 of them, and degrades performance by more than 5% in a single application (Xalan, degraded by 8%). For this application, GC throughput actually improves, but, as we let Hotspot use its default resizing policy, this modifies the heap resizing behaviour, causing the number of collections to double. In summary, even for small heaps, more than half of the applications see significant overall improvement thanks to the the better memory access locality of NumaGiC, and with the sole exception of Xalan, NumaGiC never degrades performance relative to InterPS.

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 more than half 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.

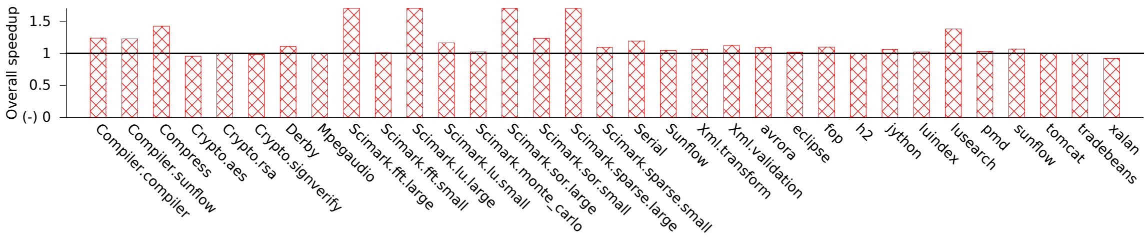


Figure 5.7 – Evaluation of applications with small heap (DaCapo and SPECjvm2008). Higher is better.

5.4.7 Scalability

Recalling from section 3.2 of chapter 3, we identified different ways in which garbage collection time affects the five applications that we have been using to analyse GC performance in this thesis. For Sunflow, Lusearch, and Tomcat we identified increase in pause time with increasing number of cores as a concern. For Spark we observed that although the application scales, the pause time stays constant with increasing number of cores. This eventually leads to a situation where GC consumes more than 50% of the total completion time. Finally, for Neo4j, we showed that both the pause time and the completion time become stagnant with increasing number of cores. All these cases reflect that garbage collectors scalability is crucial in all cases.

In this regard, we conducted the same experiment with NumaGiC as the GC. Figure 5.8 shows the results of the experiments. As shown, in case of Sunflow, Lusearch, and Tomcat, the pause time does not increase with increasing number of cores anymore. Unfortunately, due to very small working set size, the GC does not have enough work to utilize all the cores that a contemporary hardware provides. After a certain number of cores, the cost of synchronisation among all the GC threads overtakes the amount of GC work that each thread can perform during a collection cycle. Therefore, with these three applications the NumaGiC fails to scale.

In case of Spark and Neo4j, NumaGiC continues to reduce pause time with increasing number of cores. Furthermore, it ensures that the proportion of time spent in garbage collection out of total completion time does not grow with increasing number of cores. The experiment results shown in figure 5.8 confirm that NumaGiC ensures that GC does not become a hurdle in the scalability of an applications' overall performance.

Next we study the scalability of all the garbage collectors discussed in this thesis. For all the six applications, we study the application scalability, GC scalability with increasing number of NUMA nodes.

As shown in the figure 5.9, NumaGiC scales with increasing number of NUMA nodes substantially better than all the other three GCs. NumaGiC scales almost linearly in case of SPECjbb2005, slightly less in case of Spark, and further less in case of Neo4j. This variation of scalability is directly related to the proportion of clustered objects of each application, as shown in Table 5.1. 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.

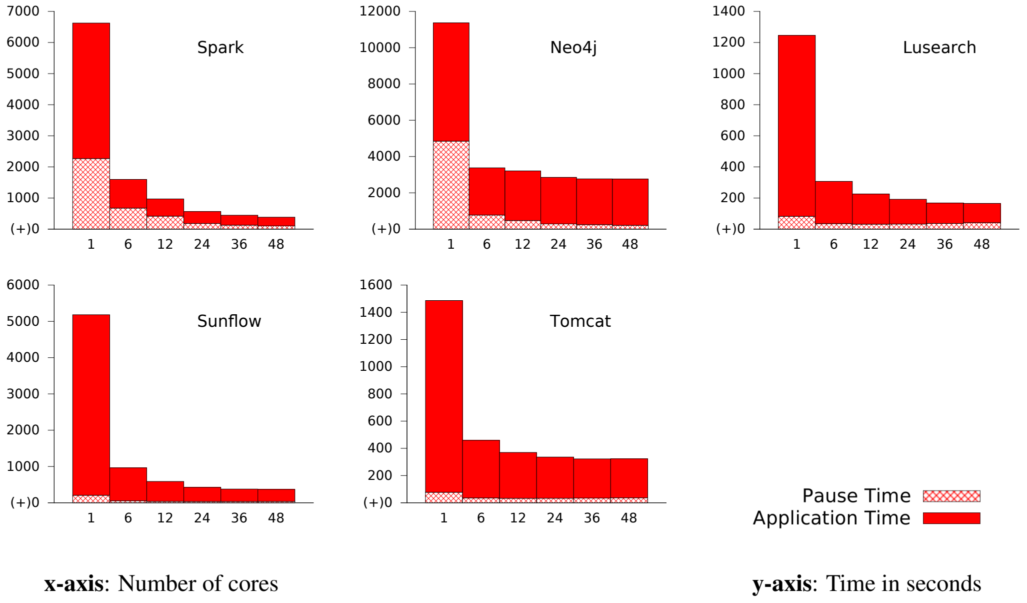


Figure 5.8 – Total completion time of the application and the time spent in GC.

In case of less memory intensive applications, namely, Sunflow, Lusearch, and Tomcat, NumaGiC always scales better than all the other GCs with increasing number of NUMA nodes. However, the scalability is much poorer than in the case of memory intensive applications discussed in the previous paragraph. This is due to smaller working set size of these applications. The size of live data that is to be processed by GC threads during every collection cycle is much smaller in case of Sunflow, Lusearch, and Tomcat and hence the time spent by GC threads in doing real work eventually becomes substantially smaller, while the burden of synchronisation among GC threads, at best, stays constant. To give an idea, in case of Tomcat, the total amount of live data processed during the entire application run is apprximately 17GB, whereas in case of Spark, it is approximately 433GB.

Finally, in the application scalability curves of the six applications also, NumaGiC lets the application perform better than all the other GCs. Although this experiment has more influence of applications' own scalability rather than GCs', still the purpose of these curves was to demonstrate that NumaGiC aids the applications in attaining the best possible performance among all the GCs that we have studied in this thesis.

5.5 Conclusion

In this chapter, 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 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 94% over InterPS on the two ccNUMA machines for applications with large heaps.

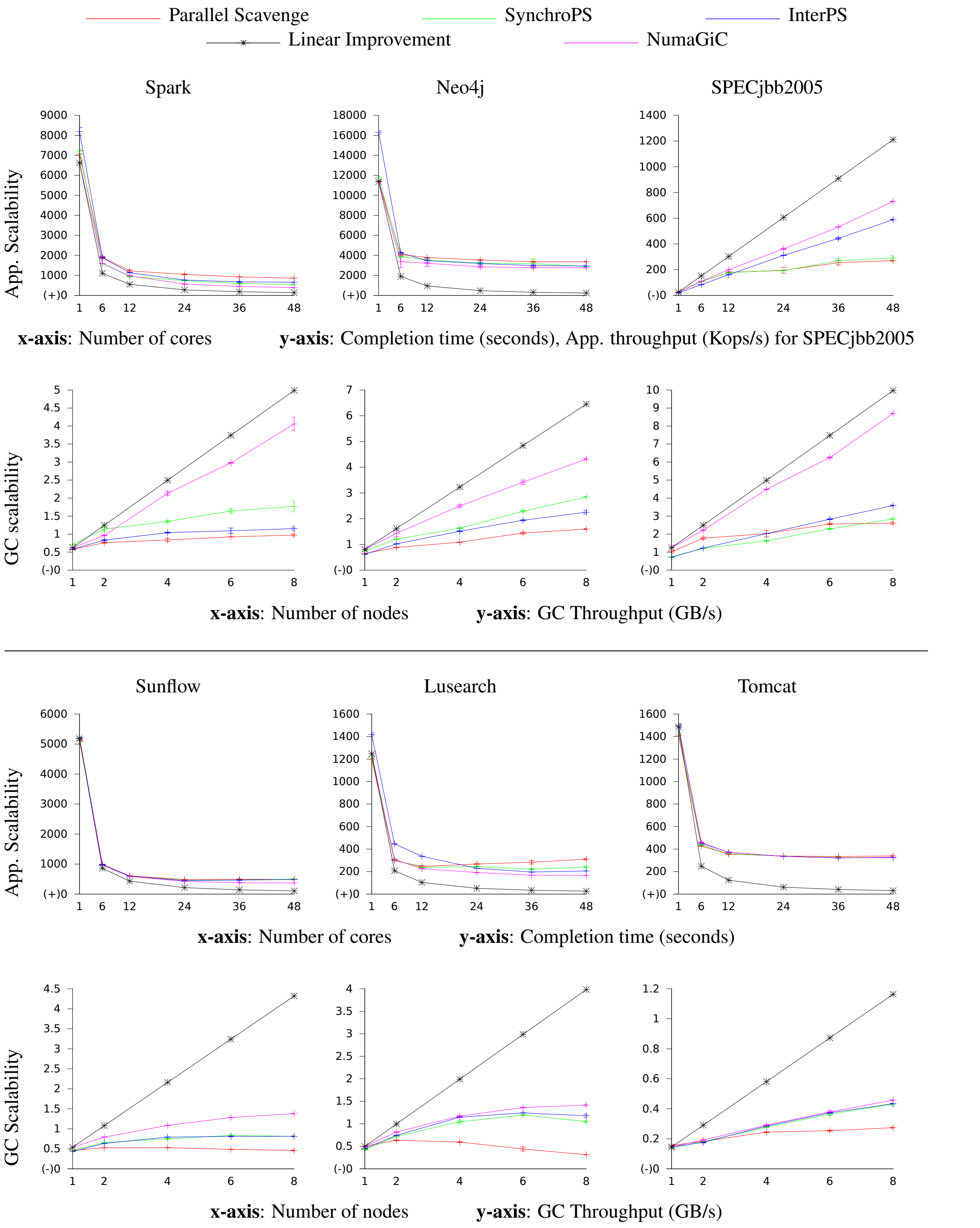


Figure 5.9 – Application and GC scalability for the six applications. (+) = lower is better, (-) = higher is better.

Chapter 6

Conclusion and Future Work

This chapter concludes this dissertation. It also discusses future work direction which is not only an obvious extension to our work, but also a necessary one.

6.1 Conclusion

In this research work, a study of effects of NUMA architecture on garbage collectors has been conducted. In a preliminary evaluation of existing garbage collectors on a NUMA machine revealed that the existing garbage collectors fail to scale with increasing number of NUMA nodes. In fact, in some cases they scale negatively. Furthermore, garbage collectors tend to consume a non-negligible amount of total completion time of applications. This makes the problem of poor garbage collector scalability a vital issue to be resolved.

On further analysis, two issues were identified as the root cause for the poor scalability of garbage collectors. Firstly, high contention while synchronizing access to GC-internal shared data structures was observed, specially in case of applications which are less memory intensive. This problem was solved using standard parallel-programming techniques by replacing lock-based data structures with their lock-free counterparts.

The second problem, and rather more complex one, is of NUMA-obliviousness in the way garbage collectors process live objects. We identified that the applications naturally create clusters of objects with few inter-cluster references. However, due to NUMA-obliviousness, Parallel Scavenge destroys this naturally created clusters when it moves objects for promotion/compaction.

To solve the issue, we proposed NumaGiC, a mostly-distributed garbage collector. NumaGiC strives to esnure that object-clusters which are created by the application threads are not destroyed. In fact, it uses load-balancing as an opportunity to enhance size of the clusters, further reducing inter-cluster references, and hence inter-node references.

In our evaluation, NumaGiC showed substantial improvement in GC's performance, which in turn leads to up-to 61% speedup in application performance with heap sizes that provide best performance. Furthermore, our scalability analysis show thaat NumaGiC scales substantially better than other Parallel Scavenge variants. Interestingly, NumaGiC's scalability depends on the density of object clusters. Applications with smaller clusters (high inter-node sharing, like Neo4j) leads to more messages to be sent/received and hence leaves less room for garbage collector to be scalable. On the other hand, applications with dense clusters (and hence few inter-node references, like SPECjbb2005) scale much better.

6.2 Future Work

Parallel Scavenge, a stop-the-world garbage collector, was chosen as our baseline because among the three garbage collectors provided by Hotspot, Parallel Scavenge offered the best application throughput. However, as the heap size requirements of applications are growing, the consequent increase in garbage collection pauses are becoming intolerably large. Therefore concurrent garbage collectors are fast replacing their stop-the-world counterparts. This raises certain questions:

1. Would the learnings of this dissertation of NUMA-effects on garbage collector apply to concurrent garbage collectors too?

2. If concurrent garbage collectors get adversely affected by NUMA-effects, then, considering that concurrent garbage collectors do not block application threads' progress[1], does it matter if a concurrent garbage collector does not perform efficiently on a NUMA machine?

3. If the answer to both the above questions is yes, then what needs to be done to make concurrent garbage collectors perform efficiently on NUMA architecture? Would the ideas of NumaGiC also apply to concurrent garbage collectors?

To shed some light on the importance of NUMA-awareness in concurrent garbage collectors, we discuss two issues that would arise while using a NUMA-oblivious concurrent garbage collector on a NUMA machine:

- Indeed concurrent garbage collectors do not block application threads' progress directly. However in situations where before a garbage collection cylce could finish and return back free space, if application threads run out of free space for object allocation, then they are forced to block until the garbage collection cycle finishes. With the adverse effect of NUMA-architecture, this cycle could take substantially longer to finish, and hence causing the situation described above. Furthermore, since the completion time of a GC cycle is proportional to the amount of live data to be processed, in case of memory-intensive applications using very large heaps, the situation can arise easily.

- In case of stop-the-world garbage collectors, if the garbage collector does not perform efficiently, the direct impact is only on its own performance. The application only gets indirectly affected due to longer GC pauses. On the other hand, in case of concurrent garbage collectors, since the GC threads work concurrently with application threads, they can severely affect the application's performance as well. For instance, if the GC threads create a lot of interconnect traffic, in a concurrent setup, this could slowdown application threads, even if they are doing very little inter-node access.

[1] Even if threads are blocked, the pause is not proportional to heap size (refer to chapter 2).

The above two issues, we believe, creates a strong case for designing a NUMA-aware concurrent garbage collector. The design of NumaGiC is applicable to any parallel garbage collector which moves objects. Therefore, we strong believe, that the ideas proposed in this dissertation would be very useful for building a NUMA-aware concurrent garbage collector.

Résumé de la thèse

Les applications comme les moteurs de recherches [42], les bases de données [32], les applications scientifiques, les serveurs d'applications et les serveurs web nécessitent l'utilisation de ressources matérielles conséquentes. La quantité de données et/ou le nombre de requêtes à traiter amènent ces applications à consommer tout le temps CPU, la mémoire et la bande passante disponible sur un serveur, voir sur plusieurs serveurs.

Les langages basés sur une machine virtuelle (MREs), tels que Java et C#, sont de plus en plus utilisés pour le développement de telles applications du fait de leur vitesse, leur portabilité et de la sûreté de développement qu'ils permettent. L'améliorations de techniques comme la compilation à la volée (JIT) ou le ramasse-miettes (GC) en ont fait des langages compétitifs, au même titre que d'autres langages de programmation. Cependant, comme pour tout autre logiciel, il est essentiel, pour que les applications soient efficaces, que le MRE et ses composants soient efficaces.

Un des goulots d'étranglement d'un MRE est le ramasse-miettes (GC). Un ramasse-miettes libère la mémoire allouée aux objets morts (inatteignables), de manière à ce qu'elle puisse être utilisée pour de nouvelles allocations. Pour repérer ces objets, il parcours entièrement le graphe d'objets, en partant des variables globales et de celles sur la pile, appelées *racines* dans le contexte des ramasse-miettes. Les objets inatteignables par le parcours de ce graphe le sont également pour l'application. Aussi sont-ils considérés morts et leur mémoire peut être libérée. Le ramasse-miette s'assure également de compacter le tas de manière à défragmenter la mémoire, en déplaçant tous les objets vivants vers une nouvelle partie du tas.

Le coût d'un cycle de ramasse-miettes[2] dépend de la quantité d'objets vivants traités, qui elle, dépend de la quantité de mémoire nécessaire à l'application. Puisque la plupart des applications considérées sont intensives en mémoire, le ramasse-miettes constitue une dépense de temps considérable, aussi sa performance est-elle critique. Cette thèse a pour objet la performance du ramasse-miettes pour ces applications sur les architectures matérielles actuelles.

A la différence des *Symmetric Multi-Processors* (SMP), dotées d'une architecture avec des latences d'accès uniformes entre les différents matériels, les architectures multi-coeur actuelles, appelées *cache-coherent Non-Uniform Memory Access* (cc-NUMA) ont une structure *hiérarchique*. Le premier niveau est composé de noeuds, chacun composé de plusieurs coeurs, d'un cache partagé, d'au moins un contrôlleur mémoire responsable d'un ensemble de modules mémoire, ainsi que d'un bus système qui relie tous ces composants. Au niveau supérieur se trouve un réseau qui inter-connecte ces noeuds pour permettre la communication inter-noeuds. De cette manière, le matériel masque la topologie mémoire au logiciel, de manière à assurer la rétro-compatibilité avec les applications multi-threadées qui fonctionnent en mémoire partagée. Sur ces architectures, les latences d'accès à la mémoire locale au noeud est bien plus faible que pour un accès à la mémoire d'un noeud distant.

Il existe plusieurs ramasse-miettes parallèles qui exploitent la multiplicité des processeurs [12, 15, 16, 35, 48, 52]. Cependant, ils ont été développés durant l'ère des multiprocesseurs symétriques (SMP). En conséquence, l'impact du passage de la conception des architectures SMP aux architectures cc-NUMA sur les performances du ramasse-miette restent à déterminer. Dans cette thèse, j'ai pour but de trouver des réponses à ces questions. Plus précisément, je souhaite répondre aux questions suivantes :

1. Lors de l'utilisation des ramasse-miettes existants sur des architectures cc-NUMA, comment sont impacté leur performances ?

[2] Un cycle de ramassage est une itération du ramasse-miettes parmis toutes les itérations déclenchées pendant l'exécution de l'application.

2. Si les ramasse-miettes existants échouent à passer à l'échelle sur des architectures modernes, où se trouve le goulot d'étranglement ?
3. Est-ce que ces ramasse-miettes existants peuvent être modifiés pour éviter ces goulot d'étranglement ? Qu'est-ce que serait un ramasse-miette idéal pour une architecture NUMA ?

Un large spectre d'applications réelles et de benchmarks sont évalués pour déterminer si les ramasse-miettes continuent à améliorer leur performances avec chaque avancement matériel. Le ramasse-miette de la JVM, HotSpot, est très utilisé, et fais parti du kit de développement java OpenJDK. Les expériences sont effectuées sur une machine AMD de 48 cœurs répartis sur 8 nœuds. Cependant, l'évolution du matériel est émulé en effectuant des expériences sur un nombre de CPU qui varie de 1 à 48, avec un incrément fixe à chaque étape. Les résultats montrent que au delà d'un certain nombre de cœurs, le débit du ramasse-miette, soit le montant de donnée vivantes traitées par unité de temps, continue de diminuer avec un nombre de cœurs qui augmente. Deux sources d'inefficacités ont été identifiées expérimentalement : des cœurs oisifs et un trafic inter-nœuds excessif.

Les cœurs oisifs Puisque les ramasse-miettes parallèles ont été développés à l'ère des multiprocesseurs, leurs structures de données internes utilisent des primitives de synchronisation qui passent à l'échelle pour quelques fils d'exécutions, mais qui impactent très négativement le passage à l'échelle quand un grand nombre de fils d'exécutions ramasse-miette sont utilisés, ce qui est le cas sur du matériel moderne avec un grand nombre de cœurs. Aujourd'hui les fils d'exécutions de ramasse-miette passent un temps extrêmement important à se synchroniser et très peu de temps à faire des opérations sur les structures de données. Le problème à été résolu en remplaçant les structures de données internes au ramasse-miette par des structures de données efficaces et sans verrous. Utiliser ces structures a amélioré le passage à l'échelle durant l'exécution des collections parallèles. Un autre problème relié à la synchronisation vient avec le réveil et la suspension des fils d'exécutions du ramasse-miette au début et à la fin des phases parallèles du ramasse-miette. Pour résoudre ce problème, le code de gestion des fils d'exécutions est simplifié pour assurer que le réveil et la suspension des fils d'exécutions est fait avec un minimum de synchronisation. Ensemble, ces deux solutions ont complètement retiré l'attente explicite des fils d'exécutions du ramasse-miette pendant la phase parallèle du ramasse-miette, rendant cette phase sans verrou [20].

Trafic inter-nœud excessif. Les expériences révèlent qu'un nombre substantiellement haut d'accès distants inter-nœud sont effectués pendant une collection du ramasse-miette. Les deux raisons suivantes sont identifiées comme principales causes :

1. **Accès mémoire non local :** Le matériel cc-NUMA cache la nature distribuée de la mémoire à l'application. Ainsi l'application crée sans le savoir des références inter-nœud quand elle stocke une référence à un objet localisé sur un nœud donné dans la mémoire d'un autre nœud. À son tour, quand un fil d'exécution ramasse-miette traverse le graphe d'objet, il traverse silencieusement des références inter-nœud et ainsi traite des objets pouvant provenir de n'importe quel nœud. En conséquence, les fils d'exécution ramasse-miette accèdent souvent à la mémoire distante, ce qui cause un trafic inter-nœud important et ralenti les accès mémoires.
2. **Accès mémoire déséquilibrés :**

   Lorsque la mémoire physique associée au tas de l'objet est allouée depuis quelques nœuds mais est accédée par des threads s'exécutant sur chacun d'eux, cela sature le(s) contrôleur(s) mémoire de ces nœuds, ce qui limite fortement le passage a l'échelle. Ce problème est rencontré par les threads du GC car ils sont déployés sur tous les nœuds. Ce problème se pose en raison d'un comportement habituel des applications, où l'application a une phase d'initialisation en *série*. Cette phase force une majorité des pages physiques (qui correspondent au tas) à être allouées uniquement depuis quelques nœuds, où le(s) threads d'initialisation s'exécutent. Par ailleurs, puisque le même tas est utilisé après chaque cycle de ramasse miettes (GC), le problème se pose à chaque ramassage.

Pour surmonter ces problèmes, nous proposons NumaGiC [21]. D'abord, NumaGiC règle le problème des accès mémoire non-locaux en utilisant une conception essentiellement distribuée. À la base de NumaGiC, nous observons que le problème des accès mémoire non-locaux peuvent être résolus en s'assurant qu'un thread GC traite uniquement les objets présents dans la mémoire de son noœud. Lorsqu'un thread GC découvre un objet présent dans un nœud différent, il pourrait notifier le thread GC présent sur le nœud qui héberge l'objet, qui continue le scan localement. Cependant, appliquer strictement cette solution dégrade le parallélisme, puisqu'un thread GC reste inactif tant qu'il n'a pas d'objet locaux à récupérer, attendant que des threads GC d'autres nœuds atteignent des objet(s) présents sur son propre nœud. Ainsi, NumaGiC est conçu pour viser la localité des accès mémoire pendant le ramassage, sans dégrader le parallélisme du GC, en permettant à chaque thread GC de passer entre deux modes d'exécution. Un thread GC commence en *mode local*, dans lequel le GC se concentre sur la localité des accès mémoire. Dans ce mode, le thread GC suit strictement la contrainte décrite précédemment. Lorsqu'un thread GC n'a plus d'objets locaux à traiter, il entre en *mode vol*. Ce mode se concentre sur le parallélisme et permet à un thread GC de "voler" des références à des processus depuis d'autres nœuds, et d'accéder à ces référence par lui même, même si elles sont distantes. Un thread GC en mode vol peut régulièrement revenir en mode local pour vérifier si de nouvelles références locales sont disponibles, soit envoyées par un autre thread, soit découvertes en scannant une référence volée.

Ensuite, pour résoudre le problème des accès mémoire déséquilibrés, mais aussi pour préparer le tas au design essentiellement distribué, NumaGiC inclut un ensemble de politiques de placement adaptées au architectures NUMA. Ces politiques permettent d'abord que le GC équilibre les accès mémoire entre les nœuds, ce qui permet de résoudre le problème des accès mémoire déséquilibrés. Ceci permet de faire en sorte que chaque nœud contribue à l'utilisation de la mémoire, mais aussi de répartir le travail entre tous les threads GC. Les politiques permettent aussi de minimiser le nombre de références inter-nœuds. Puisqu'envoyer une référence est plus coûteux que d'accéder à distance l'objet correspondant, envoyer une référence est bénéfique uniquement si, en moyenne, l'objet référencé référence plusieurs autres objets sur son propre nœud. Dans ce cas, le coût de l'envoi d'une référence est amorti par la localité des accès mémoire du thread GC receveur.

NumaGiC est implémenté dans Hotspot, la machine virtuelle Java d'OpenJDK 7. Il cible les applications ayant de longues phases de calculs sur de larges ensembles de données pour lesquelles un ramasse-miettes bloquant *"stop-the-world"* orienté débit n'est pas approprié[3]. NumaGiC est basé sur Parallel Scavenge, le ramasse-miettes par défaut orienté débit de Hotspot. Les expérimentations comparent NumaGiC avec Parallel Scavenge. NumaGiC est évalué sur deux applications big-data largement utilisées, Spark [42] et Neo4j [32], avec une taille du tas Java allant de 110 GB à 350 GB. Nous évaluons également deux benchmarks de niveau industriel, SPECjbb2013 [44] et SPECjbb2005 [43],

[3] Un ramasse-miettes bloquant suspend l'application pendant la collection des objets afin d'éviter les accès concurrents au tas. Un ramasse-miettes bloquant est l'opposé d'un ramasse-miettes concurrent [25] qui favorise le temps de réponse aux dépens du débit car il requiert une synchronisation à grain-fin entre l'application et le ramasse-miettes.

ainsi que les benchmarks DaCapo 9.12 et SPECjvm2008. Les expérimentations ont été effectuées sur un serveur AMD de 48 cœurs et 8 noeuds ayant 256 GB de mémoire vive, ainsi que sur un serveur Intel Xeon E7 hyper-threadé de 40 cœurs avec 2 unités d'exécution par cœur, 4 noeuds et ayant 512 GB de mémoire vive. L'évaluation montre que :

- Sur les applications avec un tas de taille importante, NumaGiC améliore toujours les performances générales sur les deux machines. Avec la taille de tas permettant d'avoir les meilleures performances possibles avec les deux ramasse-miettes évaluées, NumaGiC améliore les performances générales des applications de 12% à 62% par rapport à Parrallel Scavenge. Ce résultat montre qu'une conception essentiellement répartie augmente considérablement les performances des applications avec un tas de taille importante sur des architectures NUMA, et semble améliorer les performances indépendamment de l'architecture.

- NumaGiC passe correctement à l'échelle par rapport au nombre de nœuds. Pour une taille de tas constant, le débit du ramasse-miette *i.e*, le nombre d'octets traités par unité de temps, augmente avec le nombre de nœuds. Par rapport à Parrallel Scavenge, NumaGiC ne dégrade jamais les performances.

- Sur les applications avec un tas de taille importante avec la taille la plus efficace pour tous les ramasse-miettes, NumaGiC augmente le débit de collection sur les deux machines de 2.2–5.2×, comparé à Parallel Scavenge.

- Sur les 33 applications de DaCapo 9.12 et SPECjbb2008 avec des plus petites charges de travail, NumaGiC augmente considérablement les performances de 19 applications de plus de 5%, et dégrade les performances d'une seule application de plus de 5% (de 8%). Ce résultat montre qu'une conception principalement répartie est presque toujours bénéfique et statistiquement bénéfique pour 50% des applications avec des charges de travail moderées.

Organisation du document. La thèse est structurée de la manière suivante :

- Le Chapitre 2 donne le contexte et les informations requis pour comprendre cette recherche. Il inclut des explications détaillées des architectures NUMA actuelles et du ramasse-miettes Parallel Scavenge qui est le point de comparaison de notre travail. Ce chapitre présente également l'état de l'art du domaine de cette recherche.

- Le chapitre 3 établie notre problématique. Nous présentons notre évaluation du passage à l'échelle de tous les ramasse-miettes d'OpenJdk7. Il présente aussi les goulots d'étranglement que nous avons identifiés dans le passage à l'échelle des ramasse-miettes.

- Le chapitre 4 présente notre solution au problème de cœurs oisifs durant une collection du ramasse-miette. Ce chapitre évalue notre solution sur des benchmarks réels.

- Le chapitre 5 présente NumaGiC, un ramasse-miette distribué que nous avons développé pour résoudre le problème de trafic inter nœud excessif. Il compare aussi les performances de NumaGiC avec le ramasse-miette de base sur un grand nombre d'applications.

- Finalement, le chapitre 6 conclue la thèse et discute des futures directions de recherche.

Bibliography

[1] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *proceedings of International Symposium on Memory Management '10*, pages 21–30. ACM, 2010.

[2] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with k42, an open-source, linux-compatible, scalable operating-system kernel. *IBM Syst. J.*, 44(2):427–440, Jan. 2005. ISSN 0018-8670. doi: 10.1147/sj.442.0427. URL http://dx.doi.org/10.1147/sj.442.0427.

[3] A. W. Appel. Simple generational garbage collection and fast allocation. *Software-Practice & Experience*, 19(2):171–183, 1989.

[4] 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 *proceedings of Symposium on Operating Systems Principles '09*, pages 29–44. ACM, 2009.

[5] 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 *proceedings of Object-Oriented Programming, Systems, Languages & Applications '06*, pages 169–190. ACM, 2006.

[6] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278667. URL http://doi.acm.org/10.1145/1278480.1278667.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855745.

[8] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1924943.1924944.

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

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2):16–29, 2010.

[11] 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 *proceedings of Architectural Support for Programming Language and Operating Systems '13*, pages 381–394. ACM, 2013.

[12] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *proceedings of International Symposium on Memory Management '04*, pages 37–48. ACM, 2004.

[13] D. Dice, M. Moir, and W. Scherer. Quickly reacquirable locks, Oct. 12 2010. URL http://www.google.com/patents/US7814488. US Patent 7,814,488.

[14] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of ACM*, 21(11):966–975, 1978.

[15] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *proceedings of Symposium on Principles of Programming Languages '93*, pages 113–123. ACM, 1993.

[16] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *proceedings of Java Virtual Machine Research and Technology Symposium '01*, pages 21–21. USENIX Association, 2001.

[17] H. Franke and R. Russell M. K. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *proceedings of Ottawa Linux Symposium, OLS '02*, pages 479–495, 2002.

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

[19] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 87–100, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL http://dl.acm.org/citation.cfm?id=296806.296814.

[20] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *proceedings of Architectural Support for Programming Language and Operating Systems '13*, pages 229–240. ACM, 2013.

[21] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: A garbage collector for big data on big numa machines. In *proceedings of Architectural Support for Programming Language and Operating Systems '15*, pages 661–673. ACM, 2015.

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

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

[24] R. L. Hudson and J. E. B. Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 48–57, New York, NY, USA, 2001. ACM. ISBN 1-58113-359-6. doi: 10.1145/376656.376810. URL http://doi.acm.org/10.1145/376656.376810.

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

[26] H. Kermany and E. Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134023. URL http://doi.acm.org/10.1145/1133981.1134023.

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

[28] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *proceedings of USENIX Annual Technical Conference '12*, pages 65–76. USENIX Association, 2012.

[29] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *proceedings of International Symposium on Memory Management '11*, pages 21–32. ACM, 2011.

[30] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: The programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.53. URL http://dx.doi.org/10.1109/SC.2010.53.

[31] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *proceedings of Principles of Distributed Computing '96*, pages 267–275. ACM, 1996.

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

[33] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629597. URL http://doi.acm.org/10.1145/1629575.1629597.

[34] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *proceedings of Object-Oriented Programming, Systems, Languages & Applications '09*, pages 377–390. ACM, 2009.

[35] OpenJDK Memory. Memory management in the Java HotSpot^{TM} virtual machine. Technical report, Sun Microsystems, 2006.

[36] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *proceedings of International Symposium on Memory Management '07*, pages 159–172. ACM, 2007.

[37] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *proceedings of Programming Language Design and Implementation '10*, pages 146–159. ACM, 2010.

[38] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring garbage collection with haswell hardware transactional memory. In *Proceedings of the 2014 International Symposium on Memory Management*, ISMM '14, pages 105–115, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2921-7. doi: 10.1145/2602988.2602992. URL http://doi.acm.org/10.1145/2602988.2602992.

[39] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. Eliminating read barriers through procrastination and cleanliness. In *proceedings of International Symposium on Memory Management '12*, pages 49–60. ACM, 2012.

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

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

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

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

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

[45] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *proceedings of Knowledge Discovery and Data Mining '12*, pages 1222–1230. ACM, 2012.

[46] G. L. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of ACM*, 18(9):495–508, 1975.

[47] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *proceedings of International Symposium on Memory Management '00*, pages 18–24. ACM, 2000.

[48] G. Tene, B. Iyengar, and M. Wolf. C4: the continuously concurrent compacting collector. In *proceedings of International Symposium on Memory Management '11*, pages 79–88. ACM, 2011.

[49] M. M. Tikir and J. K. Hollingsworth. NUMA-aware Java heaps for server applications. In *proceedings of International Parallel & Distributed Processing Symposium '05*, pages 108–117. IEEE Computer Society, 2005.

[50] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *proceedings of SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments '84*, pages 157–167. ACM, 1984.

[51] 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.

[52] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In *proceedings of International Symposium on Memory Management '12*, pages 3–14. ACM, 2012.