NUMA Awareness:
Improving Thread and Memory Management in the JVM

by

Maria Patrou

Bachelor of Informatics and Telecommunications, Kapodistrian University, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Kenneth B. Kent, PhD, Computer Science
Gerhard W. Dueck, PhD, Computer Science

Examining Board: Eric Aubanel, PhD, Computer Science, Chair
Suprio Ray, PhD, Computer Science,
Yves Losier, PhD, Mechanical Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK
August, 2017
©Maria Patrou, 2017
Abstract

Specific characteristics of hardware can affect the performance of applications. Hardware awareness becomes essential during runtime for both memory allocation, access and threads’ processor locations. Non-uniform Memory Access (NUMA) systems use different types of memory accesses and depend on several hardware resources in specific topologies. IBM’s Java Virtual Machine (JVM) examines the underlying hardware and identifies a NUMA architecture, while using memory and threads from the available nodes in a distributed way. A design for a node-isolated memory and thread policy is proposed. A node-heap reserve functionality to retrieve memory, based mostly on each node’s memory information, is further described. Finally, different modes regarding hardware and thread characteristics are used in order to find an optimal one and identify the application attributes that can benefit from specific modes, based on the underlying hardware.
Acknowledgements

I would like to thank my supervisors Kenneth B. Kent and Gerhard Dueck for all the support during my Master’s program, the CASA project manager Stephen MacKay for providing valuable help and my colleagues in the CASA project that provided help in every level. I would, also, like to thank my family and friends for the moral support they have always provided.

I would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, special thanks to the New Brunswick Innovation Foundation for contributing to this project and the Centre for Advanced Studies—Atlantic for access to the resources for conducting my research.
Table of Contents

Abstract ii
Acknowledgments iii
Table of Contents iv
List of Tables vii
List of Figures viii
List of Acronyms xii

1 Introduction 1

2 Background and Related Work 3
  2.1 Background ............................................... 3
  2.1.1 Java Virtual Machine ................................. 3
  2.1.2 Dynamic Memory Management .................... 3
    2.1.2.1 Balanced Garbage Collection ................. 4
    2.1.2.2 Heap Expansion ............................... 4
List of Tables

5.1 Best NumaVM Configuration - Baseline Execution Time(ms) . 68
5.2 Worst NumaVM Configuration - Baseline Execution Time(ms) 68
5.3 Best NumaVM configuration - Baseline - Baseline Numa Un-
    aware (BU) Execution Time(ms) . . . . . . . . . . . . . . . . 70
5.4 Average number of PGCs . . . . . . . . . . . . . . . . . . . . 70
5.5 Average duration (msec) of PGCs . . . . . . . . . . . . . . 71
5.6 Average duration (msec) of GGCs . . . . . . . . . . . . . . 71
5.7 Memory Pattern/Usage Comparison . . . . . . . . . . . . . . 72
5.8 Best NumaVM Modes . . . . . . . . . . . . . . . . . . . . . . 86
5.9 Internal JVM Threads . . . . . . . . . . . . . . . . . . . . . . 87
List of Figures

1.1 NUMA - 2 nodes with shared Last Level Cache . . . . . . . 2

2.1 CC-NUMA Organization [6] . . . . . . . . . . . . . . . . . . 9
2.2 A-NUMA simplified version from [26] . . . . . . . . . . . 10
2.3 L2 NUCA architecture [28] . . . . . . . . . . . . . . . . . . 11
2.4 COMA Organization . . . . . . . . . . . . . . . . . . . . . . 12
2.5 NUMA as a Cluster . . . . . . . . . . . . . . . . . . . . . . 13
2.6 Intel Nehalem-based NUMA Node . . . . . . . . . . . . . 14
2.7 IBM Power 750 NUMA Node . . . . . . . . . . . . . . . . 16

3.1 Execution time on different subsets of Nodes per server (Intel
Nehalem-based on left and IBM power 750 Express on right ) 21
3.2 Execution time on different subsets of Nodes per server (Intel
Nehalem Based on left and IBM power 750 Express on right) 21

4.1 Heap Organization . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Allocation Context Policy . . . . . . . . . . . . . . . . . . . . 27
4.3 Allocation Context Resize Policy . . . . . . . . . . . . . . . 33
4.4 Node Resize Policy - Overview . . . . . . . . . . . . . . . . 39
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMA</td>
<td>Non-uniform Memory Access</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>BGC</td>
<td>Balanced Garbage Collection</td>
</tr>
<tr>
<td>GGC</td>
<td>Global Garbage Collection</td>
</tr>
<tr>
<td>PGC</td>
<td>Partial Garbage Collection</td>
</tr>
<tr>
<td>AC</td>
<td>Allocation Context</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processor Unit</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessor</td>
</tr>
<tr>
<td>QPI</td>
<td>QuickPath Interconnect</td>
</tr>
<tr>
<td>DDR3</td>
<td>Double Data Rate Type 3</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Memory management and thread organization becomes important in systems with multiple cores and memory. Hardware awareness during runtime can improve application performance. Non-uniform Memory Access (NUMA) systems group different resources together into nodes that are connected with each other through interconnection structures. An example of a NUMA system is shown in Figure 1.1. The architecture necessitates different types of memory accesses. The location of a thread compared to the memory it accesses can create slowdowns, if the memory is farther from the thread’s node.

IBM’s Java Virtual Machine (JVM) identifies the NUMA hardware and organizes memory and threads accordingly in the Balanced Garbage Collection policy. The main characteristic is that memory and thread distribution around nodes is as balanced as possible. However, threads are allowed to
perform allocations in neighbor nodes. This approach, even though it keeps the memory balanced, creates remote accesses and can hurt locality that in turn results in performance degradation.

In this research, a technique that does not maintain memory balance between nodes is proposed, which is referred to as NumaVM. Instead, the goal of NumaVM is to reduce the remote accesses, by keeping threads and memory as close as possible, by not allowing threads to perform allocations to neighbor nodes. To this end, NumaVM utilizes a heap resize technique, which provides memory for each node separately. Also, different thread affinity policies are explored and used by NumaVM, based on hardware and application characteristics to identify the optimal one. Finally, in this thesis the results of a set of experiments comparing NumaVM with the current JVM’s implementation are presented.
Chapter 2

Background and Related Work

2.1 Background

2.1.1 Java Virtual Machine

A Java Virtual machine is a platform for JAVA programs. One of a JVM’s responsibilities is to make the compiled code hardware independent. It manages the memory at runtime, including memory access and update operations, such as object allocation [27].

2.1.2 Dynamic Memory Management

Almost all modern programming languages allow dynamic object allocation and deallocation, without knowing the object’s total size at compilation time and also providing its extended lifetime. These types of objects are allocated
on the heap, rather than on the stack of the caller’s procedure stack frame and are accessed by references from the stack directly or indirectly. As long as there are reference paths from the stack that point to the objects on the heap, the objects can be accessed by the program and they are considered alive. If they are inaccessible, they are considered dead.

For cleaning the heap and reclaiming memory of the dead objects, the garbage collection algorithm runs periodically. It is responsible for identifying and removing dead objects and returning the freed memory back for new object allocations. This procedure is referred to as Garbage Collection (GC) [25].

2.1.2.1 Balanced Garbage Collection

Balanced Garbage Collection (BGC) is a specific type of GC policy applied in the IBM JVM. It is beneficial for applications that use a large heap and its goal is to minimize global garbage collection times [34]. In BGC, the heap is divided into equal-sized regions. Object allocations happen in a specific type of region, called an Eden region. This architecture gives the opportunity to reclaim memory from subsets of regions, triggering Partial Garbage Collections (PGC) more frequently, rather than stopping the program flow to collect all dead objects (Global Garbage Collection-GGC) [7].

2.1.2.2 Heap Expansion

The number of GC cycles depends on the number and size of objects allocated by the applications. When there is no more free space for new allocations a
GC is triggered. In the Balanced policy, the GGC is called as a last resort to free up memory. If there is still not enough space, the heap expands. The JVM takes into consideration the active part of the heap as well as the maximum amount already reserved for the heap. This functionality does not increase the JVM’s storage, since it has already been allocated at startup [17]. During startup, the JVM defines the maximum heap size and, also the virtual memory it can use. It is not active, yet, though. Instead, it keeps the metadata from the chunks of virtual memory, for later usage. So, the same memory chunks are used, during application execution. More specifically, during expansion, the memory chunks are transformed to active memory, while during contraction, they become inactive.

Heap expansion and amount calculation is based on the allocation request requirements and JVM’s flags (-Xminf, -Xmaxt, -Xmine, -Xmaxe) regarding the free heap space, time spent in GC, minimum/maximum expansion amount [22], etc.

2.1.2.3 Heap Contraction

Occasionally, the heap can contract immediately following garbage collection. As in heap expansion, it is an operation that happens when the JVM has exclusive access (single-threaded). When the heap shrinks, the memory is not returned back to the operating system, but internally into the JVM, as described in the previous section. Heap contraction happens based on the allocation request requirements, JVM’s flags (-Xmaxf, -Xminf) regarding the
free heap space and if the heap has not been expanded in the previous three GCs. The contraction amount is calculated accordingly, using the JVM’s flags such as -Xmaxf and -Xms [22].

2.1.3 Non-uniform Memory Access

Non-uniform Memory Access is an architecture used in multiprocessors and parallel computing. It is a memory design in which memory access is not uniform. Several resources are grouped together in nodes, such as CPUs, memory and cache. Each node can be viewed as a Symmetric Multiprocessor (SMP) system with processors accessing the main memory and the I/O through a shared bus [33]. The nodes are connected to each other through an interconnection mechanism sharing the same virtual address space. The result of this architecture is accessing memory within a node (local access) is faster than accessing a neighboring one (remote access) [5].

The motivation for NUMA hardware relies on the fact that when the number of processors increases, traffic in their interconnection structure also increases causing a performance bottleneck because of the data and signals transmitted between the processors. However, building a system with multiple nodes, in which the traffic will be restricted only between the available resources of that particular node should reduce the overall traffic. Also NUMA takes advantage of nodes’ caches that should minimize the memory accesses, in general [33].

On the other hand, a NUMA system does not offer symmetric characteristics
and should require changes from software, when moving from a SMP system. The changes include page and process allocations and load balancing. Also, as the remote memory accesses increase, the performance is decreased [33].

2.1.3.1 Interconnection structures

Different types of interconnection structures are available. One approach for connecting the CPUs between nodes is the point-to-point interconnect. An example of a such structure is Intel’s QuickPath Interconnect (QPI). The components are connected in pairs and data is transferred in packets between them. The data is transmitted through directed connections, using groups of lanes for each direction [33].

On the other hand, the processors within the nodes can be connected with their memory through the interconnection bus. The bus is a pathway that connects multiple devices. All the data transmissions are available among the connected processors [33].

An example of how the cores and memory can be connected in an actual NUMA environment is the topology of DDR3 RAM, which is briefly described below. This memory uses memory controllers and channels to connect with the processors. This gives them the advantage of accessing their local memory directly and reducing any latency and bottleneck from a bus system. A high speed link, such as QPI, is used for the processors to access the memory from the neighboring nodes [20].
2.1.3.2 Non-uniform Memory Design Types

The non-uniform characteristic introduces many types of NUMA related memory designs. Below are some of them:

- **Symmetric NUMA or NUMA**: This is the most common type of NUMA design. Typically nodes contain memory, CPUs and cache, all, connecting to each other in a symmetric way, meaning that the distance from a node to any other node is the same, which introduces two types of distances, local and remote. This category can be further split into Cache-Coherent (CC-NUMA) and Non-Cache-Coherent (Non-CC-NUMA) NUMA.

Since each node contains its own cache architecture, it is essential to keep the same copies of data across nodes (cache coherency). Also, within each node there can be multiple caches for each processor or group of processors with a need for cache coherency, again. Whenever the data is changed in one place, they should be consistent to any other cache of the system. This can be achieved from the hardware itself using several protocols, such as snoopy and directory-based, CC-NUMA. In Figure 2.1, there is an example of CC-NUMA, using directories for each node that provide information about the cache status and node locations of their data. Cache coherency within each node is achieved through a snoopy bus, that broadcasts any change of a cache across all the other caches [33].
Another way to achieve cache coherence is to synchronize data using software (Non-CC-NUMA). These solutions offer the ability to reduce the hardware overhead. Some techniques include analyzing the code and adding instructions during compilation time in specific points to force cache coherence [33].

- **Asymmetric Interconnect NUMA:**

  In this case, the remote access is not uniform. Meaning, the interconnection links do not share the same characteristics. They have different bandwidth, the speed of sending data depends on the direction, they are used by a different number of nodes and they can be unidirectional. All of these introduce different types of accesses, including reads and
writes, between nodes [26]. An example of such a system is shown in Figure 2.2.

• **Non-uniform Cache Access (NUCA):**

Besides non uniform memory, there is non uniform cache. Traditionally, there are multiple levels of cache: L1 for data and instructions, L2 per processor and a larger L3. The farther away we are from the processor(s), the larger the cache is. One disadvantage of such an architecture is that the data can be replicated in the different cache levels.

In NUCA, the cores have their own Level 1 cache for data and instructions. The L2 cache is divided into smaller banks that are connected to each other through a 2D interconnection network, as it is shown in Figure 2.3. Each processor has its own local and central shared banks.
Figure 2.3: L2 NUCA architecture [28]
This architecture allows each core to use 16 local, 16 central and 224 distant banks [28]. Different mapping, search and movement policies are found for this model.

- **Cache Only Memory Architecture (COMA):**

  In NUMA, the data is allocated in the memory of a specific node (home node). On the other hand, in the COMA model, the memory blocks when requested from another node, are replicated into processor’s cache and node’s memory. This memory is called Attraction Memory (AM) [18]. The AM acts like a cache, while each line has a tag, containing information about the state and address of the data [35]. Different block placement mechanisms introduce different types of COMA designs.

  ![Figure 2.4: COMA Organization](Image)

- **NUMA as a Cluster:**

  Besides having a NUMA hardware, the non uniform memory access effect can happen within a cluster [30]. Each machine can represent a node. Naturally, the processors access the data of their own machine
faster rather than the ones found in a neighbor. Using software solutions, the global address space is distributed around the nodes, which are connected using the internet as their network (Figure 2.5).

![Figure 2.5: NUMA as a Cluster](image)

### 2.1.3.3 Example of NUMA Systems Architecture

After presenting the different types, two examples of NUMA architectures found in industry are shown below. Even though they are both NUMA servers, unique and different hardware characteristics are found in each one. Besides the types of resources that are found in a NUMA node and the interconnection link, there are different architecture characteristics of hardware including processors and cache layout.

First, in Figure 2.6, there is a CC-NUMA node from an Intel E-7520 Nehalem-based server. The server contains four nodes, with each node having four cores. Each core supports two hardware threads (hyperthreading). Also, each
node has 16GB DDR3 RAM. The node has a multilevel cache hierarchy in which each core has its own L1 instruction 4-way and data 8-way associative cache (32KB each) and a L2 8-way cache (256KB). The node has a shared 24-way associative L3 cache size of 18MB. This cache is shared across the cores in a way that, if all cores are active, they symmetrically use the cache, but if some are inactive, the other cores can use more [36]. Shared cache aims for sharing data between cores without using any interconnect structure [31]. Finally, remote access is 1.1 times slower than local access.

Figure 2.6: Intel Nehalem-based NUMA Node

In Figure 2.7, the CC-NUMA node from an IBM Power 750 (8233-E8B) server is shown. The server contains four nodes, with each one having eight cores. Each core supports four hardware threads. Also, each node has 64GB DDR3 RAM. Again, each node has a multilevel cache hierarchy. Each core has a 8-way L1 data cache (32KB), 4-way L1 instruction cache (32KB), L2 8-way associative cache (256KB) and a shared L3 cache (32MB). The L3 cache
is shared among the cores of the node, producing different latencies. Each core has its own private local region of 4MB, which is fast, but it can access the data from the rest of the cache, as it is shared [32]. In case a processor finds data in the shared regions, it can clone them to a private region [23]. The L3 cache is actually a NUCA design for eight cores [21]. Finally, remote access is four times slower than local access.

Despite their differences, both servers have simultaneous multithreading technology, also known as hyperthreading. This means that the physical processors can operate as multiple logical processors, sharing resources, such as cache, in order to achieve parallelism within them [33]. The Intel server has two logical cores per core, while the IBM server has four logical cores. Finally, the different cache associativities, introduce different times in searching and (re)placement policies at each cache level.

2.2 Related Work

2.2.1 Scheduling Algorithms based on Hardware

In the literature, there are schedulers that were created based on specific hardware characteristics of NUMA systems. A first example is a thread scheduler and memory placement policy based on a specific type of NUMA in which each interconnection link has different properties, causing asymmetric delays when connecting to different nodes (Asymmetric Interconnect NUMA). After taking the measurements, a decision is made on whether
Figure 2.7: IBM Power 750 NUMA Node
threads and memory pages should migrate to a specific subset of nodes [26]. Also, there are other cache-aware scheduling schemes focusing on memory management that take into consideration not only data locality but also, cache contention using specific metrics to decide about the mapping of processes [29].

2.2.2 Thread Affinity - Memory Management Policies

Thread affinity refers to threads bound on a specific central processing unit (CPU) or set of CPUs; they are assigned to run only on these CPUs. Currently, there are many policies regarding the order of cores on which threads can be pinned in a NUMA environment. Some memory management policies, used also for memory page allocations and processor order, are [24]:

- **First-touch.** Allocations happen on the node that the first processor has access to.

- **Round-Robin.** Threads are spread among nodes equally in a cyclic order.

- **Buddy.** Allocations are based on the requirements of data size and the free memory space. In CC-NUMA systems, such policies do not perform well, because they increase the memory access on a subset of nodes.

- **Skew-mapping.** The processors’ order is cyclic from left-to-right by
skipping the previous first processor in every cycle.

- **Prime-mapping.** The processors’ order is cyclic, but it uses a prime number (greater or equal to the total number of processors) to determine the exact order. The page allocations are modulo that number, which causes a cyclic order to subsets that are not requested sequentially.

### 2.2.3 NUMA Aware Garbage Collection

There are algorithms that perform specific tasks during the Garbage Collection cycle to avoid remote accesses. They achieve this by first processing objects that are allocated on the same node as the GC threads and then, by stealing object references from the other threads. Other implementations include heap organization in such a way that enables different garbage collection phases with some of them running concurrently with the application and in parallel with each other [19, 1, 2].

### 2.2.4 NUMA Awareness in Balanced Garbage Collection Policy

IBM’s JVM identifies the characteristics of a NUMA architecture. Using the Balanced Garbage Collection policy, the regions of the heap are bound to NUMA nodes. The majority of threads are pinned on nodes in a round robin fashion. Each thread tries to allocate objects in its own node and if that
fails, it borrows regions from the other nodes, in order to avoid unscheduled GCs or an OutOfMemoryError.

During garbage collection, GC threads mostly process objects that are located on the same node. Also, they try to move the objects to regions that are closer to the objects and threads referring to them [3, 7].

2.2.5 New approach

The proposed technique of this thesis does not utilize thread or memory migration; therefore, no calculations that increase the overall overhead occur. Instead, memory is organized per node, which enables the resizing of the heap without using extra locks. The multiple Thread Affinity policies presented require little calculation and place threads during their creation based on specific hardware and thread characteristics. Finally, even though the new technique does not take place during the garbage collection operations, because it aims in organizing the memory, it also affects the GCs.
Chapter 3

Motivation

A NUMA environment is not always beneficial for JAVA applications. The unique design of this model affects the performance and can either speed up or slow down an application’s execution time [3]. The way that the memory and threads of a program are distributed around nodes is a key factor.

The experimental evaluation on the DaCapo benchmarks [4] showed that the benchmarks’ experiments did not scale equally on different NUMA servers. The benchmarks were conducted using the \texttt{-s} small flag, the \texttt{numactl} command with \texttt{-N} and \texttt{-m} flag to bind the applications on specific nodes and the default JVM settings.

The experiment with Avrora (Section 5.2) was conducted on an IBM power 750 express (8233-E8B) and an Intel GenuineIntel 46 server as described in the previous chapter (Figures 2.6, 2.7). Avrora has a single external and multiple internal threads, that interact with each other [4]. Therefore, it
Figure 3.1: Execution time on different subsets of Nodes per server (Intel Nehalem-based on left and IBM power 750 Express on right)

appears that when memory and threads are spread across nodes, the remote accesses increase and the performance decreases. The results are similar for both servers, as shown in Figure 3.1. The execution time is increased on both of them when more than two nodes are used. However, on the Intel machine, the execution time does not increase further with more than three nodes, while on the IBM server the execution time becomes stable with more than two nodes. This is an example showing that the behavior of an application is important to actually benefit from the hardware.

Figure 3.2: Execution time on different subsets of Nodes per server (Intel Nehalem Based on left and IBM power 750 Express on right)
In Figure 3.2, the execution time of Xalan (Section 5.2) is shown, while running the same experiment on the above servers. Xalan is a multithreaded program with threads, being driven by the number of available hardware threads, that work with elements from a work queue [4]. Here, the execution times were improved by adding more nodes on the Intel server. On the IBM server, the execution time is approximately the same in any number of nodes. It probably indicates that there was limited interaction between the threads and running them on more nodes, given the opportunity to use more cores and more cache, did not increase the remote access dramatically. The architecture of the Intel server took advantage of that, but since the IBM node has already more cores and cache, it achieved a good execution time in every case. If resource utilization is a factor to consider, on Intel the best binding will be to use three nodes, while on IBM one node. So, here it appears that the application performed better when it was spread around nodes on one server, but stayed in one node for the other one. The hardware architecture awareness was important for this example.

Based on the above observation, the research focuses on organizing memory and threads. The JVM and region-based Garbage Collection policy (Balanced), which identifies the NUMA environment and organizes and retrieves the regions from the available nodes, was used. This gives the opportunity to manage regions more efficiently, since the information about the nodes they come from already exists.

The focus of this research is to create a per node isolated memory policy. To
this end, a per node resize technique is presented. During heap expansion
and contraction, which mostly happen right after the GC, the regions can be
organized. Therefore, there is no need to add any extra locks or even create
too much overhead. Since the heap will expand/contract regardless, the
goal is to resize the heap on each node, based on the information that can be
retrieved regarding the available and free memory per node. After organizing
the heap, different affinity policies are explored, based on the knowledge
about the hardware architecture and thread characteristics. Finally, certain
application attributes are investigated to find the most appropriate affinity
modes.
Chapter 4

Design and Implementation

Memory management under the Balanced GC policy of the IBM J9 JVM includes allocation and garbage collection [14] operations on regions of the heap. This chapter elaborates on heap organization in a NUMA environment, the allocation operation, the two types of GC that happen in Balanced and the details of the heap resize procedure. Then, the changes made regarding the memory organization, the allocation procedure and the resize policy are presented. Finally, the various thread affinity AC policies that are part of the design are explained.

4.1 Heap Organization

Balanced GC in the JVM supports NUMA. It divides the heap into equally sized regions. In case of NUMA, it collects the metadata from regions that
are physically stored in the available NUMA nodes. Each region stores information including its age. A region’s age indicates that it contains objects of similar age [7]. Age indicates the number of GCs an object has survived. Balanced offers 24 possible generations. Regions of age 0, are known as Eden space and they are used for allocating the newest objects. While regions of maximum age contain long-lived objects [15].

During startup, without any flags specified, there are two regions per node. Figure 4.1 shows an example of the initial heap using four nodes. Threads acquire memory from specific nodes and most of them are bound to CPUs from a specific node.

Figure 4.1: Heap Organization
4.2 Allocation

The JVM uses the Allocation Context (AC) structure to allocate objects into regions under the Balanced Garbage Collection (BGC) policy. BGC creates as many ACs as nodes plus one more. Therefore, for a NUMA environment with four nodes, there are five ACs, in which ACs 1-4 correspond to nodes 1-4 respectively and AC0 (discussed below). This technique creates ACs that can represent a physical node. Each AC that represents a node provides memory and CPUs to threads. When the threads are created, they acquire a specific AC, they never migrate to another one and they use only the CPUs provided by that node.

On startup, each AC has two regions (1 MB total), by default, available for usage. During the initialization procedure, the JVM directly expands, without following the usual 2-step procedure, that is described below. Afterwards, several threads are spawned. When the first thread is created, it acquires AC1, the second thread AC2, following a round robin order between the ACs. The allocation order is shown with green dotted arrows in Figure 4.2.

AC0, also known as the common allocation context, does not have any memory or CPUs to provide its threads. It is mostly used for the main thread, for threads that are identified as common ones, and for shared data. Threads can be identified as common by the JVM (main thread is an example that is identified as a common thread), or by the user with the numaCommonThreadClass flag. As a result, the related threads have no affinity; they can work on any
CPU of the available machine, decided by the OS. Also, since the common AC does not provide any memory, a stealing policy is required.

Every time an AC, including the common one, does not have enough regions it steals a region from its stealing cousin. Initially, the stealing cousin is the AC with the next lower AC number. As a result, the AC steals regions from the previous AC. An example is shown in Figure 4.2, in which the purple dashed arrows show the stealing policy. For example AC0 can steal a region from AC4, AC4 from AC3, AC3 from AC2 etc. However, the JVM aims at a distributed stealing policy, so that the same AC will not give its regions every time another AC needs more memory. Therefore, the next time an AC acquires a region from its cousin again, it will get it from its cousin’s cousin following a reverse round robin order.

Nevertheless, the original AC owner is kept, in order to return the region

![Figure 4.2: Allocation Context Policy](image)
back, when it becomes empty (after GC). The regions migrate temporarily as stealing storage or when they reach the maximum age. In the first case, they are used as a borrowed storage (foreign regions) from the contexts that need regions and they do not have enough (local regions). However after the GC, they return the borrowed ones to the pool of local regions of the original owners for future usage. The contexts keep track of the local and foreign regions they have. In the second case, regions migrate in the common allocation context, by just being part of the foreign regions of the common AC.

4.3 Collection

4.3.1 Global Garbage Collection

Heaps that become full and cannot satisfy the memory demands and specific System.gc() calls trigger the Global GC [11]. It is an operation that stops the normal execution of the program, so it should occur very rarely. It performs mark and sweep and less frequently compaction.

In the mark phase, all the live objects are traced and identified using a mark bit vector. Marking threads use input work packets, that contain mark stacks, to read the references and output packets to push the marked references as live objects. During this procedure the root objects are checked first and then their reference paths. In the JVM, the root objects consist of threads’ stacks and registers, JAVA static fields and local and global Java
Native Interface (JNI) references [12]. JNI references are objects that are passed to the JVM using the Java Native Interface and they refer to native code [22].

After the mark phase is completed the sweep phase starts. During the sweep phase, the mark bit vector is used to identify the memory that can be reclaimed. The GC examines the sequences of zeros in the bit vector and the objects’ sizes and reclaims any chunks with size more than the minimum free size (512 bytes for 32-bit platforms and 768 bytes for 64-bit platforms). Smaller areas are not added in the freelist, also known as “dark matter”, and they are reclaimed when objects next to them become garbage [16].

The compaction phase can occur after all garbage is removed in order to de-fragment the heap. It is a complicated procedure in which objects are moved and their references need to be updated, too. It occurs rarely in specific cases after the sweep phase, such as when there is not enough available memory to satisfy allocation requests etc [10]. The goal of the Balanced policy is to avoid global compactions because they are costly [7].

Finally, after all the above steps are executed, the regions, that still contain live objects are set to the maximum age. Therefore, they are not part of the eden space anymore and they migrate to the common AC, as described above.
4.3.2 Global Marking Phase

The Global Marking Phase (GMP) is a common operation that tracks the live objects of the heap. It occurs partially concurrently with application execution. It does not return free memory back to the heap, but instead it provides information for Partial Garbage Collection regarding which regions of the heap are more appropriate for collecting garbage [7].

4.3.3 Partial Garbage Collection

Partial GC collects garbage from a specific set of regions, called the collection set. PGCs occur very frequently and they are the most common GC cycles. They are operations that stop the normal execution of the application, like the Global GC. Their collection set contains eden regions, regions that are provided by the GMP or regions that are expected to find much garbage based on the statistics the GC gathered. Usually, they perform a copying collector in which live objects are moved to free regions. However they can switch to compaction mode in certain cases [7].

4.4 Heap Resize

4.4.1 Resize Points

Heap resize occurs after Global GC, Partial GC or in case of context allocation replenish failure (failed attempt to acquire context during allocation).
During context allocation replenish failure, the JVM attempts to perform context allocation and if that fails it tries to resize the heap before invoking a collection. Usually, the above allocation requests happen on the Thread Local Heap (TLH). In TLH, threads acquire specific areas of the heap exclusively, so locks are not required [13].

Also, the heap can expand by one region in case the collector needs a region for its Copy-Forward procedure (during Partial GC). The Copy forward operation copies live objects to regions, therefore the PGC needs to have available destination regions. Because, this does not happen exclusively and other threads can steal the extra region, a repeated expansion by one region occurs until the collector takes the region.

Finally, unlike all other resizing cases, the first time the JVM expands directly with an initial heap size.

### 4.4.2 Process

Heap resize happens in two steps: first the resize amount is calculated and then the heap is resized. The JVM first checks if the heap needs to contract and if it does not, then it checks if it needs to expand.

The JVM decides if the heap can contract. It checks the following conditions [22]:

- After GC there is not enough memory to meet the allocation request.

  In this case the heap is not allowed to shrink.
• Heap expansion has not occurred in the previous three GCs. In this case it can shrink.

• The maximum free memory space, as defined by the -Xmaxf flag (default is 60%) and the minimum free space by the -Xminf (default is 30%). The free heap size should be between these boundaries. If there is more than the maximum free it can shrink, but if it is less than the minimum free, it is not allowed to shrink.

Besides the above conditions the -Xsoftmx parameter is checked also. The flag provides a soft limit on maximum heap size [9]. If it is specified, the JVM checks if the softmx’s size is less than the memory size in order to do a more aggressive contraction. In the end, if the contraction conditions are met the shrinkage amount is calculated inside the allowed minimum and maximum free heap space boundaries.

Finally, there is a dependency between compaction and contraction. Compaction can take place before heap shrinkage in case the latter has not occurred in this GC cycle, no shrinkage and compaction happened on the previous GC cycle or there is not enough free contiguous memory (less than 10% of the required shrinkage amount) at the end of the heap [22].

Afterwards, the shrinkage amount is calculated (in terms of the number of regions). The regions should be returned to the original virtual space. These regions are gathered from each AC that represents a node, in a distributed way. A round-robin procedure occurs in which one region is returned at
a time from each AC starting from the one that resized the last time and moving backwards. An example is shown in Figure 4.3. If an expansion happened in a previous GC cycle causing the AC3 to be the last one that took a region, this AC will return first a region. So, if the JVM has to contract by 2 regions, AC 3 and 2 will return one region each. Then, the starting point will be AC1 for future contraction purposes. The contraction order is shown in purple dashed arrows.

On the other hand, if contraction does not meet the conditions, the JVM checks if the heap can expand. This happens if any of the following cases are true [22]:

- There is not enough free memory to meet the allocation request.
- The free memory space is less than the minimum one (-Xminf flag).

![Figure 4.3: Allocation Context Resize Policy](image-url)
• Too much time is spent in garbage collections. The threshold is defined by the -Xmaxt parameter with default being 13%.

If any of the above is true, the exact expansion amount is calculated. First, the heap expansion is calculated based on the minimum free amount and the current free amount. The amount must be within the boundaries defined by the maximum and minimum expansion amount -Xmine, -Xmaxe), -Xsoftmx and -Xmoi limits (if set to 0 no expansion is allowed [8]). The default values are 1 MB minimum and no maximum limit. Then, if the JVM had spent too much time during GC, the expansion amount is calculated as 17% of extra free space with respect of the above expansion limits. Finally, if there is not enough free memory, the expansion amount becomes equal to the allocation request amount.

After the expansion amount is calculated, the regions need to be added to the ACs. As in contraction, the regions follow a distributed policy. They are added to ACs in forward round-robin fashion. Again, there is a starting point based on which the AC resized the previous time to keep a memory balance between ACs. The expansion policy is shown with green dotted arrows in Figure 4.3. The example shows how the first expansion could take place after the startup with default settings. Since the JVM starts with two regions per AC, the first time an expansion happens, it will start from the first AC (the last AC that was expanded in the startup was the 4th). In this case the JVM adds 3 regions on the heap, so they will be added to ACs: 1, 2 and 3. In case there is a need for further expansion in the next GC, regions will be added.
in order 4, 1, 2, 3 etc, but in case the heap should contract regions they will be removed from ACs in the order 3, 2, 1, 4 etc.

4.5 Memory Organization

The key idea of this thesis is to organize the heap and threads in a NUMA architecture in order to avoid remote allocations and therefore reduce remote accesses and also, improve locality. To this end, a per node isolated design is used with the end goal to improve the performance of applications.

4.5.1 Design

NUMA systems can be considered as a collection of independent nodes and therefore, multiple memory resources. Applications can be organized in a way that they can use memory and cache from different nodes, rather than just one. The biggest drawback of this architecture is the expensive remote access. Therefore, in this design, the first aspect of organizing the heap is how to replace the remote accesses with local ones. This is achieved by keeping the memory associated with its threads tied tightly together. Instead of distributing the memory, a policy is created to keep memory closer to the threads that are using it, whenever possible.

The cornerstone of NUMA heap organization is resizing the heap. This gives the opportunity to expand or shrink the heap for each node. Each node can be treated as an isolated heap and it can provide more accurate information
about the memory its threads need to use. All threads are pinned on specific
nodes, so it becomes easier to identify the memory usage.
The biggest difference from the related work found in literature is that no
metrics are used to calculate and evaluate threads’ locality. The moment
the threads are created, they are assigned to a node and use memory only
from it. No memory or thread migration happens, since they are expensive
procedures. Also, different thread affinity policies based on hardware and
thread characteristics can occur at the moment of their creation, in order to
improve the temporal locality.
Overall, this technique has the following unique and important features:

• It works on top of a runtime heap management system that is mul-
tithreaded, splits the memory into numerous regions and organizes
threads and regions into per NUMA node allocation contexts.

• Each allocation context manages regions from exactly one NUMA node.

• On creation and based on any affinity and scheduling policy, all threads
are associated with a specific allocation context and pinned to its re-
spective NUMA node.

• It does not require the dynamic calculation of metrics, which would
otherwise cause significant overheads and delays.

• Because the heap is not managed as a whole but per node, the number
of regions that each allocation context uses can vary among them and
does not have to be equal; thus, local threads have access to the size of local memory they need.

- Threads and regions are never migrated to minimize remote accesses.

- The common AC, which otherwise was allowed to steal from any node, is now associated with exactly one node, further improving the locality of its threads and regions.

- Because contraction per node is enabled, depending on the mutator application’s execution patterns, the memory usage on some nodes can be reduced in order to conserve resources.

- No changes are required to the code of the mutator application since our technique abides by the expected runtime specifications.

To this end, the J9VM was used. Several changes to allocation contexts and heap resizing regarding the memory organization were made. First, the allocation context was changed.

4.5.1.1 Allocation Structure

As mentioned above, Allocation Contexts are not allowed to steal regions from any other ACs. They can only use their own available regions. Therefore, each AC uses only regions that are coming from the node it represents. Therefore, the Common AC cannot steal regions from the other ACs in a distributed way. Instead, it takes regions only from the last node giving
the ability to keep all the common AC data together. This technique, gives the opportunity to keep each AC tightly bound on its node. Therefore, all threads, including the common ones, are affinitized to their ACs’ nodes. So, each thread is working on the same node as the region/s it acquired.

The result of such a design is that now each AC’s memory demands depend on the working threads. Each AC will need more or less memory, independently from the others, based on the memory the threads are using. As a result, the remote access should be reduced and memory imbalance between nodes should occur naturally.

4.5.1.2 Resize Policy

As Figure 4.4 shows, a per node resize policy is described. Instead of treating the heap as one big chunk of memory, the resize calculations are split into the number of available nodes (ACs). A resize amount for each node is calculated, based on its available memory data. Both contraction and expansion checks happen, almost at each attempt for resize. There are cases, explained further below, that checks do not take place in order to avoid redundant checks.

Also, if there is information about the allocation request that failed, it is used. In contraction, the allocation node is excluded for calculating its contraction amount, and therefore it remains zero. Since the node cannot satisfy the request, it should expand rather contract. Therefore the contraction check for that node is not necessary and the contraction amount can be set immediately to zero without further calculations. On the other hand, in expansion, the
Figure 4.4: Node Resize Policy - Overview
size of the allocation request is used in the allocation node to calculate the expansion amount.

Also, the nodes that need to contract are not checked for expansion. Finally, after the calculation step is finished, contraction and expansion on the nodes happen, if needed.

During replenish allocation context failure, all nodes are checked for expansion, as described above. On the other hand, during the collector expansion, the amount for the allocation node is set immediately to one region size and all the other nodes have expansion size 0.

4.5.2 Implementation Details

More detailed information is presented regarding the details of the new implementation.

4.5.2.1 Node Resize

Since the ACs are isolated, the heap resize should serve each AC’s memory demands. So, a per AC/Node heap resize policy was created. More specifically, a whole functionality that is based on the existing one was created for handling each AC differently in each GC cycle.

As in the initial design, the heap resize is performed in two steps. First, a check whether the heap can be resized is performed and the resize amount for each AC is calculated separately. Then, each AC’s memory is resized.

In order to check and store the resize information per AC, a reverse linked

40
list was used, that stores the resize amount at each GC cycle for each node. In this step, information is stored for each node based on the node index id, rather than each AC, because in the end the regions are processed per node. Also, the list can be used anytime there is a need for accessing data on previous GC cycles faster.

Another important change occurs when the checks for expansion and contraction happen. All nodes are checked for expansion after GCs and after replenish allocation failure. Nodes are checked for contraction only after GCs, but the node that is related to the allocation request (in case there is such information) is excluded. This policy introduces a more aggressive expansion and a slightly conservative contraction. Since during the startup the JVM has a rather small heap (by default), the goal is to expand more at each node. Even though an allocation failure might trigger the resize point for a specific node, it is beneficial to check if the other nodes need more space, too, and not wait until they become full. On the other hand, the contraction policy needs to be used, when the workload of the other nodes is dropping and they do not need the extra regions. The most appropriate point for that check is after GCs, because some regions have been freed. In case of replenish allocation failure, the collector has not yet been invoked, so the contraction might be triggered prematurely. There might be regions that have not been used yet, but they might be needed soon. Also, the node that triggered the resize point is not checked for contraction, because it is the one that does not have enough memory, so it should not contract.
In order to achieve a per AC/Node strategy, it is important to get information from the allocation request regarding which node failed to satisfy it. To this end, the threads’ affinity is used. As described previously, all threads have affinity, so through that persisting feature, the node or AC index number can be stored into the allocation request’s information and passed through to the resize functionality. This happens during Thread Local Heap allocation failure, in which objects are allocated using the allocation contexts.

Another feature added in the resize policy is that the collector expansion immediately gives the region for the requested node. The collector might need a region during its copy-forward procedure. It requests a region from a specific allocation context. However, the requesting AC might not have any free regions. Therefore, an expansion by one region occurs on the appropriate AC. It is a function that operates while several threads are still running, so this happens repeatedly (as in the initial design), until the collector obtains its region.

Also, the functionality of the first expansion is maintained. The first time, the JVM should expand in a distributed way at all nodes, since they do not have any memory yet. However, the information for the future expansions is needed, thus the expansion amount is recorded per node using the resize list, mentioned above. This way is chosen because the first expansion is not always two regions per AC. It depends on the maximum heap size, and therefore region size (region size is determined by the heap size).

Finally, the last AC uses memory data from its own and the common AC.
Since the common AC is using regions only from the last node, its data is included in every type of calculation of the last AC. However, it should include regions from other nodes that have migrated by ownership there, when they reached the maximum age. Therefore, the other ACs include the regions that have remained there in their memory calculation data. No separation between the regions coming from the last node and the other ones is performed, since it should be a costly operation.

4.5.2.2 Resize Procedure

The first step of resizing the heap is to check for contraction and expansion. In comparison to the initial design, a few flag checks for adjusting the resize amounts, such as \(-Xmoi\), \(-XsoftMx\), \(-Xmine\) and \(-Xmaxe\), and logs regarding the resize procedure were skipped to reduce the overhead. The contraction and expansion size is stored for each node separately, as described above.

The first check is the contraction per node. First, the information about the time spent in the GC (for the whole heap) is gathered. Also, a check occurs whether the whole heap’s minimum free size is more than the one before a system GC was triggered. In that case, contraction does not happen, as in the initial design. No further checks occur and the amount for contraction for each node remains zero.

In any other case, the resize data are collected from the previous GC cycles. In order to obtain the most detailed information, the required number of GC cycles is calculated as the maximum number of either the heap expansion
count (default is zero) or the heap contraction count (default is three). Then each amount is calculated and stored into an array. If the amount for a node is zero, it means no resize probably happened in the previous GCs, if it is negative, contraction was preferred and if its positive the expansion was preferred. That shows the trend in terms of resize amount.

Then, a check whether there is an allocation size that needs to be satisfied (allocation request information) is performed, in order to exclude that node. If there is no such information, all nodes are checked. Then for each one of them, an inspection of their available regions for contraction occurs. If they do not have any, the node’s contraction amount is set to zero.

In any other case, the total and free memory amount for the AC is collected. The contraction size is found based on the AC’s current total and free size. The target contraction uses this information to find the target AC’s heap resize within the maximum and minimum contraction sizes, using the information about the time spent in the GC and rounding on the region size multiple. A final check is performed to ensure that contraction will not eliminate the AC’s memory, because of rounding on the region size.

Afterwards, the ACs with their contraction amount still zero, are checked for expansion. Again, a check is performed for gathering the allocation request information. If this is the case, the allocation size that needs to be met by the node that is responsible is gathered. All the other nodes do not have to satisfy that request, so their allocation size for them is set to zero.

The expansion size is calculated based on each AC’s local regions and free
memory size. It is calculated and adjusted based on the various parameters, such as -Xminf, -Xmaxf, time spent in the GC and bytes required for the allocation request applied on the AC’s memory information. Also, the number of GCs is not calculated, since the ACs need to expand regardless (even though the default number is zero, the check was avoided). In the end, the amount is rounded up to the region size multiple.

After the calculations are performed, the resize step happens. The array of the resize data is updated and both contraction and expansion take place, if there is at least one AC that has to contract and at least one that has to expand. If no ACs need to expand or contract the appropriate resize function is not called. Since each AC has its own resize amount, both operations iterate on each AC’s resize information and they either get more or return the appropriate regions to or from the corresponding node, until they meet the desired amount.

4.6 Thread Affinity AC Policy

The order of threads and which AC they acquire matters more in case of an isolated node design. Since they cannot migrate or use memory from any other context, placing them on different nodes gives different performance results. A work overload might occur on a specific node, if threads with high memory and CPU demands end up on the same node. However, splitting threads with high demands into different nodes can balance the system’s
workload.

Therefore, a Thread Affinity AC Policy is introduced, in which several configurations can be applied. More specifically, the order of the ACs that the threads acquire was changed and different policies (modes) were investigated. Modes regarding the order of the threads spawned around nodes based on hardware and thread characteristics were created. Applications can benefit from different modes. Different modes can either increase or decrease the performance based on the applications’ and hardware characteristics. With an isolated node memory policy, the distribution of the threads on the available nodes impacts the performance. Below, the modes that were created are described.

- The first one is the hop delay. A delay of the thread distribution around nodes occurs, in order to put closer together the threads that are temporally spawned with the end goal to improve temporal locality. In order to achieve that, the AC index functionality was changed, by creating the slots required to fulfill a whole AC cycle. For example, in case of one hop (the usual Round-Robin), the slots are four while for two hops is eight because the threads will be spawned in order: 1, 1, 2, 2, 3, 3, 4, 4. Now, a slot index is kept, in order to track the next index. Then, this slot index is divided by the hop delay and this gives the correct AC index. Below, it is shown how a full cycle would be performed with 1, 2 and 4 hops using four ACs. Figures 4.5, 4.6 and 4.7 show the different policies.
Figure 4.5: Thread Affinity AC Policy - 1 hop

Figure 4.6: Thread Affinity AC Policy - 2 hops
The four hops case needs 16 threads to fulfill a complete cycle, while the simple case of one hop just needs four. The first policy should be more appropriate for threads that have a more distributed design, while the four hops delay for threads, that can benefit from sharing the same cache. However, as the hop delay is increased, the possibility for an uneven number of threads per node is increased, too. That does not mean that it creates issues in every case, since the threads might not perform the same work. Also, in extreme cases, the hop delay can be such that threads will not complete one cycle and some node(s) might have fewer or no threads at all. Finally, the hop delay configuration can be used and combined with any of the ones described below.

- Another configuration that was investigated is to restrict the above
hops into the first three nodes and leave the last one for common allocation purposes only. Below, it is demonstrated how a two hop delay with three nodes is used for eight threads. The idea is to leave just one node for handling the common threads and apply the hop delay to the other nodes, as shown in Figure 4.8.

- Finally, the idea of separating the system and application threads was explored. As mentioned before, the JVM is multithreaded; system threads, such as GC, JIT and Finalizer threads can be identified and grouped together on the common AC. The application ones can use the other three nodes. Again, the purpose of this configuration is to improve temporal locality, by putting threads that share more similar characteristics together.
Chapter 5

Performance Evaluation

In this chapter the performance evaluation between the current implementation of the JVM with the new approach is described. The impacts of the new technique are presented by using different metrics. The result of different configuration modes regarding the thread AC policy with implemented changes made for the heap resize, ACs, etc. are presented. The new design is referred to as NumaVM following the AC configuration used, which is a J9VM project in which the changes mentioned above were applied for BGC. NumaVM is compared with the J9VM (Baseline) under Balanced GC. Both of them were modified in the same way to measure the required metrics. To minimize overhead and prevent any changes in the execution patterns, data was directly printed to the standard output without allocating any extra resources inside the JVM.
5.1 Performance Metrics

In this section the performance metrics that were measured in the Baseline and NumaVM are described.

- **Execution Time.** The execution time was measured from the execution time provided by the DaCapo benchmarks. The benchmarks were run 26 times, excluding the 3 best and 3 worst samples. Then, the average and standard deviation were calculated from the remaining 20 samples. This way, the sample contains values that are closer together.

- **Cache Misses.** The cache misses were measured using the `perf` command: `perf stat -e cache-misses`. The same technique for calculating the average and standard deviation of cache misses was used as the execution time, in order to have a more concentrated sample. These two metrics are calculated slightly different from the rest, since they are more susceptible to interference.

- **Number of PGCs, GGCs.** The number of PGCs and GGCs were found by running each benchmark 20 times and then the average and standard deviation were calculated.

- **Duration of PGCs, GGCs.** The duration of PGCs and GGCs were found by running each benchmark 20 times and then the average and standard deviation were calculated.
• **Memory Pattern/Usage.** Graphs and tables to show the memory pattern for each node during and at the end of execution were made. Since this is a nondeterministic metric, a specific run was picked and shown. So, 20 runs were performed and then their average execution time was calculated. The run that was closest to the average was selected and shown.

• **Thread Workload.** This metric shows the CPU percentage usage of the threads of the different benchmarks. It was calculated using the command `top -H -u <username> -d <delay screenshots> -b -n <repeats>`. For each run, screenshots from the above command are gathered, in which the CPU % usage is added for each process. Then, the average is calculated for each process. In case there are processes with the same name on the same screenshot, such as GC Threads, the weighted average is calculated using their percentage active time, instead. In order to calculate the active time for each process, on each run the process that appears more frequently is picked to find the duration of the application. The benchmarks run in the background, so selecting one of its processes gives that information. Therefore, for each thread/process the active time % is calculated using one process as a baseline. The benchmarks were run 20 times, and the average and standard deviation of CPU% usage and active time are gathered and presented. In the graphs below, only the threads that have CPU usage more than zero at least once (one run) are shown in CPU% Usage.
descending order. This metric gives more detailed information about the number, duration and CPU usage of the application threads.

5.2 Benchmarks

In order to evaluate the performance of the Baseline and the different modes of NumaVM, the following DaCapo Benchmarks, which were released in 2009, were used [4]:

- **Avrora.** It is a simulator of programs that run on AVR microcontrollers. Each simulated element uses a thread and there is high interaction between these simulator threads.

- **Batik.** It creates Scalable Vector Graphics (SVG) images. It uses worker threads for handling the images, while using the Java2D Disposer creates another worker thread, too.

- **Fop.** It takes as input an XSL-FO file, it parses and generates a PDF file.

- **H2.** It performs transactions on a banking application. It has client threads, one per hardware thread, a server thread for each client and other support threads.

- **Jython.** It is a python interpreter. It has a single external thread. It uses one thread per hardware thread internally, but it is mostly single threaded.
- **Lusearch.** It is a Multithreaded searching tool. It has one client thread per hardware thread and there is little interaction between these threads.

- **Pmd.** It analyzes Java source code. It is internally multithreaded with one worker thread per hardware thread.

- **Sunflow.** It is a tool for rendering images using ray tracing. It is multithreaded with one thread per hardware thread processing a tile of work. There is also the thread created by the Java2D Disposer.

- **Xalan.** It is an XML-to-HTML document transformer. It is multithreaded, with one thread per hardware thread, while each thread takes elements from a work queue.

### 5.3 Evaluation Results

In this section the results from running the Baseline and NumaVM are presented. The DaCapo Benchmarks were run on an Intel E-7520 Nehalem-based server with 4 NUMA nodes. Detailed information is provided on Section 2.1.3.3. The benchmarks were run using the default settings (maximum heap size 512MB). In Lusearch, the maximum heap size was set to 4GB (`-mx4g`), because of the need for more memory. On the other hand, H2 was run with `-mx8g` (8GB), in order to provide results with higher heap sizes (so different regions sizes).
5.3.1 NumaVM Configurations

The different modes of NumaVM are presented below:

- **4 nodes - 1 hop.** All nodes are used for all threads in round-robin fashion.

- **4 nodes - 2 hops.** All nodes are used for all threads with two hops delay. The two hops delay was tested in order to check a less distributed design than the default round robin one.

- **4 nodes - 4 hops.** All nodes are used for all threads with four hops delay. The four hops were chosen by the number of cores per node.

- **4 nodes - 8 hops.** All nodes are used for all threads with eight hops delay. The eight hops were chosen by the number of hardware threads per node.

- **3 nodes - 1 hop.** The last node is used only by the common AC and all the other threads acquire ACs in round-robin fashion.

- **3 nodes - 2 hops.** The last node is used only by the common AC and all the other threads acquire ACs with two hops delay.

- **3 nodes - 4 hops.** The last node is used only by the common AC and all the other threads acquire ACs with four hops delay.

- **3 nodes - 8 hops.** The last node is used only by the common AC and all the other threads acquire ACs with eight hops delay.
• 3 nodes - s4th - 1 hop. The last node is used only by the common AC and the system threads. The application threads are distributed in the first three nodes in round-robin fashion.

• 3 nodes - s4th - 2 hops. The last node is used only by the common AC and the system threads. The application threads are distributed in the first three nodes with two hops delay.

• 3 nodes - s4th - 4 hops. The last node is used only by the common AC and the system threads. The application threads are distributed in the first three nodes with four hops delay.

• 3 nodes - s4th - 8 hops. The last node is used only by the common AC and the system threads. The application threads are distributed in the first three nodes with eight hops delay.

5.3.2 Execution Time - Cache Misses

First, the execution time and cache misses were measured for the Baseline and NumaVM modes. Figures 5.1-5.18 show the results of the above configurations and the Baseline for each DaCapo benchmark. Overall, there is at least one NumaVM mode that provides better execution time than the Baseline. Each benchmark, though, benefits more or less from different modes. More specifically, Avrora has better performance when the number of hops is increasing. The best execution time and the fewest cache misses happen when the last node is used only by the common AC. Nevertheless, all but
Figure 5.1: Avrora Execution Time Comparison

Figure 5.2: Avrora Cache Misses Comparison
one configuration have better execution time than the Baseline and all modes result in fewer cache misses (Figures 5.1, 5.2).

Figure 5.3: Batik Execution Time Comparison

Figure 5.4: Batik Cache Misses Comparison

Batik has the best execution time and the fewest cache misses with 4nodes-
1hop mode. Here, the variations on the execution time have to do with the different combinations of number of hops and the number of nodes. Overall, using the 4th node for system threads, provides steadily less benefit than the other modes. However, the usage of three and four nodes can either produce better or worse results than the Baseline regarding the execution time. On the other hand, the cache misses are not always less than the Baseline (Figures 5.3, 5.4).

Fop always has better execution time in every mode, while the cache misses are worse only on the 3nodes-1hop configuration compared to the Baseline. The biggest benefit for both execution time and cache misses appears in 4nodes-1hop. The 3nodes configuration appears to be better in execution time steadily for each hop mode (Figures 5.5, 5.6) compared to the Baseline. H2 is a benchmark that has high standard deviation in both cache misses and execution time. Nevertheless, there are configurations that achieved lower standard deviation and better execution times and cache misses at the same time, such as 4nodes-2hops, 3nodes-1hop, etc. It appears that only one case is slower than the Baseline (3nodes-2hops) with high standard deviation and three cases producing more cache misses. Overall, the biggest benefit appears when system threads are used on the 4th node (Figures 5.7, 5.8).

All NumaVM modes produce better execution times and fewer cache misses in the Jython benchmark. Overall, combinations of nodes and hops produce better results in execution time than separating the system threads and putting them in the last node, while the opposite effect happens on the cache
Figure 5.5: Fop Execution Time Comparison

Figure 5.6: Fop Cache Misses Comparison
Figure 5.7: H2 Execution Time Comparison

Figure 5.8: H2 Cache Misses Comparison
Figure 5.9: Jython Execution Time Comparison

Figure 5.10: Jython Cache Misses Comparison
misses (Figures 5.9, 5.10).

On the other hand, Lusearch produces better execution time only in two modes in which the system threads are assigned to the last node. However, there always more cache misses than the Baseline. Also, there are modes that greatly harm the execution time and cache misses, such as 3nodes-2hops (Figures 5.11, 5.12).

Pmd produces the fewest cache misses in every 3nodes mode and the fastest execution time in every s-4th mode. The most cache misses and the worst execution times appear on the 4 nodes mode, in which some modes are worse than the Baseline (Figures 5.13, 5.14).

Sunflow produces very good results in execution time and cache misses in every s-4th configuration. Cache misses are always less than the Baseline, but there is one case that the Baseline is faster than the NumaVM (Figures 5.15, 5.16).

Xalan runs faster in s-4th mode in every hop delay. Cache misses are fewer except in 4nodes-8hops configuration. However, the execution time is worse in 3nodes configuration and increasing while the hop delay is increasing. The 4nodes configuration provides better execution time than the Baseline depending on the number of hops used (one and two hops are better than four and eight) (Figures 5.17, 5.18).

In Table 5.1 the best configurations of NumaVM in comparison with the Baseline are presented, regarding the execution time. The best speedups are achieved in Sunflow (30.84%) and Avrora (35.7%), while the worst ones
Figure 5.11: Lusearch Execution Time Comparison

Figure 5.12: Lusearch Cache Misses Comparison
Figure 5.13: Pmd Execution Time Comparison

Figure 5.14: Pmd Cache Misses Comparison

65
Figure 5.15: Sunflow Execution Time Comparison

Figure 5.16: Sunflow Cache Misses Comparison
Figure 5.17: Xalan Execution Time Comparison

Figure 5.18: Xalan Cache Misses Comparison
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NumaVM Mode</th>
<th>NumaVM</th>
<th>Baseline</th>
<th>Speedup (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>3nodes-8hops</td>
<td>8586</td>
<td>13353</td>
<td>35.70</td>
</tr>
<tr>
<td>Batik</td>
<td>3nodes-4hops</td>
<td>15399</td>
<td>15963</td>
<td>3.53</td>
</tr>
<tr>
<td>Fop</td>
<td>4nodes-1hop</td>
<td>4721</td>
<td>5107</td>
<td>7.56</td>
</tr>
<tr>
<td>H2</td>
<td>3nodes-s4th-1hop</td>
<td>23906</td>
<td>32797</td>
<td>27.11</td>
</tr>
<tr>
<td>Jython</td>
<td>3nodes-2hops</td>
<td>29405</td>
<td>30681</td>
<td>4.16</td>
</tr>
<tr>
<td>Lusearch</td>
<td>3nodes-s4th-1hop</td>
<td>3012</td>
<td>3183</td>
<td>5.38</td>
</tr>
<tr>
<td>Pmd</td>
<td>3nodes-s4th-1hop</td>
<td>4565</td>
<td>4985</td>
<td>8.44</td>
</tr>
<tr>
<td>Sunflow</td>
<td>3nodes-s4th-2hops</td>
<td>6491</td>
<td>9386</td>
<td>30.84</td>
</tr>
<tr>
<td>Xalan</td>
<td>3nodes-s4th-1hop</td>
<td>3288</td>
<td>3841</td>
<td>14.40</td>
</tr>
</tbody>
</table>

Table 5.1: Best NumaVM Configuration - Baseline Execution Time(ms)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NumaVM Mode</th>
<th>NumaVM</th>
<th>Comparison (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>4nodes-1hops</td>
<td>13427</td>
<td>-0.55</td>
</tr>
<tr>
<td>Batik</td>
<td>3nodes-2hops</td>
<td>15973</td>
<td>-0.06</td>
</tr>
<tr>
<td>Fop</td>
<td>3nodes-s4th-1hop</td>
<td>5037</td>
<td>1.38</td>
</tr>
<tr>
<td>H2</td>
<td>3nodes-2hops</td>
<td>33652</td>
<td>-2.61</td>
</tr>
<tr>
<td>Jython</td>
<td>3nodes-s4th-4hops</td>
<td>30103</td>
<td>1.88</td>
</tr>
<tr>
<td>Lusearch</td>
<td>3nodes-2hops</td>
<td>4088</td>
<td>-28.42</td>
</tr>
<tr>
<td>Pmd</td>
<td>4nodes-4hops</td>
<td>5452</td>
<td>-9.36</td>
</tr>
<tr>
<td>Sunflow</td>
<td>4nodes-2hops</td>
<td>9395</td>
<td>-0.10</td>
</tr>
<tr>
<td>Xalan</td>
<td>3nodes-8hops</td>
<td>5057</td>
<td>-31.67</td>
</tr>
</tbody>
</table>

Table 5.2: Worst NumaVM Configuration - Baseline Execution Time(ms)
are Batik (3.53%) and Jython (4.16%). On the other hand, in Table 5.2 in which the worst configurations are compared with the Baseline, Jython and Fop can always benefit from the NumaVM implementation, while Xalan and Lusearch can have 32% and 28% slowdowns. The best configurations and the Baseline were also compared with Baseline Numa Unaware. In this case, the Balanced policy is used with one Allocation Context for all threads. There is again always benefit in execution time as it is shown on Table 5.3. Sunflow (35%) and Lusearch (31%) benefit more, while Batik and Jython produce smaller speedups, 1.9% and 2.9% respectively. So, even though some benchmarks can benefit from the NUMA implementation in the JVM (Lusearch, Pmd, Sunflow and Xalan), the best configuration of this research’s technique can produce better execution times. However, this creates the need to find the best configuration based on the application characteristics, that will be discussed later.

5.3.3 GC-Memory Impacts

The impacts on PGCs and GGCs were analyzed. Batik and Fop perform 36% and 47% fewer PGCs, while H2 and Lusearch do 11% and 5% more respectively (Table 5.4). H2 and Avrora spent 34% and 56% less time in PGCs, but Fop and Pmd spent 4.84% and 4.9% more (Table 5.5). The number of GGCs is the same for Baseline and NumaVM for all benchmarks. However, Lusearch spent the least time in GGC with 36%, while H2 the most time with 10% (Table 5.6).
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>BU</th>
<th>NumaVM-BU</th>
<th>Baseline-BU</th>
<th>NUMA Benefit?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>12136</td>
<td>29.25</td>
<td>-10.03</td>
<td>FALSE</td>
</tr>
<tr>
<td>Batik</td>
<td>15696</td>
<td>1.90</td>
<td>-1.70</td>
<td>FALSE</td>
</tr>
<tr>
<td>Fop</td>
<td>5013</td>
<td>5.82</td>
<td>-1.88</td>
<td>FALSE</td>
</tr>
<tr>
<td>H2</td>
<td>28902</td>
<td>17.29</td>
<td>-13.48</td>
<td>FALSE</td>
</tr>
<tr>
<td>Jython</td>
<td>30284</td>
<td>2.90</td>
<td>-1.31</td>
<td>FALSE</td>
</tr>
<tr>
<td>Lusearch</td>
<td>4395</td>
<td>31.46</td>
<td>27.57</td>
<td>TRUE</td>
</tr>
<tr>
<td>Pmd</td>
<td>5047</td>
<td>9.57</td>
<td>1.23</td>
<td>TRUE</td>
</tr>
<tr>
<td>Sunflow</td>
<td>10001</td>
<td>35.10</td>
<td>6.15</td>
<td>TRUE</td>
</tr>
<tr>
<td>Xalan</td>
<td>4050</td>
<td>18.83</td>
<td>5.18</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Table 5.3: Best NumaVM configuration - Baseline - Baseline Numa Unaware (BU) Execution Time(ms)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>NumaVM PGCs</th>
<th>NumaVM PGCS StDev</th>
<th>Baseline PGCs</th>
<th>Baseline PGCS StDev</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>9.3</td>
<td>1.0</td>
<td>14.1</td>
<td>1.1</td>
<td>34.4</td>
</tr>
<tr>
<td>Batik</td>
<td>10.3</td>
<td>0.6</td>
<td>16.1</td>
<td>0.4</td>
<td>36.2</td>
</tr>
<tr>
<td>Fop</td>
<td>11.6</td>
<td>0.5</td>
<td>22.0</td>
<td>0.0</td>
<td>47.3</td>
</tr>
<tr>
<td>H2</td>
<td>27.0</td>
<td>0.6</td>
<td>24.3</td>
<td>0.7</td>
<td>-11.3</td>
</tr>
<tr>
<td>Jython</td>
<td>75.2</td>
<td>2.5</td>
<td>99.3</td>
<td>4.1</td>
<td>24.3</td>
</tr>
<tr>
<td>Lusearch</td>
<td>38.1</td>
<td>0.2</td>
<td>36.4</td>
<td>0.5</td>
<td>-4.5</td>
</tr>
<tr>
<td>Pmd</td>
<td>20.9</td>
<td>0.8</td>
<td>26.1</td>
<td>1.9</td>
<td>20.0</td>
</tr>
<tr>
<td>Sunflow</td>
<td>58.6</td>
<td>3.4</td>
<td>57.1</td>
<td>2.7</td>
<td>-2.7</td>
</tr>
<tr>
<td>Xalan</td>
<td>31.9</td>
<td>0.4</td>
<td>33.0</td>
<td>0.6</td>
<td>3.34</td>
</tr>
</tbody>
</table>

Table 5.4: Average number of PGCs
### Table 5.5: Average duration (msec) of PGCs

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>NumaVM PGCs duration</th>
<th>NumaVM PGCs duration StDev</th>
<th>Baseline PGCs duration</th>
<th>Baseline PGCs duration StDev</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>10.8</td>
<td>4.1</td>
<td>24.6</td>
<td>1.0</td>
<td>56.0</td>
</tr>
<tr>
<td>Batik</td>
<td>25.0</td>
<td>1.3</td>
<td>25.1</td>
<td>1.0</td>
<td>0.3</td>
</tr>
<tr>
<td>Fop</td>
<td>21.5</td>
<td>1.7</td>
<td>20.5</td>
<td>0.9</td>
<td>-4.8</td>
</tr>
<tr>
<td>H2</td>
<td>56.2</td>
<td>1.3</td>
<td>84.8</td>
<td>9.1</td>
<td>33.7</td>
</tr>
<tr>
<td>Jython</td>
<td>19.6</td>
<td>0.5</td>
<td>20.5</td>
<td>0.6</td>
<td>4.6</td>
</tr>
<tr>
<td>Lusearch</td>
<td>22.3</td>
<td>0.3</td>
<td>21.8</td>
<td>0.8</td>
<td>-2.3</td>
</tr>
<tr>
<td>Pmd</td>
<td>22.6</td>
<td>1.2</td>
<td>21.6</td>
<td>1.9</td>
<td>-4.9</td>
</tr>
<tr>
<td>Sunflow</td>
<td>32.7</td>
<td>3.3</td>
<td>37.5</td>
<td>1.1</td>
<td>12.6</td>
</tr>
<tr>
<td>Xalan</td>
<td>26.3</td>
<td>0.9</td>
<td>25.5</td>
<td>1.2</td>
<td>-3.2</td>
</tr>
</tbody>
</table>

### Table 5.6: Average duration (msec) of GGCs

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>NumaVM GGCs duration</th>
<th>NumaVM GGCs duration StDev</th>
<th>Baseline GGCs duration</th>
<th>Baseline GGCs duration StDev</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>134</td>
<td>71</td>
<td>127</td>
<td>72</td>
<td>-5.4</td>
</tr>
<tr>
<td>Batik</td>
<td>133</td>
<td>75</td>
<td>123</td>
<td>71</td>
<td>-4.3</td>
</tr>
<tr>
<td>Fop</td>
<td>132</td>
<td>73</td>
<td>136</td>
<td>77</td>
<td>2.9</td>
</tr>
<tr>
<td>H2</td>
<td>5511</td>
<td>3306</td>
<td>5002</td>
<td>2883</td>
<td>-10.2</td>
</tr>
<tr>
<td>Jython</td>
<td>487</td>
<td>273</td>
<td>458</td>
<td>253</td>
<td>-6.2</td>
</tr>
<tr>
<td>Lusearch</td>
<td>178</td>
<td>100</td>
<td>278</td>
<td>157</td>
<td>36.1</td>
</tr>
<tr>
<td>Pmd</td>
<td>140</td>
<td>80</td>
<td>142</td>
<td>79</td>
<td>1.4</td>
</tr>
<tr>
<td>Sunflow</td>
<td>187</td>
<td>105</td>
<td>198</td>
<td>110</td>
<td>5.8</td>
</tr>
<tr>
<td>Xalan</td>
<td>174</td>
<td>98</td>
<td>165</td>
<td>93</td>
<td>-5.9</td>
</tr>
</tbody>
</table>

71
The impacts on memory usage vary, too. H2, Lusearch and Xalan use 21, 20 and 21% less memory, while Jython uses 50% and Fop 35% more (Table 5.7). In Figures 5.19-5.36, the detailed memory pattern for each benchmark is shown. Based on Baseline, there are four different memory patterns:

- **Aggressive steady memory growth.** Avrora, Batik, Fop and H2 (Figures 5.19, 5.20, 5.21, 5.22, 5.23, 5.24, 5.25, 5.26) grow their heap steadily during their execution time. In the NumaVM implementation, each node is managed and grown differently. In H2, the heap of the last node is increasing remarkably more than the other nodes. Almost the same applies for Batik, but the other nodes use more memory with number of regions being closer to each other. Fop uses the most memory in the last node, but the second node appears to have a

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>NumaVM (KB)</th>
<th>NumaVM Regions</th>
<th>Baseline (KB)</th>
<th>Baseline Regions</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batik</td>
<td>87040</td>
<td>10-8-7-63</td>
<td>84992</td>
<td>28-28-28-28</td>
<td>-2</td>
</tr>
<tr>
<td>Fop</td>
<td>69632</td>
<td>26-31-30-83</td>
<td>51712</td>
<td>42-42-41-41</td>
<td>-35</td>
</tr>
<tr>
<td>H2</td>
<td>720896</td>
<td>9-55-13-59</td>
<td>917504</td>
<td>26-25-25-25</td>
<td>21</td>
</tr>
<tr>
<td>Jython</td>
<td>138240</td>
<td>77-42-12-139</td>
<td>92160</td>
<td>45-45-45-45</td>
<td>-50</td>
</tr>
<tr>
<td>Lusearch</td>
<td>1220608</td>
<td>73-184-77-44</td>
<td>1527808</td>
<td>86-86-85-85</td>
<td>20</td>
</tr>
<tr>
<td>Pmd</td>
<td>193536</td>
<td>76-75-70-77</td>
<td>175104</td>
<td>94-93-93-93</td>
<td>-11</td>
</tr>
<tr>
<td>Sunflow</td>
<td>311296</td>
<td>88-83-81-47</td>
<td>293376</td>
<td>95-95-94-94</td>
<td>-6</td>
</tr>
<tr>
<td>Xalan</td>
<td>153088</td>
<td>164-143-120-181</td>
<td>193536</td>
<td>144-143-143</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 5.7: Memory Pattern/Usage Comparison
steady growth that almost reaches the memory used by the last node. Finally, Avrora appears to have much memory in the last node most of application execution time, but in the end the third node grows more. In all the above situations, the new implementation managed to grow the heap accordingly for each node, resulting in more or less memory used compared to the Baseline

- Conservative memory growth - rarely aggressive. Lusearch and Xalan (Figures 5.27, 5.28, 5.29, 5.30), appear to grow at a small rate, until they reach a point that they increase their growth rate. Pmd (Figures 5.31, 5.32) has a steady growth, until it reaches some points that result in bigger growth than the rest of the resize points. In the NumaVM implementation, all nodes increase with different rates and in the end less memory is used than the Baseline for Lusearch and Xalan.
Figure 5.20: Avrora Baseline Memory Usage Comparison

Figure 5.21: Batik NumaVM Memory Usage Comparison
Figure 5.22: Batik Baseline Memory Usage Comparison

Figure 5.23: Fop NumaVM Memory Usage Comparison
Figure 5.24: Fop Baseline Memory Usage Comparison

Figure 5.25: H2 NumaVM Memory Usage Comparison
In Lusearch, the heap of the first three nodes grows almost at the same rate, while in the end the last node starts growing faster until it reaches their memory sizes. In Xalan, the memory growth rate of the first three nodes is again similar. However, the memory of the last node never becomes as large as the others. Instead, its memory growth is more steady. Regarding Pmd, more memory is used in the end than the Baseline. A steady growth is noticed and when the application is close to the end of the resize points, a very aggressive growth appears, especially in the second node.

- Memory growth - rarely shrinkage. In Sunflow (Figures 5.33, 5.34), the heap grows most of the time, but there is a point that the heap shrinks steadily and then grows back. In the NumaVM implementation, there are points at which the heap of the first three nodes slightly shrank. In the end, the new implementation uses more memory than the Baseline.
Figure 5.27: Lusearch NumaVM Memory Usage Comparison

Figure 5.28: Lusearch Baseline Memory Usage Comparison
Figure 5.29: Xalan NumaVM Memory Usage Comparison

Figure 5.30: Xalan Baseline Memory Usage Comparison
Figure 5.31: Pmd NumaVM Memory Usage Comparison

Figure 5.32: Pmd Baseline Memory Usage Comparison
Figure 5.33: Sunflow NumaVM Memory Usage Comparison

Figure 5.34: Sunflow Baseline Memory Usage Comparison
• Memory growth - shrinkage. Jython (Figures 5.35, 5.36) appears to be a benchmark that the heap grows and shrinks at different points. The NumaVM implementation did not identify the difference in the workload as well as in the other cases, resulting in memory usage of 50% more (which is the worst case). It seems to be a result of a more conservative shrinkage policy.
5.3.4 Overall Impact per Benchmark

More specifically the following impact happens for each benchmark in comparison with the Baseline:

- **Avrora.** In 3nodes-8hops mode, Avrora achieves 35.7% speedup and fewer cache misses, 34.4% fewer PGCs with 56% less duration. However, the GGC consumes 5.42% more time. Finally, 11.67% less memory is used in comparison with the Baseline. Regarding the memory usage per node, a huge memory imbalance occurs with the first node having three regions, while the third has 25. Overall, Avrora performs better in almost every aspect compared to the Baseline. The NumaVM mode seems to be optimal, because there are major differences in performance between the Baseline.

- **Batik.** In 3nodes-4hops mode, Batik achieves a 3.53% speedup, slightly fewer cache misses, 36.14% fewer PGCs with 0.3% less duration. However, the GGC requires 4.32% more time and 2.41% more memory is used. A memory imbalance occurs with the last node having 63 regions and the second and third having 8 and 7 regions respectively. The major benefit here is the number of PGCs, while the differences in the other metrics are not so different.

- **Fop.** In 4nodes-1hop mode, Fop achieves a 7.56% speedup and fewer cache misses, 47.27% fewer PGCs with 4.84% more duration. The GGC spends 2.94% less duration, but the benchmark uses 34.65% more
memory. So, even though the number of PGCs are almost half fewer, much more memory is used. Regarding the memory pattern per node, the first three nodes have almost the same number of regions, but the last one has more than twice as many regions.

• **H2.** In 3nodes-s4th-1hop, H2 achieves a 27.11% speedup and fewer cache misses, 11.34% more PGCs with 33.65% less duration, but 10.18% more duration in GGC. Regarding the memory usage, 21.43% less memory is used. The second and fourth node use the most regions, while the first and third use over four times less. Overall, H2 benefits from the NumaVM configuration in many aspects.

• **Jython.** In 3nodes-2hops, Jython achieves a 4.16% speedup and fewer cache misses, 24.27% fewer PGCs with 33.65% less duration and 6.24% more duration in GGC. However regarding the memory usage, Jython uses 50% more memory in the NumaVM implementation than with the Baseline. Even though, in most metrics Jython performs better in the NumaVM implementation than the Baseline, there is a big drawback of using 50% more memory.

• **Lusearch.** In 3nodes-s4th-1hop mode, Lusearch achieves 5.38% speedup, slightly more cache misses, 4.53% more PGCs with 2.26% more duration. The duration of GGC is 35% less that the Baseline. Finally, the NumaVM implementation uses 20% less memory. As in most cases, in the end memory usage imbalance occurs. However, the major benefit
here is the decrease in memory usage.

- **Pmd.** In 3nodes-s4th-1hop Pmd achieves 8.44% speedup and fewer cache misses, 20% fewer PGCs with 4.9% more duration and 1.4% less duration during GGC. Regarding the memory usage, 10.53% less memory is used. The difference, here, though in comparison with any other benchmark, is that, there is a better memory balance. The difference in regions used by each node is not as significant.

- **Sunflow.** In 3nodes-s4th-2hops Sunflow achieves 31% speedup and fewer cache misses, 3.34% more PGCs with with 12.64% less duration and 5.83% less duration in GGC. Regarding the memory, the first three nodes use roughly the same number of regions, while the last node uses almost half of the others. Also, the NumaVM implementation uses 6.11% more memory than the Baseline. So, the biggest benefit using the new implementation is the speedup, but the drawbacks are more PGCs and memory.

- **Xalan.** In 3nodes-s4th-1hop Xalan achieves 14.4% speedup and fewer cache misses, 3.34% fewer PGCs with 3.17% more duration, and 5.89% more duration in GGCs. Finally, 20.9% less memory is used, with each node using a different number of regions and the last one using the most. The biggest benefit is the memory usage, but at the cost of the duration of GCs.
Overall, each benchmark is affected differently by the NumaVM implementation. The most common characteristic is that there is better execution time in every case, but sometimes the drawback is using much more memory and/or GCs.

### 5.3.5 Thread Workload Analysis

In order to understand how and why the different configurations affect each benchmark, a more detailed analysis of the application behavior is required regarding the thread workload and distribution. In Table 5.8 the applications are grouped based on their best NumaVM configuration. This gives an insight into the common characteristics the benchmarks share, when they perform better in the same mode. In addition to the best configuration, the second best mode was added for some benchmarks with pink and underlined in order to provide a further understanding.

<table>
<thead>
<tr>
<th>3nodes 2hops</th>
<th>3nodes 4hops</th>
<th>3nodes 8hops</th>
<th>4nodes 1hop</th>
<th>3nodes-s4th 1hop</th>
<th>3nodes-s4th 2hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jython</td>
<td>Batik</td>
<td>Avrora</td>
<td>Fop</td>
<td>Fop</td>
<td>Sunflow</td>
</tr>
<tr>
<td>Fop</td>
<td></td>
<td></td>
<td>Jython</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Batik</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lusearch</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Pmd</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Xalan</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sunflow</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>H2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lusearch</td>
</tr>
</tbody>
</table>

Table 5.8: Best NumaVM Modes

In Table 5.9, the different internal JVM threads created for each application are presented. Even in the case of the HelloWorld program six JVM applica-
tion threads are created. There are 32 GC threads (one per hardware thread) and seven JIT threads that compose the system threads. There is also one application thread and one system thread both identified as common ones. During the thread AC acquirement, the system threads first are bound to ACs and then the application ones, with the exception of one application thread.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application Threads</th>
<th>System Threads</th>
<th>Nodes used for App Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avrora</td>
<td>11</td>
<td>39</td>
<td>1,3</td>
</tr>
<tr>
<td>Batik</td>
<td>13</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Fop</td>
<td>6</td>
<td>39</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>H2</td>
<td>39</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Jython</td>
<td>6</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Lusearch</td>
<td>37</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Pmd</td>
<td>38</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Sunflow</td>
<td>72</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>Xalan</td>
<td>37</td>
<td>39</td>
<td>1,2,3</td>
</tr>
<tr>
<td>HelloWorld</td>
<td>6</td>
<td>39</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.9: Internal JVM Threads

Therefore, the hops can actually affect the number of nodes used for application threads. A representative example is Avrora. While all the other benchmarks use the available nodes based on the number of nodes they are allowed to be bound to, Avrora ends up using just two nodes for all its application threads. This is an indication why Avrora performs better in the 3nodes-8hops mode. Since the threads interact with each other, they can benefit from sharing the cache while placing them on the same node.
Also, there are only 11 internal JVM threads being created, and the number of hardware threads per node is eight, so splitting them into two nodes is feasible.

Regarding the rest of the benchmarks, two major groups are identified. The benchmarks can either perform better by separating the systems threads or not. Lusearch, Pmd, Xalan, H2 and Sunflow perform better in s4th mode, while Avrora, Jython, Batik and Fop on three and four nodes mode, with the last three of them being grouped on the 4nodes-1hop mode when including the second best configurations. As Table 5.9 shows, the benchmarks that produce better results in the s-4th mode have at least 37 threads (almost as many as the number of hardware threads). On the other hand, the other benchmarks have a few threads (no more than 13). Also, Lusearch, Pmd, Sunflow and Xalan (from Table 5.3) benefit when they are using the JVM with NUMA option on. Therefore, it seems that the benchmarks that benefit from the JVM’s Numa configuration benefit from the new implementation using the s4th mode.

These results show that separating system threads from application ones, when there is a large number of application threads, can improve locality and execution time. Such a separation means that the system threads need to access regions from the other nodes using remote accesses, which also results in remote accesses during GCs. Lusearch, Pmd and Xalan have longer PGC duration, while Sunflow and H2 have shorter compared to the Baseline. Thus, it is expected that GCs will be impacted due to increased interconnection
traffic when collecting garbage from any of the first three nodes. Nevertheless, this depends on the collection sets of the PGCs. The benefit, though, should be improved locality even during GCs, since the GC threads work on the same node and share the L3 cache.

In Figures 5.40, 5.42, 5.43, 5.44, 5.45, the workload of the processes from H2, Lusearch, Pmd, Sunflow and Xalan is shown. These benchmarks have many application threads working with Avg CPU% higher than their system threads (GC, JIT threads). All the above benchmarks have similar behavior. More specifically, Lusearch and H2 contain 32 application threads that are active \(\sim 51\%\) and use \(\sim 49\%\) CPU on average respectively. Xalan has 32 application threads being active \(\sim 51\%\), but Avg CPU usage is \(\sim 80\%\). It is noticed that the more CPU time a process uses the less time it is active. However, Xalan is an application whose threads use a work queue and therefore, each thread is not expected to perform the same task on every run. Regarding Pmd, the 32 application threads are active on average \(\sim 23\%\) and their average CPU is \(\sim 67\%\). So, in Pmd there are threads that are active for a short period of time and have high CPU requirements. Finally, Sunflow has 55 application threads (Java2D_Disposer thread not included, because the average CPU time is close to \(\sim 0\%\) that are active approximately \(\sim 41\%\), with CPU requirements \(\sim 55\%). However, it appeared that 23 of the aforementioned threads are active at most \(\sim 6\%\) of the execution time.

All the above thread workload patterns show that even though the applications have many threads, all the threads are not active during the whole
application’s execution time. Even though the 3nodes-s4th mode offers four fewer cores (24 available processor units), it might not necessarily cause work overload. To this direction, the hops should provide a separation between threads that should lead to better resources utilization.

On the other hand, Jython and Fop, as shown in Figures 5.39, 5.41, have the main thread working with $\sim 87\%$ Avg CPU%. Also, the rest of the system threads are active during the whole application’s procedure. Batik (Figure 5.38) shows similar characteristics with the main threads using approximately $\sim 80\%$ CPU. It has four application threads that they use on average $\sim 27\%$ and being active $\sim 1.2\%$. All the above benchmarks produce better results in 4nodes-1hop. Separation between system and application threads, would not be as beneficial since the main thread is the one that works the most and should be bound in the last node with fewer GC threads.

However, Jython produces better execution time in 3nodes-2hops configuration. In this mode, all system threads are distributed around the first three nodes. So, the main thread (that is a common thread, so it is bound to the last node) does not have to share any CPU with any system threads. The same applies for Batik whose best configuration is the 3nodes-4hop. It also has four more application threads with very small active time, so the locality of these threads should matter, too (the node that will be bound to and how close together they are).

Finally, Avrora (Figure 5.37) has six threads that are active $\sim 77\%$ of the execution time with Avg CPU usage $\sim 58\%$. It seems that these threads
when put together, because of their high interaction with each other, produce better results. From a thread workload point of view, one node contains eight processors, so it can support the 3nodes-8hops mode, even though they are active most of the application execution time.

![Avrora Thread Workload](image)

Figure 5.37: Avrora Thread Workload
Figure 5.38: Batik Thread Workload

Figure 5.39: Fop Thread Workload
Figure 5.40: H2 Thread Workload
Figure 5.41: Jython Thread Workload
Figure 5.42: Lusearch Thread Workload
Figure 5.43: Pmd Thread Workload
Figure 5.44: Sunflow Thread Workload
Figure 5.45: Xalan Thread Workload
Chapter 6

Conclusion and Future Work

Overall, each benchmark is affected differently by the NumaVM implementation in the Balanced Garbage Collection-BGC. The most common characteristic is that there is better execution time in every case, but sometimes the drawback is the use of much more memory and/or more GCs. The number and the duration of the GCs are mostly affected by the regions. This approach changes the location of the regions, and therefore the collection sets. Objects are allocated in different nodes than those in the baseline, therefore the location of the garbage and live objects are different, too. Heap fragmentation and locality should be affected, too. Even though a per node heap resize technique was designed, the PGCs contain subsets of regions from different nodes. Also, this technique did not have the overhead of many flags that were skipped from resize amount calculations. Instead information can be used, as part of the future work, from the trend the different node resize
shows in previous resize points.

Knowing \textit{a priori} the memory pattern of the application can give an insight into the selection of the appropriate configuration, since it can harm the application if the wrong one is selected. Applications that can scale out in a NUMA environment, therefore have many internal threads, can benefit by putting the application threads closer to each other and force the system ones into one node. Applications with fewer threads can be grouped to either all nodes or just the first three. In case the main thread is the one that performs the most work, using four nodes seems more appropriate, since all nodes will have system threads.

However, in case there are other application threads, putting all the non-common threads in three nodes and leaving the last one for common AC purposes can produce better locality. To this end, the hops provide a delayed distribution that can separate and group together temporal threads.

Nevertheless, it is noticed that the majority of the benchmarks benefit from a mode that includes using fewer nodes at least by the systems threads. It seems that it is better to have remote accesses during GCs, than distributing all threads in all nodes, regardless of the number and the duration of GCs. However, the server used has remote accesses 1.1 times slower than local ones that impacts the performance.

As part of future work, a resize stealing policy could be investigated. In the current technique, in case the node uses all its available memory, it cannot steal any regions from any other node causing the application to be out of
memory. The current solution could not identify the different workloads during the application's execution time well enough, using more memory than needed. Also, a flag or a dynamic way to use the desired mode can be added. Finally, more experiments in other types of NUMA servers, can produce more results regarding the benefits and drawbacks of this technique.
References


104


Vita

Candidate’s full name: Maria Patrou
University attended (with dates and degrees obtained): Bachelor Of Informatics and Telecommunications, Kapodistrian University, 2012
Publications: N/A
Conference Presentations: N/A