Enabling Hardware Affinity in JVM-based Applications: A Case Study for Big Data

Roberto R. Expósito, Jorge Veiga, and Juan Touriño

Universidade da Coruña, CITIC, Computer Architecture Group, A Coruña, Spain
{roberto.rey.exposito,jorge.veiga,juan}@udc.es

Abstract. Java has been the backbone of Big Data processing for more than a decade due to its interesting features such as object orientation, cross-platform portability and good programming productivity. In fact, most popular Big Data frameworks such as Hadoop and Spark are implemented in Java or using other languages designed to run on the Java Virtual Machine (JVM) such as Scala. However, modern computing hardware is increasingly complex, featuring multiple processing cores aggregated into one or more CPUs that are usually organized as a Non-Uniform Memory Access (NUMA) architecture. The platform-independent features of the JVM come at the cost of hardware abstraction, which makes it more difficult for Big Data developers to take advantage of hardware-aware optimizations based on managing CPU or NUMA affinities. In this paper we introduce `jhwloc`, a Java library for easily managing such affinities in JVM-based applications and gathering information about the underlying hardware topology. To demonstrate the functionality and benefits of our proposal, we have extended Flame-MR, our Java-based MapReduce framework, to provide support for setting CPU affinities through `jhwloc`. The experimental evaluation using representative Big Data workloads has shown that performance can be improved by up to 17% when efficiently exploiting the hardware. `jhwloc` is publicly available to download at https://github.com/rreye/jhwloc.

Keywords: Big Data · Java Virtual Machine (JVM) · Hardware Affinity · MapReduce · Performance Evaluation

1 Introduction

The emergence of Big Data technologies offers great opportunities for researchers and scientists to exploit unprecedented volumes of data sources in innovative ways, resulting in novel insight discovery and better decisions. Distributed processing frameworks are the great facilitators of the paradigm shift to Big Data, as they enable the storage and processing of large datasets and the application of analytics techniques to extract valuable information from such massive data.

The MapReduce paradigm [5] introduced by Google in 2004 and then popularized by the open-source Apache Hadoop project [19] has been considered as a game changer to the way massive datasets are processed. During the past decade, there has been a huge effort in the development of Big Data frameworks.

Some projects focus on adapting Hadoop to take advantage of specific hardware (RDMA-Hadoop [16]), or to provide improved performance for iterative algorithms by exploiting in-memory computations (Twister [7], Flame-MR [22]). Other frameworks have been designed from scratch to overcome other Hadoop limitations such as the lack of support for streaming computations, real-time processing and interactive analytics. Many of these frameworks are developed under the umbrella of the Apache Software Foundation: Storm [10], Spark [24], Flink [2], Samza [14]. Despite the large amount of existing frameworks, Hadoop and its ecosystem are still considered the cornerstone of Big Data as they provide the underlying core on top of which data can be processed efficiently. This core consists of the Hadoop Distributed File System (HDFS) [17], which allows to store and distribute data across a cluster of commodity machines, and Yet Another Resource Negotiator (YARN) [20], for the scalable management of the cluster resources. New frameworks generally only replace the Hadoop MapReduce data engine to provide faster processing speed.

Unlike the C++-based original MapReduce implementation by Google, the entire Hadoop stack is implemented in Java to increase portability and ease of setup. As Big Data users are generally non-expert programmers, the use of Java provides multiple appealing features for them: object orientation, automatic memory management, built-in multithreading, easy-to-learn properties, good programming productivity and a wide community of developers. Moreover, later Java releases adopt concepts from other paradigms like functional programming. A core feature of Java is cross-platform portability: programs written on one platform can be executed on any combination of software and hardware with adequate runtime support. This is achieved by compiling Java code to platform-independent bytecode first, instead of directly to platform-specific native code. The bytecode instructions are then executed by the Java Virtual Machine (JVM) that is specific to the host operating system and hardware combination. Furthermore, modern JVMs integrate efficient Just-in-Time (JIT) compilers that can provide near-native performance from Java bytecode.

Apart from Hadoop, most state-of-the-art Big Data frameworks are also implemented in Java (e.g., Flink, Storm 2, Flame-MR). Other frameworks rely on Scala (Spark, Samza), whose source code is compiled to Java bytecode so that the resulting executable runs on a JVM. However, the platform-independent feature provided by the JVM is only possible by abstracting most of the hardware layer away from developers. This makes it difficult or even impossible for them to access interesting low-level functionalities in JVM-based applications such as setting hardware affinities or gathering topology information for performing hardware-aware optimizations. The increasing complexity of multicore CPUs and the democratization of Non-Uniform Memory Access (NUMA) architectures [11] raise the need for exposing a portable view of the hardware topology to Java developers, while also providing an appropriate API to manage CPU and NUMA affinities. JVM languages in general, and Big Data frameworks in particular, may take advantage of such API to exploit the hardware more efficiently without having to resort to non-portable, command-line tools that offer limited functionalities. The contributions of this paper are:

- We present `jhwloc`, a Java library that provides a binding API to manage hardware affinities straightforwardly in JVM-based applications, as well as a means to gather information about the underlying hardware topology.
- We implement the support for managing CPU affinity using `jhwloc` in Flame-MR, a Java-based MapReduce framework, to demonstrate the functionality and benefit of our proposal.
- We analyze the impact of affinity on the performance of in-memory data processing with Flame-MR by evaluating six representative Big Data workloads on a 9-node cluster.

The remainder of the paper is organized as follows. Section 2 provides the background of the paper and summarizes the related work. Section 3 introduces the `jhwloc` library and its main features. Section 4 describes a case study of integrating `jhwloc` in Flame-MR, and presents the experimental evaluation. Finally, our concluding remarks are summarized in Section 5.

2 Background and Related Work

Exploiting modern hardware platforms requires in-depth knowledge of the underlying architecture together with appropriate expertise from the application behaviour. Current architectures provide a complex multi-level cache hierarchy with dedicated caches (one per core), a global cache (one for all the cores) or even partially shared caches. Moving a computing task from one core to another can cause performance degradation because of cache affinities. Simultaneous Multithreading (SMT) technologies [6] such as Intel Hyper-Threading (HT) [13] involve sharing computing resources of a single core between multiple logical Processing Units (PUs). This fact also means to share cache levels so that performance may be even reduced in some particular cases. Furthermore, NUMA architectures [11] are currently widely extended, in which memory is transparently distributed among CPUs connected through a cross-chip interconnect. Hence, an access from one CPU to the memory of another CPU (i.e., a remote memory access) incurs additional latency overhead due to transferring the data through the network. As an example, Figure 1 shows the hierarchical organization of a typical NUMA machine with two octa-core CPUs that implement two-way SMT, so 8 cores and 16 PUs are available per CPU. All this complexity of the hardware topology of modern computing platforms is considered a critical aspect of performance and must be taken into account when trying to optimize parallel [18] and distributed applications [23].

Nowadays, the hardware locality (hwloc) project [9] is the most popular tool for exposing a static view of the topology of modern hardware, including CPU, memory and I/O devices. This project solves many interoperability issues due to the amount and variety of the sources of locality information for querying the topology of a system. To do so, hwloc combines locality information obtained

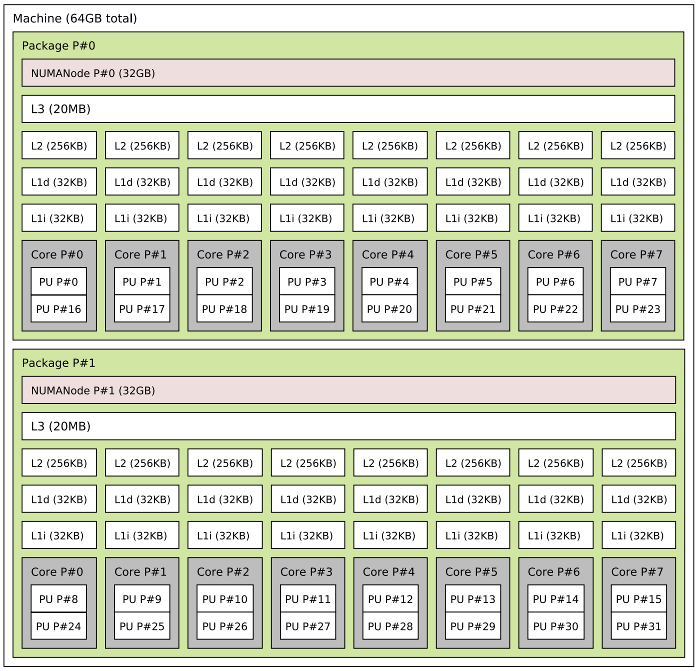


Fig. 1: Overview of a NUMA system with two octa-core CPUs

from the operating system (e.g., /sys in Linux), from the hardware itself (cpuid instruction in x86), or using high-level libraries (numactl) and tools (lscpu). Moreover, hwloc offers APIs to interoperate with device-specific libraries (e.g., libibverbs) and allows binding tasks (e.g., process, thread) according to hardware affinities (CPU and memory) in a portable and abstracted way by exposing a unified interface, as different operating systems have diverse binding support. As a consequence, hwloc has become the de facto standard software for modeling NUMA systems in High Performance Computing (HPC) environments. In fact, it is used by most message-passing implementations, many batch queueing systems, compilers and parallel libraries for HPC. Unfortunately, hwloc only provides C-based APIs for gathering topology information and binding tasks, whereas the JVM does not offer enough support for developing hardware-aware applications as it only allows to obtain the number of cores available in the system.

Overseer [15] is a Java-based framework to access low-level data such as performance counters, JVM internal events and temperature monitoring. Among its other features, Overseer also provides basic information about hardware topology through hwloc by resorting to the Java Native Interface (JNI). Moreover, it allows managing CPU affinity by relying on the Linux Kernel API (sched.h). However, the support provided by Overseer for both features is very limited. On the one hand, topology information is restricted to a few Java methods that only allow to obtain the number of available hardware resources of a certain type (e.g., cores per CPU), without providing any further details about cache hierarchy (e.g., size, associativity), NUMA nodes (e.g., local memory) or additional functionality to manipulate topologies (e.g., filtering, traversing). On the other hand, support for CPU binding is limited to use Linux-specific affinity masks that determine the set of CPUs on which a task is eligible to run, without providing convenient methods to operate on such CPU set (e.g., logical operations) or utility methods for performing more advanced binding operations in an easy way (e.g., bind to one thread of the last core of the machine). Furthermore, no kind of memory binding is provided by Overseer. Our `jhwloc` library allows to overcome all these limitations by currently providing support for more than 100 methods as the Java counterparts of the hwloc ones.

There exist few works that have evaluated the impact of managing hardware affinities on the performance of Big Data frameworks. In [1], authors analyze how NUMA affinity and SMT technologies affect Spark. They manage affinities using numactl and use hwloc to obtain the identifier of hardware threads. Their results reveal that performance degradation due to remote memory accesses is 10% on average. Authors in [4] characterize several TCP-H queries implemented on top of Spark, showing that NUMA affinity is slightly advantageous in preventing remote memory accesses. To manage NUMA affinity, they also rely on numactl. However, both studies are limited to evaluating a single machine, they lack the assessment of the impact of CPU binding on performance and they do not provide any useful API for gathering topology information and managing hardware affinities for JVM-based languages. Hence, our work extends the current state-of-the-art in all these directions.

3 Java Hardware Locality Library

The Java Hardware Locality (`jhwloc`) project has been designed as a wrapper library that consists of: (1) a set of standard Java classes that model hwloc functionalities using an object-oriented approach, and (2) the necessary native glue code that interfaces with the C-based hwloc API. In order to do so, `jhwloc` uses JNI to invoke the native methods implemented in C. Hence, `jhwloc` acts as a transparent bridge between a client application written in any JVM-based language and the hwloc library, but exposing a more friendly and object-oriented Java API to developers. One important advantage provided by `jhwloc` is that it frees Java developers from interacting directly with JNI calls, which is considered a cumbersome and time-consuming task. So, our library can be easily employed in Java applications to perform hardware-oriented performance tuning.

3.1 Java API

Currently, `jhwloc` provides Java counterparts for more than 100 hwloc functions, covering a significant part of its main functionality. A complete Javadoc documentation that describes all the public methods together with their parameters is publicly available at the `jhwloc` website. Basically, the `jhwloc` API enables Java developers to: (1) obtain the hierarchical hardware topology of key computing elements within a machine such as: NUMA nodes, shared/dedicated caches, CPU packages, cores and PUs (I/O devices are not yet supported); (2) gather various attributes from caches (e.g., size, associativity) and memory information; (3) manipulate hardware topologies through advanced operations (e.g., filtering, traversing); (4) build "fake" or synthetic topologies that allow querying them without having the underlying hardware available; (5) export topologies to XML files to reload them later; (6) manage hardware affinities in an easy way using bitmaps, which are sets of integers (positive or null) used to describe the location of topology objects on the CPU (CPU sets) and NUMA nodes (node sets); and (7) handle such bitmaps by providing advanced methods to operate over them through the hwloc bitmap API.

It is important to remark that, unlike C, the JVM provides an automatic memory management mechanism through the built-in Garbage Collector (GC), which is in charge of performing memory allocation/deallocation without interaction from the programmer. Hence, the memory binding performed by the hwloc functions that manage memory allocation explicitly (e.g., *alloc_membind*) or migrate already-allocated data (*set_area_membind*) cannot be supported in `jhwloc`. NUMA affinity management is thus restricted to those functions for performing implicit memory binding: *set_membind* and *get_membind*. These functions allow to define the current binding policy that will be applied to the subsequent calls to malloc-like operations performed by the GC.

3.2 Usage Example

As an illustrative usage example, Listing 1 presents a Java code snippet that shows the simplicity of use of the `jhwloc` API. Basic hardware topology information such as the number of cores and PUs and the available memory is obtained (lines 6-11) after creating and initializing an instance of the *HwlocTopology* class (lines 1-4), which represents an abstraction of the underlying hardware. Most of the `jhwloc` functionality is provided through this class with more than 50 Java methods available that allow to manipulate, traverse and browse the topology, as well as to perform CPU and memory binding operations. Next, the example shows how to manage CPU affinities by binding the current thread to the CPU set formed by the first and last PU of the machine. As can be seen, the Java objects that represent those PUs, which are instances of the *HwlocObject* class, can be easily obtained by using their indexes (lines 13-15). The CPU set objects from both PUs are then operated using a logical `and` (lines 16-17), and the returned CPU set is used to perform the actual CPU binding of the current thread (lines 18-19). The logical `and` operation is an example of the more than 30 Java methods that are provided to conform with the hwloc bitmap API, supported in `jhwloc` through the *HwlocBitmap* abstract class. This class is extended by the *HwlocCPUSet* and *HwlocNodeSet* subclasses that provide concrete implementations for representing CPU and NUMA node sets, respectively.

More advanced usage examples are provided together with the `jhwloc` source code. These examples include NUMA binding, manipulating bitmaps, obtaining

```
// Create, initialize and load a topology object
HwlocTopology topo = new HwlocTopology();
topo.init();
topo.load();

// Get the number of cores, PUs, and the available memory
int nc = topo.get_nbobjs_by_type(HWLOC.OBJ_CORE);
int np = topo.get_nbobjs_by_type(HWLOC.OBJ_PU);
long mem = topo.get_root_obj().getTotalMemory();
System.out.println(‘‘#Cores/PUs: ’’+nc+‘‘/’’+np);
System.out.println(‘‘Total memory: ’’+mem);

// Get first and last PU objects
HwlocObject fpu = topo.get_obj_by_type(HWLOC.OBJ_PU, 0);
HwlocObject lpu = topo.get_obj_by_type(HWLOC.OBJ_PU, np-1);
// Logical ‘and’ over the CPU sets of both PUs
HwlocCPUSet cpuset = fpu.getCPUSet().and(lpu.getCPUSet());
// Bind current thread to the returned CPU set
topo.set_cpubind(cpuset, EnumSet.of(HWLOC.CPUBIND_THREAD);
```

Listing 1: Getting hardware topology information and managing CPU affinities

cache information, traversing topologies, exporting topologies to XML and building synthetic ones, among other jhwloc functionalities.

4 Impact of CPU Affinity on Performance: Flame-MR Case Study

This section analyzes the impact of setting CPU affinity on the performance of Flame-MR, our Big Data processing framework. First, Flame-MR is briefly introduced in Section 4.1. Next, Section 4.2 describes how jhwloc has been integrated into Flame-MR to manage CPU affinities, explaining the different affinity levels that are supported. Section 4.3 details the experimental testbed, and finally Section 4.4 discusses the results obtained.

4.1 Flame-MR Overview

Flame-MR [22] is a Java-based MapReduce implementation that transparently accelerates Hadoop applications without modifying their source code. To do so, Flame-MR replaces the underlying data processing engine of Hadoop by an optimized, in-memory architecture that leverages system resources more efficiently.

The overall architecture of Flame-MR is based on the deployment of several *Worker* processes (i.e., JVMs) over the nodes of a cluster (see Figure 2). Relying on an event-driven architecture, each *Worker* is in charge of executing

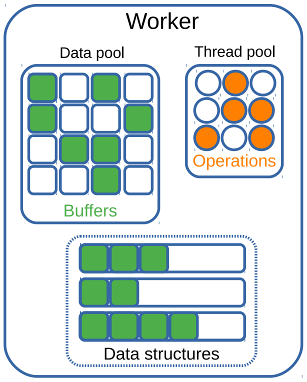


Fig. 2: Flame-MR *Worker* architecture

the computational tasks (i.e., map/reduce operations) to process the input data from HDFS. The tasks are executed by a thread pool that can perform as many concurrent operations as the number of cores configured for each *Worker*. These operations are efficiently scheduled in order to pipeline data processing and data movement steps. The data pool allocates memory buffers in an optimized way, reducing the amount of buffer creations to minimize garbage collection overheads. Once the buffers are filled with data, they are stored into in-memory data structures to be processed by subsequent operations. Furthermore, these data structures can be cached in memory between MapReduce jobs to avoid writing intermediate results to HDFS, thus providing efficient iterative computations.

4.2 Managing CPU Affinities in Flame-MR

Flame-MR has been extended to use the functionalities provided by `jhwloc` to enable the binding of computational tasks to the hardware processing elements available in the system (i.e., CPU/cores/PUs). To do so, the software components that manage such tasks, *Worker* and thread pool classes, have been modified to make them aware of the hardware affinity level that is set by the user through the configuration file of Flame-MR.

When Flame-MR starts a computational task, it first checks the configuration file to determine if the `jhwloc` library must be called and, if so, the specific affinity level that must be enforced. Then, the *Worker* initializes a *HwlocTopology* object and uses the `set_cpubind` method provided by `jhwloc` to bind computational tasks to the appropriate hardware. The configuration of the *Workers* is affected by the affinity level being used, as the number of threads launched by them to execute map/reduce operations should be adapted to the hardware characteristics of the underlying system. The intuitive recommendation would be to create as many *Workers* as CPUs, and as many threads per *Worker* as cores/PUs available in the nodes. The CPU affinity levels currently supported by Flame-MR through `jhwloc` are:

- NONE: Flame-MR does not manage hardware affinities in any way (i.e., `jhwloc` is not used).
- CPU: *Workers* are bound to specific CPUs. Each JVM process that executes a *Worker* is bound to one of the available CPUs in the system using the `jhwloc` flag *HWLOC.CPUBIND_PROCESS*. Hence, each thread launched by a *Worker* is also bound to the same CPU, which means that the OS scheduler can migrate threads among its cores. The mapping between *Workers* and CPUs is done cyclically by allocating each *Worker* to a different CPU until all the CPUs are used, starting again if there are remaining *Workers*.
- CORE: map/reduce operations are bound to specific cores. Each thread launched by a *Worker* to perform such operations is bound to one of the available cores in the system using the flag *HWLOC.CPUBIND_THREAD*. So, the OS scheduler can migrate threads among the PUs of a core (if any). The mapping between threads and cores is done by allocating a group of cores from the same CPU to each *Worker*. Note that the number of threads used by all *Workers* executed in a node should not exceed the number of cores to avoid resource oversubscription.
- PU: map/reduce operations are bound to specific PUs. Each thread launched by a *Worker* to perform such operations is bound to one of the available PUs in the system using the flag *HWLOC.CPUBIND_THREAD*. The mapping between threads and PUs is done by allocating a group of cores to each *Worker*, and then distributing its threads over the PUs of those cores in a cyclic way. Note also that the number of threads used by all *Workers* on a node should not exceed the number of PUs to avoid oversubscription.

It is important to note that all the threads launched by a *Worker* process are created during the Flame-MR start-up phase. So, `jhwloc` is only accessed once to set the affinity level, avoiding any JNI overhead during data processing.

4.3 Experimental Configuration

Six MapReduce workloads from four domains that represent different Big Data use cases have been evaluated: (1) data sorting (Sort), (2) machine learning (K-Means), (3) graph processing (PageRank, Connected Components), and (4) genome sequence analysis (MarDRe, CloudRS). Sort is an I/O-bound microbenchmark that sorts an input text dataset generated randomly. K-Means is an iterative clustering algorithm that classifies an input set of N samples into K clusters. PageRank and Connected Components are popular iterative algorithms for graph processing. PageRank obtains a ranking of the elements of a graph taking into account the number and quality of the links to each one, and Connected Components explores a graph to determine its subnets. MarDRe [8] and CloudRS [3] are bioinformatics tools for preprocessing genomics datasets. MarDRe removes duplicate and near-duplicate DNA reads, whereas CloudRS performs read error correction.

In the experiments conducted in this paper, Sort processes a 100 GB dataset and K-Means performs a maximum of five iterations over a 35 GB dataset ($N =$

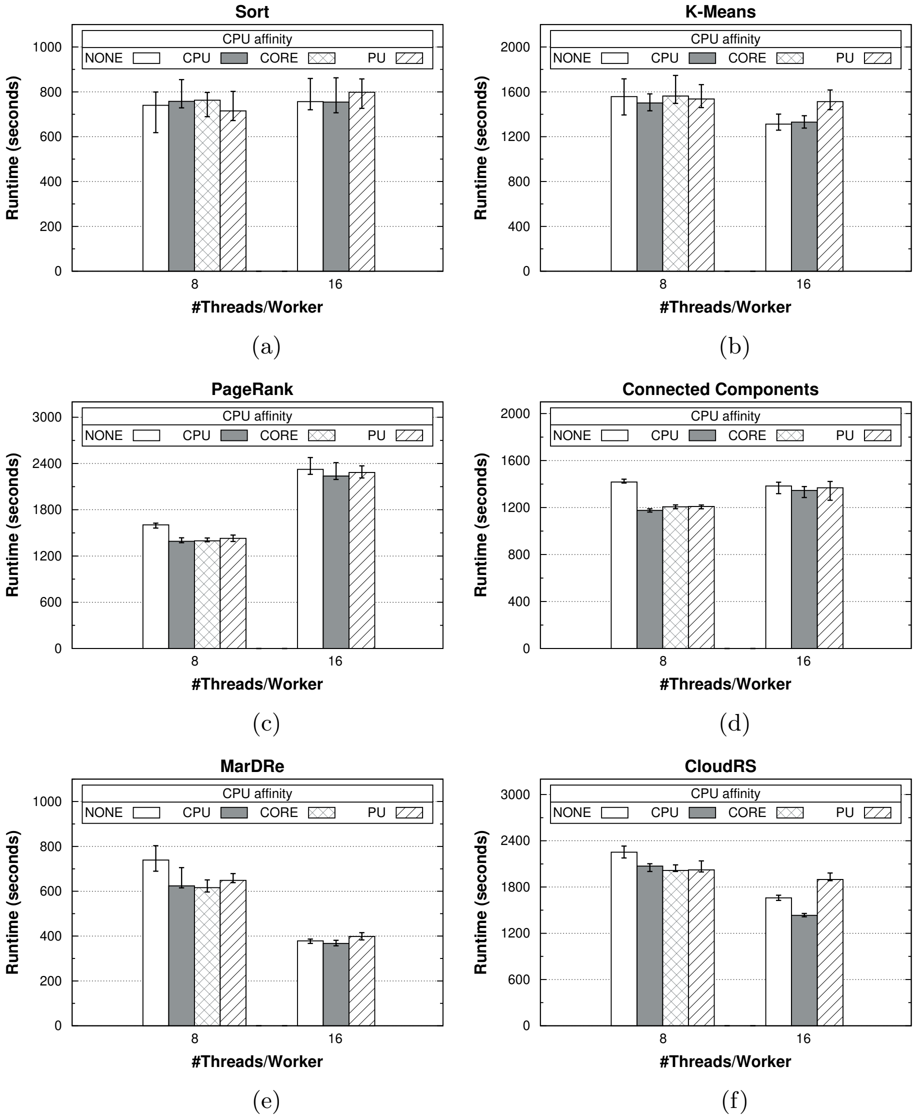
360 million samples) using 200 clusters ($K = 200$). Both PageRank and Connected Components execute five iterations over a 40 GB dataset (60 million pages). MarDRe removes duplicate reads using the SRR377645 dataset, named after its accession number in the European Nucleotide Archive (ENA) [12], which contains 241 million reads of 100 base pairs each (67 GB in total). CloudRS corrects read errors using the SRR921890 dataset, which contains 16 million reads of 100 base pairs each (5.2 GB in total). Both genomics datasets are publicly available to download at ENA website.

The experiments have been carried out on a 9-node cluster with one master and eight slave nodes running Flame-MR version 1.2. Each node consists of a NUMA system with two Intel Xeon E5-2660 octa-core CPUs. This CPU model features two-way Intel HT, so 8 cores and 16 logical PUs are available per CPU (i.e., 16 and 32 per node, respectively). Each node has a total of 64 GiB of memory evenly distributed between the two CPUs. The NUMA architecture just described is the one previously shown in Figure 1, which also details the cache hierarchy. Additionally, each node has one local disk of 800 GiB intended for both HDFS and intermediate data storage during the execution of the workloads. Nodes are interconnected through Gigabit Ethernet (1 Gbps) and InfiniBand FDR (56 Gbps). The cluster runs Linux CentOS 6.10 with kernel release 2.6.32-754.3.5, whereas the JVM version is Oracle JDK 10.0.1.

To deploy and configure Flame-MR on the cluster, the Big Data Evaluator (BDEv) tool [21] has been used for ease of setup. Two *Workers* (i.e., two JVM processes) have been executed per slave node, since our preliminary experiments proved it to be the best configuration for Flame-MR on this system. Regarding the number of threads per *Worker*, two different configurations have been evaluated: (1) using as many threads per *Worker* as cores per CPU (8 threads), and (2) using as many threads per *Worker* as PUs per CPU (16 threads), thus also evaluating the impact of Intel HT on performance. Finally, the metric shown in the following graphs corresponds to the median runtime for a set of 10 executions for each experiment, clearing the OS buffer cache of the nodes between each execution. Variability is represented in the graphs by using error bars to indicate the minimum and maximum runtimes.

4.4 Performance Results

Figure 3 presents the measured runtimes of Flame-MR for all the workloads when using different affinity levels and *Worker* configurations as previously described. When running 8 threads per *Worker* (i.e., not using Intel HT), all the workloads benefit from enforcing some hardware affinity, although the best level to use varies for each workload. On the one hand, the performance improvements for Sort (Figure 3a) and K-Means (Figure 3b) with respect to the baseline scenario (i.e., without using affinity) are lower than for the remaining workloads. The main reason is that both workloads are the most I/O-intensive codes under evaluation, being clearly bottlenecked by disk performance (only one disk per slave node is available). This fact limits the potential benefits of using an enforced hardware placement. Nevertheless, the improvements obtained are up



(a)

(b)

(c)

(d)

(e)

(f)

Fig. 3: Runtimes of Flame-MR for different workloads and CPU affinity levels

to 3% and 4% using the PU and CPU affinity levels for Sort and K-Means, respectively. On the other hand, the performance improvements for the remaining workloads (see Figures 3c-3f) are significantly higher: up to 13%, 17%, 16% and 11% for PageRank, Connected Components, MarDRe and CloudRS, respectively. Although the best affinity level depends on each particular workload (e.g., CPU for PageRank, CORE for CloudRS), it can be concluded that the performance differences between different levels are generally small. Note that the workloads benefit not only from accelerating a single *Worker* (i.e., JVM) using CPU binding, but also from reducing the impact of the synchronizations among all *Workers*, performed between the global map and reduce phases for each MapReduce job. Moreover, this reduction in the runtimes is obtained as a zero-effort transparent configuration for Flame-MR users, without recompiling/modifying the source code of the workloads.

Regarding the impact of Intel HT on performance (i.e., running 16 threads per *Worker*), note that the results for the CORE affinity level cannot be shown when using two *Workers* per node since slave nodes have 16 physical cores, as mentioned in Section 4.3. In the HT scenario, K-Means, MarDRe and CloudRS take clear advantage of this technology. In the case of K-Means (see Figure 3b), execution times are reduced with respect to the non-HT counterparts by 16% and 12% for the baseline and CPU affinity scenarios, respectively. The improvements for these two scenarios increase up to 26% and 32% for CloudRS (Figure 3f), and 48% and 41% for MarDRe (Figure 3e). However, the impact of HT can be considered negligible for Sort, as shown in Figure 3a, whereas the performance of PageRank and Connected Components is even reduced in most scenarios. Note that using HT technology implies that the two logical PUs within a physical core must share not only all levels of the cache hierarchy, but also some of the computational units. This fact can degrade the performance of CPU-bound workloads due to increased cache miss rates and resource contention, as it seems to be the case with PageRank. Finally, the impact of enforcing hardware affinity when using HT can only be clearly appreciated for CloudRS, providing a reduction in the execution time of 14% when using the CPU affinity level.

We can conclude that there is no one-size-fits-all solution, since the best affinity level depends on each workload and its particular resource characterization. Furthermore, the impact of managing CPU affinities is clearly much more beneficial when HT is not used. However, the results shown in this section reinforce the utility and performance benefits of managing hardware affinities in Big Data JVM-based frameworks such as Flame-MR.

5 Conclusions

The complexity of current computing infrastructures raises the need for carefully placing applications on them so that affinities can be efficiently exploited by the hardware. However, the standard class library provided by the JVM lacks support for developing hardware-aware applications, preventing them from taking advantage of managing CPU or NUMA affinities. As most popular distributed processing frameworks are implemented using languages executed by the JVM, having such affinity support can be of great interest for the Big Data community.

In this paper we have introduced `jhwloc`, a Java library that exposes an object-oriented API that allows developers to gather information about the underlying hardware and bind tasks according to it. Acting as a wrapper between the JVM and the C-based hwloc library, the de facto standard in HPC environments, jhwloc enables JVM-based applications to easily manage affinities to perform hardware-aware optimizations. Furthermore, we have extended our Java MapReduce framework Flame-MR to include support for setting hardware affinities through jhwloc, as a case study to demonstrate the potential benefits. The experimental results, running six representative Big Data workloads on a 9-node cluster, have shown that performance can be transparently improved by up to 17%. Other popular JVM-based Big Data frameworks such as Hadoop, Spark, Flink, Storm and Samza could also take advantage of the features provided by jhwloc in a similar way to Flame-MR.

The source code of the jhwloc library is released under the open-source GNU GPLv3 license and is publicly available together with the Javadoc documentation at https://github.com/rreye/jhwloc. As future work, we aim to explore the impact of setting NUMA affinities on JVM performance when using different garbage collection algorithms. We also plan to extend jhwloc to provide other functionalities such as gathering information about the network topology.

Acknowledgments

This work was supported by the Ministry of Economy, Industry and Competitiveness of Spain and FEDER funds of the European Union [ref. TIN2016-75845-P (AEI/FEDER/EU)]; and by Xunta de Galicia and FEDER funds [Centro de Investigación de Galicia accreditation 2019-2022, ref. ED431G2019/01, Consolidation Program of Competitive Reference Groups, ref. ED431C2017/04].

References

1. Awan, A.J., Vlassov, V., Brorsson, M., Ayguade, E.: Node architecture implications for in-memory data analytics on scale-in clusters. In: Proc. 3rd IEEE/ACM Int. Conference on Big Data Computing, Applications and Technologies (BDCAT 2016). pp. 237–246. Shanghai, China (2016)
2. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink: stream and batch processing in a single engine. Bulletin of the IEEE Technical Committee on Data Engineering **38**(4), 28–38 (2015)
3. Chen, C.C., Chang, Y.J., Chung, W.C., Lee, D.T., Ho, J.M.: CloudRS: an error correction algorithm of high-throughput sequencing data based on scalable framework. In: Proc. IEEE Int. Conference on Big Data (IEEE BigData 2013). pp. 717–722. Santa Clara, CA, USA (2013)
4. Chiba, T., Onodera, T.: Workload characterization and optimization of TPC-H queries on Apache Spark. In: Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS 2016). pp. 112–121. Uppsala, Sweden (2016)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
6. Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M.: Simultaneous multithreading: a platform for next-generation processors. IEEE Micro **17**(5), 12–19 (1997)

7. Ekanayake, J., et al.: Twister: a runtime for iterative MapReduce. In: Proc. 19th ACM Int. Symposium on High Performance Distributed Computing (HPDC'10). pp. 810–818. Chicago, IL, USA (2010)
8. Expósito, R.R., Veiga, J., González-Domínguez, J., Touriño, J.: MarDRe: efficient MapReduce-based removal of duplicate DNA reads in the cloud. Bioinformatics **33**(17), 2762–2764 (2017)
9. Goglin, B.: Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In: Proc. Int. Conference on High Performance Computing & Simulation (HPCS 2014). pp. 74–81. Bologna, Italy (2014)
10. Iqbal, M.H., Soomro, T.R.: Big Data analysis: Apache Storm perspective. International Journal of Computer Trends and Technology **19**(1), 9–14 (2015)
11. Lameter, C.: NUMA (Non-Uniform Memory Access): an overview. ACM Queue **11**(7), 40:40–40:51 (2013)
12. Leinonen, R., et al.: The European Nucleotide Archive. Nucleic Acids Research **39**, D28–D31 (2011)
13. Marr, D.T., et al.: Hyper-Threading technology architecture and microarchitecture. Intel Technology Journal **6**(1), 1–12 (2002)
14. Noghabi, S.A., et al.: Samza: stateful scalable stream processing at LinkedIn. Proceedings of the VLDB Endowment **10**(12), 1634–1645 (2017)
15. Peternier, A., Bonetta, D., Binder, W., Pautasso, C.: Tool demonstration: Overseer – low-level hardware monitoring and management for Java. In: Proc. 9th Int. Conference on Principles and Practice of Programming in Java (PPPJ 2011). pp. 143–146. Kongens Lyngby, Denmark (2011)
16. Wasi-ur Rahman, M., et al.: High-performance RDMA-based design of Hadoop MapReduce over InfiniBand. In: Proc. IEEE 27th Int. Symposium on Parallel & Distributed Processing, Workshops & Phd Forum (IPDPSW 2013). pp. 1908–1917. Boston, MA, USA (2013)
17. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Proc. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010). pp. 1–10. Incline Village, NV, USA (2010)
18. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: Proc. Workshop on Memory Access on Future Processors: A Solved Problem? (MAW'08). pp. 377–384. Ischia, Italy (2008)
19. The Apache Hadoop project: http://hadoop.apache.org, [cited 31 March 2020]
20. Vavilapalli, V.K., et al.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: Proc. 4th Annual Symposium on Cloud Computing (SOCC'13). pp. 5:1–5:16. Santa Clara, CA, USA (2013)
21. Veiga, J., Enes, J., Expósito, R.R., Touriño, J.: BDEv 3.0: energy efficiency and microarchitectural characterization of Big Data processing frameworks. Future Generation Computer Systems **86**, 565–581 (2018)
22. Veiga, J., Expósito, R.R., Taboada, G.L., Touriño, J.: Flame-MR: an event-driven architecture for MapReduce applications. Future Generation Computer Systems **65**, 46–56 (2016)
23. Wang, L., Ren, R., Zhan, J., Jia, Z.: Characterization and architectural implications of Big Data workloads. In: Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS 2016). pp. 145–146. Uppsala, Sweden (2016)
24. Zaharia, M., et al.: Apache Spark: a unified engine for Big Data processing. Communications of the ACM **59**(11), 56–65 (2016)