Project Number 318763

D2.2 – Architecture Patterns Specification

Version 1.0
26 January 2014
Final

Public Distribution

aicas, University of York, SOFTEAM
Scuola Superiore Sant'Anna, University of Stuttgart

Project Partners: **aicas**, **HMI**, **petaFuel**, **SOFTEAM**, **Scuola Superiore Sant'Anna**, **The Open Group**, **University of Stuttgart**, **University of York**

Project Partner Contact Information

aicas Fridtjof Siebert Haid-und-Neue Strasse 18 76131 Karlsruhe Germany Tel: +49 721 66396823 E-mail: siebert@aicas.com	**HMI** Markus Schneider Im Breitspiel 11 C 69126 Heidelberg Germany Tel: +49 6221 7260 0 E-mail: schneider@hmi-tec.com
petaFuel Ludwig Adam Muenchnerstrasse 4 85354 Freising Germany Tel: +49 8161 40 60 202 E-mail: ludwig.adam@petafuel.de	**SOFTEAM** Andrey Sadovykh Avenue Victor Hugo 21 75016 Paris France Tel: +33 1 3012 1857 E-mail: andrey.sadovykh@softeam.fr
Scuola Superiore Sant'Anna Mauro Marinoni via Moruzzi 1 56124 Pisa Italy Tel: +39 050 882039 E-mail: m.marinoni@sssup.it	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Stuttgart Bastian Koller Nobelstrasse 19 70569 Stuttgart Germany Tel: +49 711 68565891 E-mail: koller@hlrs.de	**University of York** Neil Audsley Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325571 E-mail: neil.audsley@cs.york.ac.uk

Contents

List of Figures

Document Control

Version	Status	Date
0.1	Document outline	9 January 2014
0.8	Complete First Draft	16 January 2014
1.0	QA for EC delivery	26 January 2014

Executive Summary

This document constitutes deliverable *D2.2 – Architecture Patterns Specification* of WP2 of the JUNIPER project. The purpose of this deliverable is to describe the way in which the platform-level programming model of the JUNIPER project aids the developer to better exploit the target hardware.

In big data systems, the developer will frequently develop software without a precise notion of the architecture upon which it will be deployed. In a managed cloud environment, the available resources may change rapidly, even between invocations of the software. In order to obtain high performance, the developed software must respond to these changes accordingly. The JUNIPER project provides an API to assist with architecture discovery. This API characterises the host architecture in terms of accepted patterns (SMP, NUMA etc.) and assists the development of reactionary software to exploit the hardware.

The deliverable will commence by describing the problems experienced by big data programmers when working with Java and some of the existing solutions in Section 1. Section 2 describes how the upcoming Java 8 addresses some of these problems but not all of them. Section 3 introduces architectural patterns, locales and stored collections which comprise the JUNIPER solution to the identified issues.

In order to avoid repeating content, this document provides a rationale and description of the architectural discovery features of the JUNIPER API. The API in full is detailed in deliverable D2.1.

1 Background

When building software for big data systems, their unique architectures result in interesting challenges. The JUNIPER project identifies two main levels at which issues are observed: the cluster level and the node level.

At the cluster level, big data systems are deployed 'in the cloud'. This means that they execute across a network of general-purpose, multicore computers. As a result, the programmer must consider:

- How to parallelise their application such that it can be distributed. This includes splitting and distribution of the input and output data of the application.
- Data and code locality. It is necessary to ensure that data is physically located close to the site which is executing the software that processes it to reduce the amount of required communication.
- Storage must also be located through the cloud such that communication is minimised.
- Communications are slow and unreliable and yet application performance and integrity must be maintained.

At the node level, the big data problem has been decomposed into a "normal data" problem. The problems at this level are:

- The architecture of any specific compute platform in the cloud is not known ahead of time. Programs developed for large-scale distributed applications must respond to the fact that new nodes may be added to or removed from the target compute cloud at any time.
- Further to this, many of the target nodes are virtual platforms and so the resources they provide may vary very frequently.
- Memory layouts on such systems may vary from standard SMP-style platforms to complex supercomputing platforms with non-uniform shared memory. This must be considered and the program should react accordingly.
- The use of on-disk storage must be efficient. Data will be distributed across the nodes from the cluster level, and the software running on each node must read and write that data in a way which minimises overheads.

The JUNIPER project considers cluster-level concerns in its programming model, as described in deliverable D2.1. This deliverable concentrates on the aspects of the programming model which are designed to assist with node-level issues. Specifically, the JUNIPER programming model provides the programmer with portable architectural discovery mechanisms. These assist the development of software which can observe the environment in which it is executing and react accordingly.

1.1 Current issues

The input language in the JUNIPER project is Java conforming to the Real-Time Specification for Java (RTSJ) [2]. This is an extension for Java which augments the Java programming model to include (amongst other features) real-time tasks and schedulers and a complex memory model. However currently there are a number of missing features that are required in the applications that the JUNIPER approach is tackling:

- Flexible parallelism (Section 1.1.1)
- Locality (Section 1.1.2)
- Hardware architecture discovery (Section 1.1.3)

These are described in more detail in the following sections.

1.1.1 Flexible parallelism

The RTSJ provides parallelism using *schedulable objects*. These are the objects that the language's base scheduler manages, and are commonly called 'threads' or 'tasks' in other languages. The two classes that implement schedulable objects are:

- `RealtimeThread` is an extension of `java.lang.Thread` which adds real-time services such as asynchronous transfer of control, the ability to allocate in non-heap memory, and advanced scheduler services (such as to include priorities, customizable scheduling schemes, perform scheduling feasibility tests, etc.).
- `AsyncEventHandler` is an encapsulation of code to be executed after an asynchronous event (represented by an instance of `AsyncEvent`) is triggered. These are commonly used to implement interrupt handlers in response to an external event.

The above allows the programmer to express task parallelism as Java-style threading through the use of `RealtimeThread`, and also event-driven programming with `AsyncEventHandlers`. However, in both cases these are very coarse-grained types of parallelism. Schedulable objects are relatively long-lived entities which are mapped on to JVM or OS-level threads for execution.

Another useful form of parallelism, especially common in the systems considered by JUNIPER, is *data parallelism* [3]. Data parallelism describes how the same operation can be applied to independent data items. Figure 1 illustrates the two kinds of parallelism. As can be seen, task parallelism uses explicit thread creation and patterns such as fork and join. Data parallelism (shown as `parfor` in the figure) adds to this by allowing each thread to further parallelise their execution with parallel-for loops or other SIMD (single-instruction, multiple-data) operations. For example, multiplying a vector by a constant factor is a data parallel operation. A naïve implementation could create a thread for each element of the array, but this would be very inefficient. Instead the programmer would prefer to express that this parallelism opportunity exists and allow the infrastructure of the language to implement it appropriately.

There are a wide range of existing systems which allow this kind of programming. Some of the most well-known are the languages that allow data parallel programming on CPUs and GPUs; e.g. OpenMP [4], CUDA [5] and OpenCL [6]. The languages provide programming constructs such as the parallel-for, which is a for loop in which each iteration may be performed in parallel and the frameworks handle the decomposition and collection of data. They also transparently handle the creation and synchronisation of the worker threads required to execute its code. Java does not provide such features although Java 7 did introduce the fork-join framework [7] which provides a way of decomposing large amounts of work into concurrently executing sub-jobs, the results of which are automatically collected together.

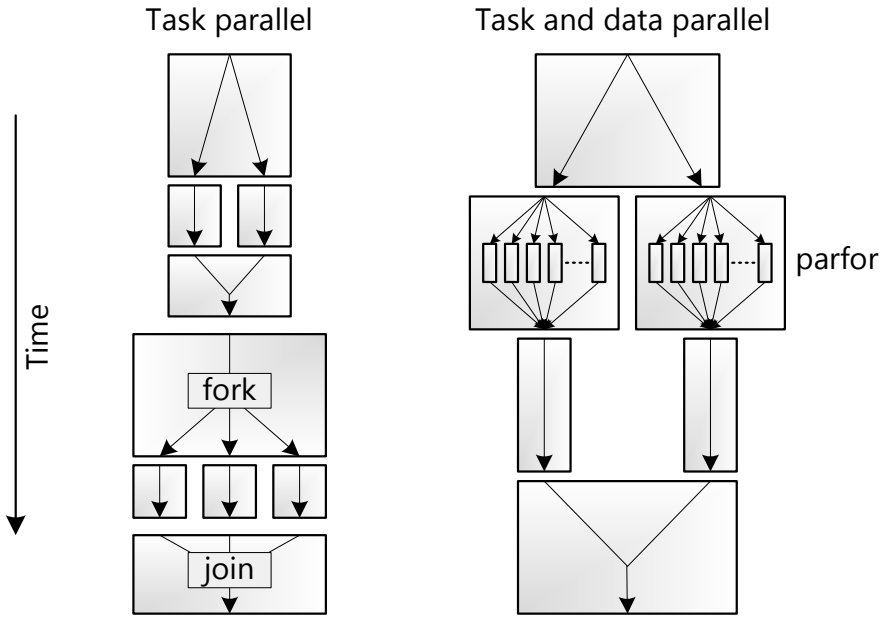

Figure 1: Task parallelism (left), task and data parallelism (right)

Currently to obtain true data parallel programming the Java programmer must rely on tools such as JOMP [8], Jconqurr [9] or Pyjama [10] which provide OpenMP-like directives, or JaMP [11] which is a full Java implementation of OpenMP 2.0. Such approaches are all outside of the scope of the Java language and do not support the use of the RTSJ which limits their potential for use in a real-time context.

This issue is being addressed by the Java language in the new Java 8 standard which is, at time of writing, due to be released in March 2014 (but with the final release candidate ready in Janurary 2014) [12]. Section 2 discusses how Java 8 will include data parallel operations.

1.1.2 Locality

The Java programming model implicitly assumes a single global memory space in which all code and data (including the heap and thread stacks) exist. On many modern architectures this assumption is only made valid through many layers of virtualisation (MMUs, virtual memory, paging, shared caches, distributed file systems etc.). For example, in a multicore system there are many layers of cache. A level of cache may be shared between some processors but not others. In this case, the programmer will want to be able to express that threads and data that are solving the same problem should be kept in processors with a shared cache, rather than processors without a shared cache. An example of this scenario is illustrated in Figure 2, in which tightly coupled threads should be allocated to processor pairs A and B or C and D to maximise cache reuse.

Similarly, in larger scale systems like supercomputers, whilst the combination of high-speed interconnect (i.e. InfiniBand [13]) and middleware (i.e. Lustre [14]) can provide the illusion of a single massive system, the programmer must still consider the locality of their threads and data in order to attain the highest performance.

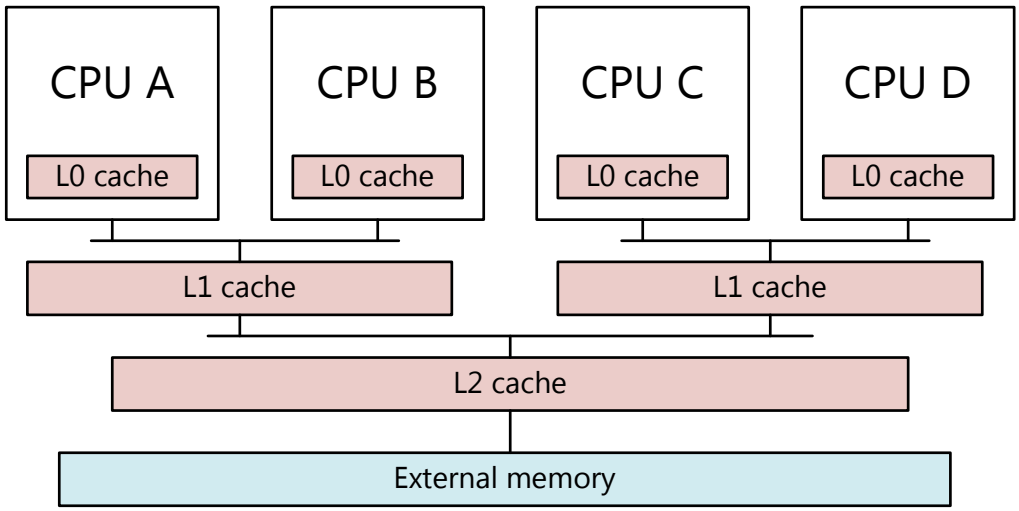

Figure 2: Multi-level caches affect optimal thead allocation.

Locality is not supported in the standard Java model, because it is a language for a virtual machine that does not display locality issues. The only way to bind a thread to a given execution site was to use JNI to call a C language library that makes use of the Linux API function `sched_setaffinity` (with similar techniques available for other OSes). This was addressed in the RTSJ with the addition of the `Affinity` class which allows access to the set of processors upon which a thread or event handler may execute. It also provides APIs to be informed of changes in the number of CPUs available for the JVM.

Unfortunately, all of these examples merely set CPU affinity for a thread. They do not address the problem of binding *sets* of threads to the same processor, and they do not address the locality of data items at all. Frequently, in a big data system the location of data is more important than the location of the thread.

To address this, the JUNIPER project is adding support for *locales*, which are a language-level construct for binding groups of threads and data together. This is discussed in Section 3.1.

1.1.3 Hardware architecture discovery

The ability to control parallelism and locality is only useful if the programmer can also obtain a model of the target architecture that allows their software to make decisions on how to effectively exploit it. Java currently only allows very basic hardware information to be obtained such as the total amount of available memory (`Runtime.totalMemory()`) or the number of processors (`Runtime.availableProcessors()`). Clearly the memory information presumes a single globally-shared memory.

The RTSJ again extends this model to provide a better correspondence with real hardware by explicitly modelling memory regions that may be of difference sizes and speeds (using `RawMemory`, `ImmortalMemory`, etc.). An upcoming future release will also model hardware interrupts to allow better interaction with devices. However it is still explicitly targeting a homogenous SMP system in which all communications are presumed equal.

Outside of Java, the Portable Hardware Locality package (hwloc) [15] provides a portable abstraction of the hierarchical topology of modern architectures. It includes NUMA memory, (shared) caches, heterogeneous cores, I/O devices, communications infrastructure (such as InfiniBand) and accelerators. Unfortunately hwloc does not have a Java binding and the few projects that are attempting this are in their infancy [16].

The JUNIPER project provides an architecture discovery model based around the idea of architectural patterns to solve these problems. This is discussed in Section 3.

2 Java 8 Additions for Big Data

This section describes two upcoming additions to Java that simplify parallel programming: lambda expressions and streams.

2.1 Lambdas

Lambda expressions, or lambdas, are the most visible addition to Java 8. They provide a concise way to express functional programming concepts in Java [17]. A lambda can be specified in place of a value whose type is a functional interface (an interface with exactly one abstract method). With a suitable library, programmers can parallelise their code by rewriting certain sections to use lambdas, allowing the library to execute them in parallel. For example, a sequential loop can be written with a library-defined `forEach` method that executes a given lambda in parallel as:

```
shapes.forEach(s -> { s.type = CIRCLE; });
```

Lambdas simply parallel programming for two reasons:

- Their limited interface encourages a functional style of programming, in which data dependencies are minimised and work is decomposed into functional units. This maximises the available parallelism in a given work load and aids the creation of data parallel streaming pipelines.
- They have a much more compact syntax than Java's original approach which used anonymous inner classes. This makes programming clearer and more concise. The other parallel programming additions to Java 8 do not *require* lambdas, but they make them more usable.

2.2 Streams

Java 8 also introduces the concept of a *stream*, which is a sequence of elements that can be operated on. Streams can be generated from collections by calling the `stream` method if the desired stream should be sequential, or the `parallelStream` method for a parallel stream.

After retrieving the stream object, a pipeline of bulk data operations [18] can be performed on it, consisting of zero or more intermediate operations followed by a terminal operation. Conceptually, an intermediate operation returns a new stream with the resulting elements, and a terminal operation returns a result that is not a stream. Since the pipeline is evaluated lazily, only enough elements are consumed as required by the terminal operation.

Bulk data operations make use of lambdas to enable parallelism. For example, `map` is a common functional programming idiom that applies a function to every element of a set. In Java 8 `map` is provided by the standard libraries. It operates on a stream, and takes a lambda as the operation to apply to every element of the stream. If the stream is a parallel stream, a system-provided fork-join task pool will execute this lambda in parallel.

Consider the following example:

```
List<Person> persons = ...;

persons.stream()
  .filter(p -> p.getAge() > 18)
  .map(p -> p.setOver18())
  .forEach(Student::print);
```

In this simple example, a `List` of `Person` objects called `persons` is created. This list can be processed as a stream by calling the list's `stream()` method, which returns a `Stream<Person>`. The `filter` operation is applied to this stream which takes a lambda that, for each element of the stream, evaluates a predicate and only passes the element through to the next part of the stream if the predicate is `true`. In this example, only `Person` objects where `Person.getAge()` is greater than 18 are passed through to the later stages of the pipeline. After the `filter` a `map` is applied which calls the `setOver18` method on each object. This is provided as a lambda. For reference, without lambdas the `map` would have to look like this:

```
.map(new Function<Person, Student>() {
    @Override
    public Student apply(Person person) {
      Person p = new Person(person);
      p.setOver18();
      return p;
    }
})
```

Finally, `forEach` is called. Whereas `map` and `filter` are *intermediate* pipeline stages (they take a stream and return a stream), `forEach` is a *terminal*, in that it takes a stream but does not return anything, therefore ending the pipeline. `forEach` simply iterates over the entire stream and applies a lambda to each element. In this case a *method reference* is used to refer to the method `Student.print()`. This is a Java 8 feature added as part of lambdas that allows references to point to methods as well as objects.

Streams are *lazy*. This means that no computation is performed until it is directly requested. Consider the following example:

```
Stream<String> filteredStream = arrayOfStrings.stream()
  .filter(s -> s.length() >= 4);
```

This uses `filter` to filter out any `Strings` in the stream which have a length less than 4. However, the filtering is only actually performed when elements are pulled from `filteredStream` by a terminal operator. Streams may be infinite in length, but only the elements requested by the terminal operator will cause computation. To execute the pipeline a terminal like `forEach` or `count` (which counts the number of elements in a stream) may be used.

```
int count = arrayOfStrings.stream()
  .filter(s -> s.length() >= 4)
  .count();
```

2.3 Spliterators

For collection-backed streams, an iterator is sufficient for traversing sequential streams, but in order to operate on parallel streams a *spliterator* is needed. Spliterators support both traversal and partitioning of elements. A spliterator implementation may support efficient bulk traversal with the `forEachRemaining` method, and determine where the next partition occurs, if at all, in the `trySplit` method. If partitioning is possible, the `trySplit` method modifies the current spliterator and returns a new one such that the elements that will be traversed by the new spliterator will not be traversed by the current one.

The discussed Java 8 features address the parallelism requirements of data-driven systems so they are adopted in the JUNIPER project. Section 3 will discuss other features which are required and how they are implemented in the project.

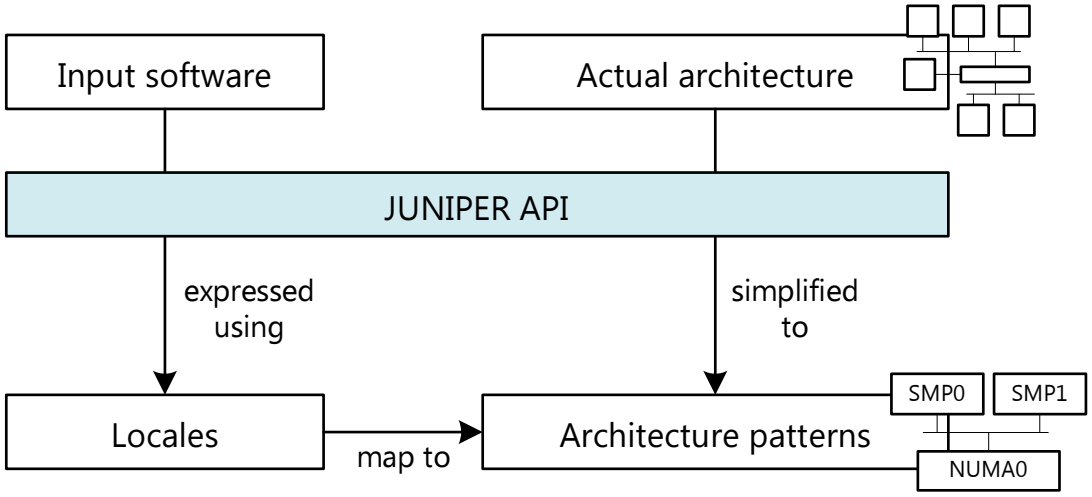

Figure 3: Architecture patterns used to assist software development

3 Architectural Patterns

Section 1 described some of the problems encountered when using Java (or the RTSJ) when targeting the platforms of big data systems. Primarily:

- Lack of flexible parallelism
- Lack of locality
- Difficulty of developing software that can adapt to changing host architectures

The parallelism requirement is covered by the JUNIPER project adopting Java 8 (Section 2), the second two points have to be addressed through the use of a JUNIPER API that provides *locality* and *architectural patterns*.

The JUNIPER project proposes the development of an infrastructure in which the programmer can manage the locality of the code and data of their system. The programmer can use the infrastructure to dynamically discover the host architecture of their software and respond accordingly by mapping the locality of their software to sets of known architectural patterns. The process is shown in Figure 3.

The programmer calls the JUNIPER API to determine the layout of their host hardware, which may be highly dynamic in a cloud environment. They can then use *locales* (Section 3.1) to tailor their application to the target by binding threads and data accordingly.

3.1 Locales to Enable Locality-Aware Mapping

The JUNIPER API adds the concept of a *locale*. A locale is used to describe the locality of elements of the software, and has the following properties:

- The threads and objects encapsulated in a locale are mapped by the JVM onto the CPUs and memories of a single architecture pattern. (Architecture patterns are discussed in the following Section.) They will not be ‘split up’ across the whole platform.

- A locale is given a resource reservation that is the result of a negotiation between the JVM and the host operating system. This document does not discuss reservations, for more information see deliverable D3.1.
- The heap, immortal and backing store memory allocated to a locale are not allocated to any other locale. However, references between locales are legal within the rules defined by the Real-Time Specification for Java (RTSJ) [2].
- Garbage collection within a locale's heap must obey the boundaries of the locale. That is, if the garbage collector supports a copying phase, no object shall be copied from the heap in one locale to the heap in another locale.

The approach taken is to provide factory methods to create threads (including real-time threads and asynchronous event handlers) and memory areas in the RTSJ. Creation of these objects outside of these factory methods have no locality defined and can be located at the JVM's discretion.

For full details of the locales API see deliverable D2.1.

3.2 Architectural Discovery

Locales provide locality information; essentially that given software elements should be located together and with a given resource reservation. In the JUNIPER approach, locales may be mapped to specific hardware elements. The JUNIPER API includes a hardware model for this purpose[1] which is based on the idea of *architecture patterns*. Patterns are used because of two main reasons:

1. Programmers of big data systems are more concerned about the *class* of architecture than its precise details. For example, whether or not it has coherent caches, or whether or not all memory is of equal speed. Architecture patterns capture this well.

2. Given the large platforms used for big data systems, the programmer does not require the low-level control afforded by techniques such as affinities, in which each thread is bound to a specific set of individual processors. This is onerous for large systems and lacks portability. Instead the programmer wishes to be able to express that a given large group of threads should be located on a given large group of tightly-coupled processors, at which point the runtime and infrastructure can be trusted to schedule and place these appropriately.

Given the above points, the patterns exposed by the JUNIPER API are:

- NUMA: The Non-Uniform Memory Architecture pattern provides few guarantees. It will contain a single address space, but caches may be incoherent and memory access times are unknown. An example of this is an integrated system with a CPU and GPU that contain separate memories.

[1] Recall that in JUNIPER, the hardware platform is constrained to be that which can be considered as a single multiprocessor platform; it does not include multiple platforms that are distributed across local or wide area networks. These wider concerns are addressed in deliverable D2.1.

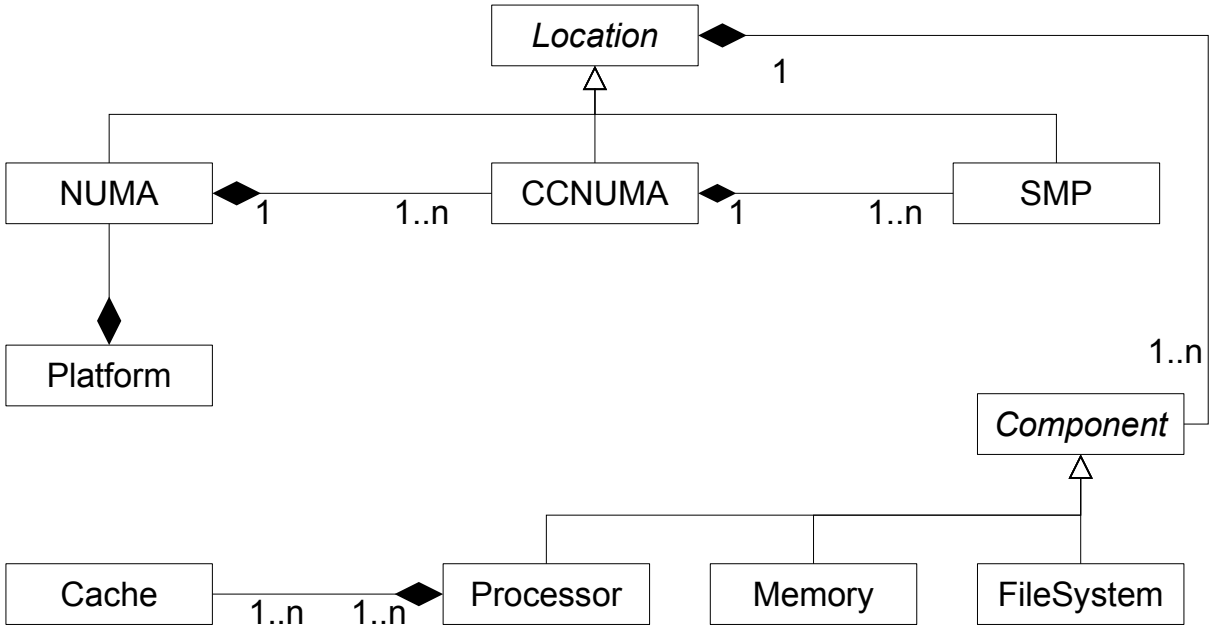

Figure 4: Class diagram for the JUNIPER architectural model.

- ccNUMA: The Cache-Coherent NUMA Architecture constrains the NUMA architecture with the guarantee that caches will be kept coherent from the point of view of the Java programmer. (Coherence may therefore be implemented in hardware or the OS.) Memory access speeds are still unknown and variable. Most server-class systems are this model, in which many processors share multiple banks of memory but a hardware cache coherency layer handles sharing of data.
- SMP: The Symmetric Multiprocessing pattern represents an architecture in which components are more tightly-coupled. Access times to memory are uniform within a reasonable error bound. Variation is only due to contention on a shared memory bus or cache effects, not because memory is a greater 'distance' from the processors. Any modern multicore CPU (i.e. Intel Core i3) follows this model.

Figure 4 shows the Java classes of the architectural model that is exposed by the JUNIPER API. These components are:

- `Location`: The ancestor of all architecture patterns, this represents a locality in the JUNIPER system. A locality is a place where the user's code may execute; it is not required to model items that the user does not control (i.e. management nodes or login nodes). Locations are multiprocessor system made up of processors, memory, and file systems.
- `NUMA`: Represents the NUMA architecture pattern.
- `CCNUMA`: Represents the ccNUMA architecture pattern.
- `SMP`: Represents the UMA architecture pattern in which its components are closely coupled, and there is uniform access time when accessing main memory. UMA systems are cache coherent.
- `Platform`: A static class which provides an interface to platform-wide methods.

- `Component`: An abstract basic building block of the platform. Each component has a distance to other components which is based on the System Locality Information Table (SLIT), a standard table introduced by the ACPI specification [19].
- `Processor`: An instance of this class represents a logical processor that can have its own caches. The cache belonging to each processor is represented separately by a `Cache` class.
- `Memory`: An instance of this class represents a memory device and may be used, for example, to determine the available storage before choosing when to execute a task, or what algorithm to use when doing so.
- `FileSystem`: Represents a device that host file systems and provides methods to fetch data efficiently from the disk. Section 3.4 discusses disk access in JUNIPER.

For full details of these API classes consult deliverable D2.1.

The architectural model provides a portable way to represent the topology of most multiprocessor platforms. A single-program JUNIPER application is constrained to run on a ccNUMA architecture subsystem within the platform. Multiple programs can be hosted on the platform, but these must communicate using a middleware or distributed computation framework.

3.3 Example

The following example shows a use of the architecture discovery API.

```
public class PrimeGenerator {
    public static void main(String[] argv) {
        long hi = /* get from user */;
        Platform p = Platform.getPlatform();
        NUMA numa = p.getRootLocation();
        CCNUMA ccnuma = numa.getChildren().get(0);
        SMP smp = ccnuma.getChildren().get(0); // Get first SMP
        Locale locale = new Locale(smp);
        int ncpu = smp.getNumCPUs();

        for (int i = 0; i < ncpu; i++) {
            final long min = ((long) i) * hi / (long) ncpu;
            final long max = ((long) i + 1) * hi / (long) ncpu;
            Thread th = locale.createJavaThread(() -> {
                // Search from min to max to find a prime
                //...detail skipped...
                if (found) System.out.println(n);
                //...
                }
            });
            th.start();
        }
    }
}
```

In the example, a set of threads will be created in order to parallelise the search for prime numbers. In order to maximise the locality of the searching threads (to minimise instruction cache misses)

the threads are all assigned to a single SMP architectural pattern. One thread is created for each CPU of the chosen SMP. Once the target number of threads is determined, the total search space is split evenly between the threads spawned. Note that hardware mapping is performed by creating a `Locale` and passing the target `SMP` to the constructor. Then, the `Locale` instance is used to spawn threads inside that locale.

```
final IntStoredCollection storedColl = new IntStoredCollection(filename);

Platform p = Platform.getPlatform();
CCNUMA ccnuma = p.getRootLocation().getChildren().get(0);
for(SMP smp : ccnuma.getChildren()) {
    Locale locale = new Locale(smp);
    for(CPU c : smp.getCPUs()) {
        Thread th = locale.createJavaThread(() -> {
            storedColl.stream().forEach(data -> processData(data));
        });
        th.start();
    }
}
```

In the second example, the platform is examined for the number of SMP patterns that exist within it. For each SMP a Locale is created and the locale used to create one thread per processor. This shows how Locales are used to create thread groups that will be tightly-coupled. Each SMP's threads will only be scheduled on the processors of that SMP. For brevity, in this example all the threads do the same thing, which is read values from disk and process them. However, to ensure that the disk is accessed efficiently a *Stored Collection* is used, which is described in the following section.

3.4 Efficient disk access

The JUNIPER architectural model assists the programmer to create parallelism to exploit the host architecture through the use of Java 8's parallel streams. It also helps the programmer to reduce communication and memory overheads through the use of locales and architectural patterns. Finally, the API also considers the efficient use of hard disk-based storage through the provision of *Stored Collections* (Section 3.4.1) and *Disk-Aware Iterators* (Section 3.4.2).

Note that the JUNIPER project also uses MPI-IO for network-based disk access, and this is described in deliverable D2.1.

3.4.1 Stored Collections

In standard Java, collections are used to store and manipulate data. However they must be first populated, which in a standard Java program requires the programmer to open a file and read its contents into memory for processing. In a big data system this causes problems because files are often very large and may be perpetually growing. The JUNIPER API solves this by providing stored collections, which provide the standard collection and stream interfaces but they only read their data from disk as it is required. Because stored collections have a standard collections interface, they

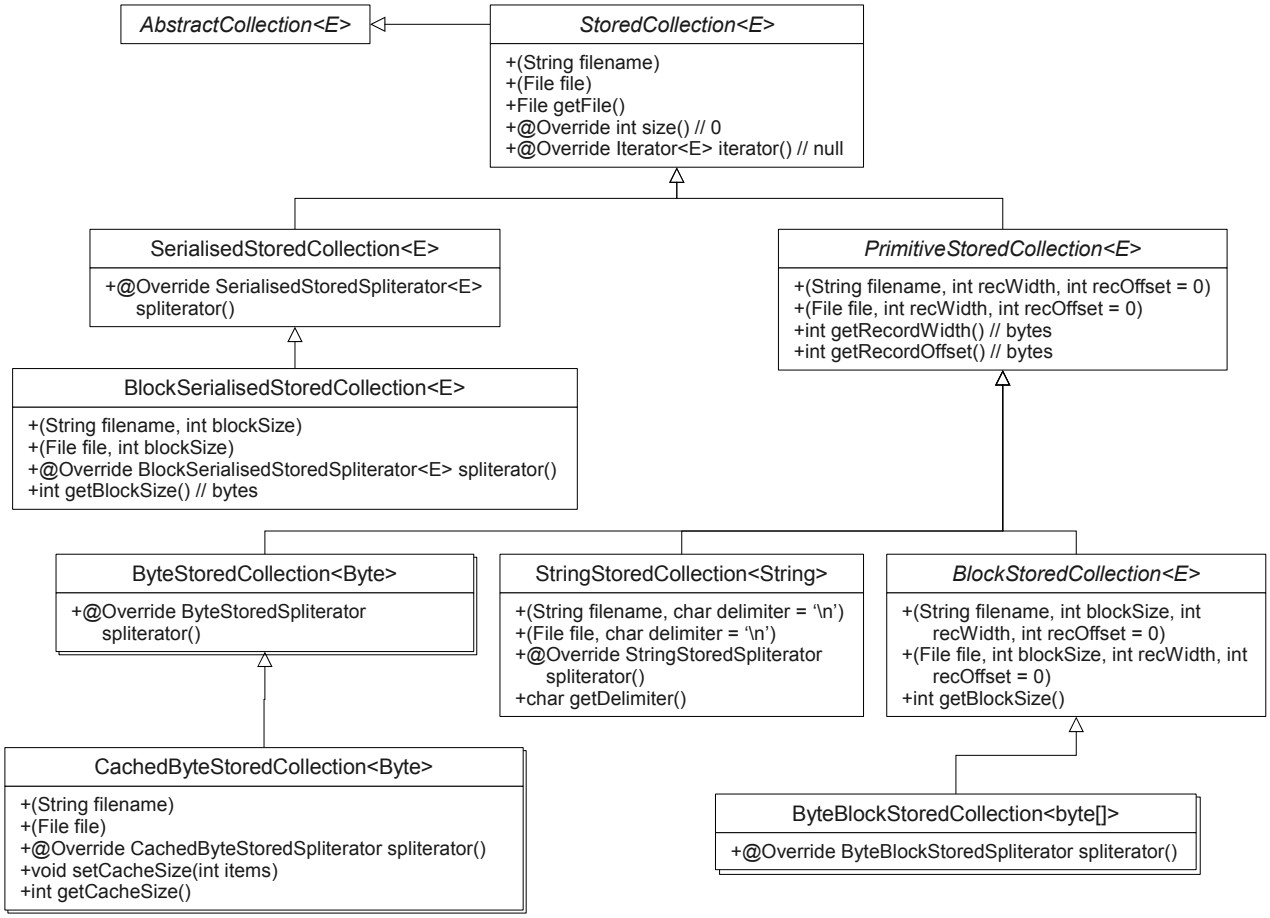

Figure 5: The stored collections in the JUNIPER API.

can transparently replace the standard Java collections. The stored collection hierarchy is shown in Figure 5.

The primitive stored collections (i.e. `ByteStoredCollection` or `CharStoredCollection`) provide an interface for files which simply hold Java primitives one after the other. When these primitives should be read in blocks, the block stored collections (i.e. `ByteBlockStoredCollection`) allow for bulk reads of larger sizes. More complex Java types can be stored in stored collections through the use of the `SerialisedStoredCollection` which holds byte arrays produced from Java's serialization mechanisms. Clearly, because the host language is Java the end user can extend these classes and provide application-specific implementations where required.

The main advantages of Stored Collections are:

- Very easy to program with due to their standard `Collection` and `Stream` interfaces.
- Predictable and small memory footprint compared to standard collections.
- Efficient disk access when combined with Disk-Aware Iterators (Section 3.4.2).

Consider the following program:

```
DataInputStream dis = new DataInputStream(new FileInputStream(filename));
LinkedList<Integer> list = new LinkedList<Integer>();

while(true) {
  try {
    int i = dis.readInt();
    list.add(i);
  } catch (EOFException eof) {
    break;
  }
}

int total = list.stream().filter(i -> validValue(i) == true).sum();
```

This program reads the entire contents of the file `filename` into a `LinkedList`. It then calls `stream()` to get a stream interface from the collection and then `filter`s and `sum`s this stream. On many systems this is a reasonable implementation, but if `filename` is a very large file then this will result in a large memory footprint. Contrast with the Stored Collection variant:

```
IntStoredCollection isc = new IntStoredCollection(filename);
int total = isc.stream().filter(i -> validValue(i) == true).sum();
```

In this version, values are only read from the file as requested by the `filter` and `sum` operations in the stream so memory usage is minimised. Table 1 shows some early results performed on an example word search application to evaluate Stored Collections. In the table, $Heap_{max}$ refers to the maximum observed heap usage of the application as it processed the input file. The test JVM was the Oracle JVM which tends not to perform garbage collection until it is running low on memory, $Heap_{gc}$ shows the maximum observed heap size when garbage collection is periodically forced. These results show, as expected, that in this preliminary test the stored collection can vastly reduce memory use by only holding a small portion of the input data at any one time. The execution times for larger files is also faster, but this is due to natural pipelining effects where the stored collection can be fetching

Table 1: Preliminary comparison of standard Java collections over Stored Collections; with and without disk-aware iterators.

Data size		In-mem collection	Stored collection (simple)	Stored collection (disk-aware)
7.5MB	Time	4.71s	7.37s	7.95s
	$Heap_{max}$	283MB	190MB	225MB
	$Heap_{gc}$	117MB	79.9MB	114MB
75MB	Time	9.22s	7.96s	8.19s
	$Heap_{max}$	760MB	1,180MB	839MB
	$Heap_{gc}$	483MB	237MB	461MB
750MB	Time	43.3s	31.1s	32.3s
	$Heap_{max}$	2,870MB	1,430MB	1,550MB
	$Heap_{gc}$	2,000MB	492MB	665MB
7.5GB	Time	Out of memory	254s	245s
	$Heap_{max}$		2,390MB	2,650MB
	$Heap_{gc}$		781MB	1,000MB
75GB	Time	Out of memory	3,020s	2,360s
	$Heap_{max}$		4,110MB	3,980MB
	$Heap_{gc}$		1,180MB	1,220MB

data from disk whilst the application processes data already loaded. The "disk aware" values refer to when the stored collection is instantiated with a disk-aware iterator, discussed in the following section.

For full details of the Stored Collections API consult deliverable D2.1.

3.4.2 Disk-Aware Iterators

Stored Collections allow the programmer easy access to on-disk data structures, however it is necessary to mediate the physical access to the disk for efficiency reasons. Hard disks are most efficient when accessed sequentially and random access can result in data rates that are orders of magnitude slower [20]. However, if the programmer uses standard Java 8 parallel streams to parallelise access to a stored collection then the access patterns will be unlikely to be optimal as the various threads compete to use the data.

The solution introduced in JUNIPER is to mediate disk access with Disk-Aware Iterators. In order to understand how Disk-Aware Iterators work it is necessary to first understand Java 8 *Spliterators*. Spliterators are used by parallel stream operations to traverse and partition elements of a stream. Normal iterators provide the `next()` method to move from the current element to the next. Spliterators also provide the `trySplit()` method which, when called instructs the Spliterator to split itself into two Spliterators which each cover different sections of the same initial data structure.

During execution of a parallel stream the Java 8 infrastructure first obtains a Spliterator from the input stream. It them repeatedly calls `Spliterator.trySplit()` until the final call returns `null` indicating that the stream can no longer be split. It then passes the Spliterators it obtained

to the threads in its executor thread pool to evaluate the stream. Each thread uses its Spliterator's `tryAdvance()` method to iterate over their section of the stream as if it were a standard iterator.

It can therefore be seen that Spliterators (along with the parallel stream thread pool) decide on the appropriate access pattern of a given data stream. The user can of course provide their own Spliterator for more complex data streams. Java 8's default implementations are tuned according to the assumption that the input stream will be in memory, but when the stream is on disk (as with a Stored Collection) this results in RAM-style access patterns being applied to the disk, which is very inefficient.

Disk-Aware Iterators are the solution to this. They are used inside the Stored Collections and make use of the architecture discovery API to determine the class of storage device they are accessing; adjusting their split behaviour accordingly. Most disks are faster when accessed in sequential blocks, so the standard behaviour of a Disk-Aware Iterator is to observe this as closely as possible. However, if the storage is a RAM disk, RAID array, or provides true parallel access (i.e. Lustre), then it can relax these restrictions.

Initial tests with Disk-Aware Iterators indicate that superior data throughputs can be achieved over the naïve Java solution without any additional work from the programmer, as shown in Table 1. Further testing and development will be continued though the JUNIPER project.

3.5 Required OS Support

Given the requirements of the architecture discovery API, the JUNIPER project will make use of the Portable Hardware Locality (hwloc) package [15]. hwloc provides an abstraction of the hierarchical topology of the platform architecture, including NUMA memory nodes, sockets, shared caches and processor cores. It also gathers various system attributes such as cache and memory information and the location of I/O devices and network interfaces. An example hwloc representation of a NUMA architecture is shown in Figure 6. hwloc supports all major operating systems and system architectures. It does not, however, provide information on the file systems installed in a host so Disk-Aware Iterators must implement this support themselves through OS-provided APIs.

hwloc will be sufficient for providing support for architectural patterns. Locales, however, require additional support. The target OS will need to provide the following support:

- Affinities: Threads must be able to have their affinities set so that they will only be scheduled for execution on a subset of the available processors.
- SMP-aware allocation: Processes must be able to request memory allocation from a specific SMP node in a ccNUMA architecture.
- hwloc support: hwloc will be used for architecture discovery as detailed above.
- Thread group budgets: Processes must be able to set resource budgets for thread groups. This may limit their CPU use, use of disk bandwidth, use of memory, or other resources.

The JVM must also support the following features:

- Affinities: Java-level schedulable objects must be able to have their affinities mapped so that they will only be scheduled for execution on a subset of the available processors.

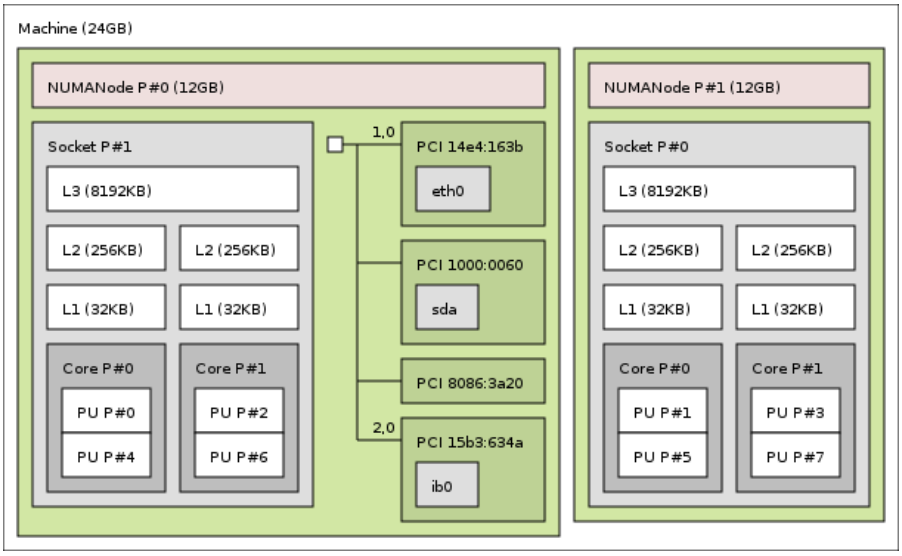

Figure 6: Architecture representation in the hwloc package. (Image source: [1])

- Physical memory areas that allow a schedulable object's backing store to be contained within an SMP's local memory.
- Support for a heap that is distributed across a ccNUMA architecture (i.e. a modern high-performance server).
- SMP-aware allocation: Schedulable objects must be able to request memory allocation from a specific SMP node in a ccNUMA architecture.
- Garbage collection that does not copy live object from one node's memory to another during any compaction phase.
- Schedulable object budgets: Processes must be able to set resource budgets for schedulable objects. This may limit their CPU use, use of disk bandwidth, use of memory, or other resources.

4 Conclusion

This document has described how the JUNIPER project assists in the development of Java-based, real-time, big data systems for deployment in modern cloud-based or high-performance environments. These environments are characterised by the following features:

- Highly-variable architectures: Servers in a cluster may have architectures quite different from each other.
- High-degree of parallelism: Simple thread-based parallelism is important, but must be paired with data parallelism in order to exploit the entire system.
- Lower reliability: Cloud-based infrastructure is subject to the ephemeral nature of the internet.
- Virtualised hardware: In both cloud and high-performance systems, access to the hardware is provided through virtual machines. The hardware (number of CPUs, memory size etc.) of these virtual machines may change frequently as the total number of users changes.

In order to assist with parallel data processing, the JUNIPER project is adopting use of the new Java 8 standard. Java 8 provides streaming constructs which make the descriptions of pipelines and data-parallel operations easy. However, Java 8 alone is not sufficient.

In order to effectively exploit such systems, the programmer must develop software which can detect its host architecture and adjust its characteristics accordingly. The JUNIPER project uses the notion of *architectural patterns* to aid this. Architectures are abstracted as `NUMA`, `ccNUMA`, and `SMP` patterns. The JUNIPER API provides *locales* which allow the programmer to group sets of threads and data. Locales are then allocated to an architectural pattern to inform the JVM and OS that the threads of the system should be allocated to the processors of the pattern. Similarly, the data in a locale, and the memory allocated by the threads of a locale, are all assigned to the physical memory of the architecture pattern. This locality information allows the programmer to control the placement of threads and data through the system in a portable way, without having to manually map specific threads to specific CPUs as required by existing approaches based on CPU affinities.

The JUNIPER API also provides *Stored Collections*, which are Java collections in which the backing data store is on disk rather than in heap memory, as with a normal Java collection. Stored Collections lazily read data from disk as required, which is important in a big data system as frequently the entire dataset will not fit into system memory. Stored Collections are also architecturally-aware because they use another feature of the JUNIPER API, *Disk-Aware Iterators*. Disk-Aware Iterators use architecture discovery to determine the kind of disk that they are accessing (standard magnetic, SSD, RAM disk etc.) and details such as RAID level or the kind of file system. From this, an efficient element access pattern is derived and the iterator mediates disk access accordingly. Initial experiments have shown Stored Collections and Disk-Aware Iterators to increase the efficiency of Java 8-style parallel streams over data stored on disk, with no increase in code complexity.

References

[1] The Portable Hardware Locality project (hwloc). http://www.open-mpi.org/projects/hwloc/, January 2014.

[2] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[3] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM.

[4] Robit Chandra et al. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[5] NVIDIA Corporation. CUDA Programming Guide ver 1.1. http://developer.nvidia.com, 2007. Accessed 23/07/2010.

[6] Aaftab Munshi, editor. *The OpenCL Specification*. Khronos OpenCL Working Group, 2008.

[7] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[8] J. M. Bull and M. E. Kambites. JOMP - an OpenMP-like Interface for Java. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 44–53. ACM, 2000.

[9] GACP Ganegoda, D Samaranayake, L Bandara, and KADNK Wimalawarne. Jconqurr - a multi-core programming toolkit for Java. *International Journal of Computer and Information Engineering*, 3(4):223–230, 2009.

[10] N. Giacaman and O. Sinnen. Pyjama: OpenMP-like implementation for Java, with GUI extensions. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 43–52. ACM, 2013.

[11] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen. JaMP: An implementation of OpenMP for a Java DSM. *Concurrency and Computation: Practice and Experience*, 19(18):2333–2352, 2007.

[12] Oracle Corporation. JDK 8 Schedule and status. http://openjdk.java.net/projects/jdk8/, September 2013.

[13] InfiniBand Trade Association. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[14] Xyratex Ltd. The Lustre file system. http://www.lustre.org, December 2013.

[15] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.

[16] Emilio Francesquini. Jhwloc - Java interface for hwloc. https://launchpad.net/jhwloc/, December 2013.

[17] Oracle Corporation. Project Lambda. http://openjdk.java.net/projects/lambda/, December 2013.

[18] Mike Duigou. JEP 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107, Sepetmber 2011.

[19] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced configuration and power interface specification revision 5.0. http://acpi.info/DOWNLOADS/ACPIspec50.pdf, December 2011.

[20] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.