High Performance Computing with Java Streams

Rui Silva and João L. Sobral(✉)

Centro Algoritmi, Universidade do Minho, Braga, Portugal
{ruisilva,jls}@di.uminho.pt

Abstract. Java streams enable an easy-to-use functional-like programming style that transparently supports parallel execution. This paper presents an approach that improves the performance of stream-based Java applications. The approach enables the effective usage of Java for HPC applications, due to data locality improvements (i.e., support for efficient data layouts), without losing the object-oriented view of data in the code. The approach extends the Java collections API to hide additional details concerning the data layout, enabling the transparent use of more memory-friendly data layouts. The enhanced Java Collection API enables an easy adaptation of existing Java codes making those Java codes suitable for HPC. Performance results show that improving the data locality can provide a two-fold performance gain in sequential stream applications, which translated into a similar gain over parallel stream implementations. Moreover, the performance is comparable to similar C implementations using OpenMP.

Keywords: Java parallel streams · Data layout · Data locality

1 Introduction

The development of high-performance applications requires the exploitation of parallelism and efficient data access. Programming languages should now support the exploitation of parallelism and efficient data storage, promoting data locality to deliver high performance. Data storage is also essential to exploit all the potential of modern processing units (e.g., vector processing). Traditionally, to improve performance, the developer introduces the optimisations in the domain code, making the code less abstract and dependent on the execution platform.

The Java language brings the *write once run anywhere* philosophy: the same program can run on any system that supports a Java Virtual Machine. Java 8 introduced the Stream API [1], which enables easy-to-use parallelism over Java collections (e.g., data-parallel processing). Unfortunately, the Java object model compromises the suitability of the Java stream-based processing for High Performance Computing (HPC). The main limitation is the lack of data locality in Java collections (including arrays of objects). Java collections are implemented with pointers to objects (e.g., an ArrayList is stored as an Array of Pointers (AoP) to objects). This is a consequence of using type erasure [2] to implement

R. Chaves et al. (Eds.): Euro-Par 2021, LNCS 13098, pp. 17–28, 2022.
https://doi.org/10.1007/978-3-031-06156-1_2

generic collections in order to avoid code bloat. With type erasure, a single collection implementation can be used for all concrete types, since the collection only needs to store pointers to [generic] objects. The AoP-based storage is also convenient to implement many object-oriented features, like polymorphism.

The AoP-based implementation of Java collections has negative consequences for HPC, namely: 1) additional memory references are required to access object fields; 2) entities in collections might not be stored in contiguous memory (no spatial locality); 3) object headers and memory alignment waste space in memory (object headers are required for JVM runtime checks). These result in a higher number of instructions (memory accesses) and lower data locality, which decreases the performance due to stronger impact of the memory bottleneck.

The data locality can be improved by using a layout with higher data locality, by storing object collections as a Structure of Arrays (SoA). However, in this case, the programmer must "give up" of the Java object-oriented view of data (e.g., using raw arrays of data). Moreover the programmer might be forced to drop the usage of the Java collections API (and consequently, the Stream API). Additionally, it is not feasible to use SoA layouts on the wide base of existing Java code, since a huge code refactoring effort would be necessary.

The research challenge addressed in this paper is how to improve the data locality of Java collections in order to make the Java stream API more suitable for HPC, without dropping the object-oriented view of data collections (i.e., preserving compatibility with the stream API and with the Java object model). Specifically, how to transparently support more efficient data layouts (e.g., SoA layouts) in Java stream-based parallel processing.

2 Java Stream API

The *Stream* interface (left of Fig. 1) provides methods to process a data stream, namely: 1) *forEach*: performs an operation on each element of the data stream; 2) *filter*: generates a new stream with a subset of the original stream and 3) reductions, such as *count*, that returns the number of elements in a stream. The *Collection* interface is the root of all Java collections and was enriched in Java 8 with the *stream* default method that returns a stream view of a collection. Thus, conceptually, stream-based processing is supported over all Java collections.

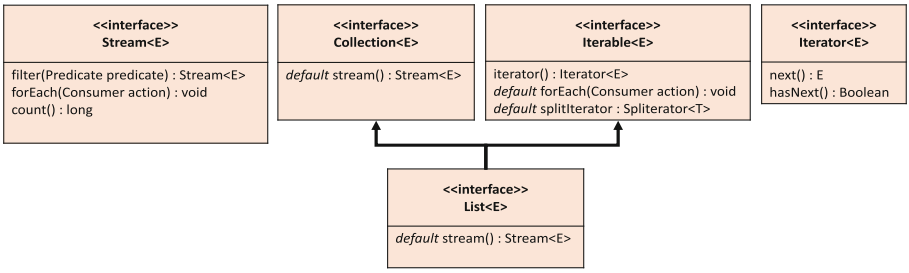

Fig. 1. Java collections and stream API

A parallel stream enables parallel processing over the objects in a collection. Thus, the introduction of the stream API enabled the specification of explicit data-parallel processing over the wide range of Java collections. In particular, the *List* interface (bottom of Fig. 1) extends the *Collection* interface, providing a set of default methods for stream-based parallel processing over index-based collections (e.g., the *ArrayList* implements *List* interface).

The stream API relies on well-defined collection interfaces, making it possible to provide new collection implementations that comply to the stream API, avoiding the need to reimplement a parallel computing infrastructure. This is the key point in the context of the presented work, since user-provided collection implementations can use the Java stream API and fully exploit the parallel processing stream infrastructure.

Java collections rely on the *Iterable* and *Iterator* interfaces (see right side of Fig. 1) to hide the implementation details of collections. The classes implementing the *Collection* interface must also implement the *Iterable* interface, which provides a method to return an *Iterator*. User-provided collections must implement a collection interface and the *Iterable/Iterator* interfaces to take advantage of the Java stream API.

The most commonly used data structure in Java, according to the study in [7], is the *ArrayList*, whose implementation is based on the AoP data layout. Figure 2 illustrates the data layout of an *ArrayList* of objects of a class *Particle*. The array of pointers itself requires a header (at least 16 bytes) and there is also a header on each object (at least 12 bytes). This layout has three main problems: i) there is a pointer de-refencing to access each object, which introduces an additional memory access per element; ii) object headers and object alignment introduce a significative memory space for small objects, which might saturate the faster levels of cache on modern processing units (e.g., the small L1 cache); iii) objects referenced by those pointers might not be in consecutive memory addresses (i.e., weak locality of reference). The locality of references can be improved by sorting objects in memory (e.g., a modified JVM implementation was presented in [9] that performs object sorting during garbage collecting), but the problems i) and ii) still remain. The AoP layout is not well-suited for HPC due to its pointer-based nature and is not compatible with vectorisation, but it can abstract from the details about the object pointed-to making it well-suited to implement object models. For instance, it supports inheritance (e.g., a collection can hold pointers

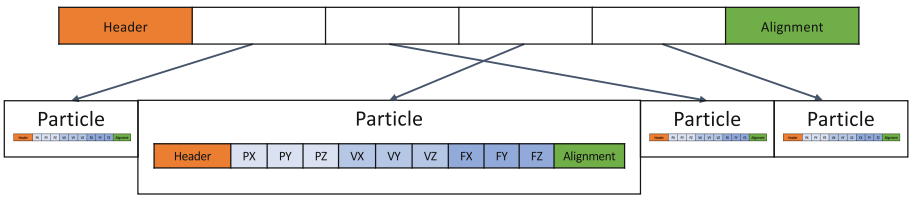

Fig. 2. Java collections data layout example

to objects from different classes that implement same [sub]type), pointers to generic objects can be used in method calls, etc.

One alternative is the use of an Array of Structures (AoS) layout where data entities are stored on consecutive memory addresses. Wimmer et al. [12] modified a JVM implementation for this purpose, which automatically inlines object fields by placing the parent and children in consecutive memory places and by replacing memory accesses by address arithmetic. They concluded that an automatic AoP to AoS transformation at JVM level requires a global data flow analysis, since Java byte-codes for accessing array elements have no static type information (i.e., they suffer from type erasure due to its AoP nature). The AoS layout is more memory-friendly than AoP making it more suitable for HPC but it is still not effective for vectorisation [3] (requires scatter and gather operations). Moreover, the Java object model is more difficult to support in the AoS layout (e.g., inheritance) and it requires low-level workarounds since Java has no support for pointers and, consequently, has limited support for this layout. Therefore, the AoS layout is not an attractive solution for HPC in Java.

The most effective layout for HPC is a Structure of Arrays (SoA), since it is also essential to enable efficient vectorisation. In this layout a collection is backed by multiple arrays, one for each data field. However, this layout drops the object-oriented view of the data, presenting several challenges to support the Java object model, namely, how to support objects and method calls on these objects, inheritance, etc. The main advantage of this work is to support the Java object model when using the more memory-friendly SoA layout.

There are some Java-based approaches that avoid the AoP layout problem for arrays of primitive types [4,5], but they do not support collections of structured data types (e.g., a transparent SoA layout for generic object collections). Moreover, these are not drop-in replacements for Java collections, since the generic Java interfaces (e.g., the *List* interface) are not supported. As a consequence they are not compatible with the stream infra-structure. The work presented in this paper supports the SoA layout for collections of complex data types and provides a drop-in replacement for Java collections, supporting the stream interface. Thus, the Java stream parallel processing infra-structure becomes available for HPC programmers.

3 Gaspar Stream-Based API and Implementation

The Gaspar framework [10] provides multiple mechanisms to improve data locality, namely, it encapsulates the data entities into framework provided collections that support efficient data layouts. The contribution of this paper is to show how to make the Gaspar data API compatible with the Java stream API, in order to: i) use the stream parallel processing infra-structure over Gaspar collections; ii) improve data locality of stream-based applications by using the Gaspar supported locality optimisations and iii) ensure compatibility with the Java API (including the object model).

3.1 Stream-Based Gaspar API

The Java collections API (i.e., Fig. 1) can hide details of the collection representation (e.g., the internal usage of an array or linked-list). However, it can expose details of the object layout, so it cannot hide a change in a collection layout from an AoP to a SoA. The Gaspar data API can hide additional details concerning the data representation, including the concrete data layout. The key difference to the Java API is that data entities should be represented using interfaces with getter and setter methods for each data property.

The left side of Fig. 3 presents an example of a *Particle* interface with PX, PY and PZ properties. The code on the right illustrates the Gaspar API usage to get the position (x, y, and z) of a particle. The key difference to the Java API is the usage of getter methods in lines 5–7.

The Gaspar API was extended to provide compatibility with Java collections and streams, by extending the framework collections the implement the Java *List* interface. This opens the door to the usage of Gaspar collections in Java applications that rely that interface: only an update to the usage of getter and setter methods is required, something that well-designed code probably already includes. The Gaspar collections now also implement the Java *Iterable/Iterator* interfaces, which were already used in the example of Fig. 3.

Parallel processing can now be transparently explored using parallel streams. This enables the usage of the stream parallel processing infra-structure, instead of the built-in Gaspar parallel maps. Figure 4 illustrates the usage of the *forEach* method on a collection of *Particles* to call the *move* method on each particle in parallel. This uses a lambda function define the operation to apply to each stream element.

3.2 Design and Implementation

The Gaspar API was designed for HPC and to support most features of the Java object model, some of which are difficult to support when using SoA or AoS layouts as a backend of object collections. This is a consequence of using an object model relying on AoP data layouts. For instance, in the Java model, objects are always passed by reference.

```
interface Particle {
  double getPX();
  double getPY();
  double getPZ();
}
```

```
1 // col is a List<Particle>
2 Iterator<Particle> it = col.iterator();
3 while (it.hasNext()) {
4   Particle p = it.next();
5   x = p.getPX();
6   y = p.getPY();
7   z = p.getPZ();
8   (...)
9 }
```

Fig. 3. Gaspar API usage example (*Particle* example). The code on the left defines the particle interface and the code on the right in an example of the Gaspar API usage

```
col.parallelStream().forEach( p -> p.move(); );
```

Fig. 4. Example of parallel streams: *col* is a Gaspar collection (*gCollection<Particle>*) that implements the interface *List<Particle>*. The method *move* of class particle can be called, in parallel, on all particles in the stream

```
// Gaspar API
force(Particle p){
  dx=this.getX()-p.getX();
  ...
}
```

```
// AoP layout (JGF MD)
force(Particle p){
   dx = this.x - p.x;
   ...
}
```

```
// SoA layout
force(Particles p,
      int p1, int p2){
  dx = p.x[p1]-p.x[p2];
  ...
}
```

Fig. 5. Particle force computation example in Gaspar API, Java AoP and SoA

Figure 5 illustrates the code of several data layouts for a collection of *Particles* using a code snippet from the MD case study. The Gaspar API (left of the figure) is used for all layouts, but plain Java requires different codes for AoP and SoA layouts (e.g., the AoP layout is from the JGF MD benchmark [11]). This example illustrates the complexity of using the efficient SoA layout in Java streams:

1. *Entities and methods.* The *force* method computes the force between two particles. In the AoP layout, the class *Particle* represents an entity from the domain and that entity defines a method *force*. In a SoA implementation there is no particle entity in the code (only array of properties, such as the x array), so the force method must be declared outside of the *Particle* class. This has a huge impact when using Java streams since there is no object *Particle* (e.g., it is not possible write the example of Fig. 4, which calls the *move* method on each particle of a collection).
2. *Object references.* The method *force* receives another *Particle* as an argument. In the AoP layout this is implemented by simply passing a pointer to another object. In the SoA layout the integer index of each particle should be used instead. Moreover, if the force is an external method that receives two particles, it will also require access to the *Particles* data. An alternative would be to "construct" a particle from the SoA data, when required, but it could lead to additional overhead and cannot be used in cases where the original particle is updated, since in the Java model objects are always passed by reference. This layout also makes the support for Java iterators complex since the next method should return a reference to an object in the collection (which might be updated). Providing support for Java iterators is fundamental to use the Stream API.
3. *Compositions of objects.* In object-oriented applications it is common to define objects as being composed of other objects (e.g., a particle might be composed by three objects: position, velocity and force). This is trivially supported in AoP layouts by using pointers to other objects. In the SoA layout these compositions can be manually implemented by using a single structure of arrays for all object fields.

Implementation Overview. The Gaspar API relies on the *gCollection* and *gIterator* interfaces, which now also support the Stream API (e.g., *List* and *Iterator*), implementing the necessary methods as *default methods*. As a consequence, *gCollections* can now be used in the Java stream processing infra-structure.

The development of an application in the Gaspar framework starts with a specification of a diagram similar to an UML domain model (e.g., yellow box in Fig. 6), where programmers specify the properties of each domain entity (i.e., their getter and setter methods and other relevant properties) as well as the relationship among them (aggregations). The Gaspar framework provides a visual tool (eclipse plug-in) to support this step.

A second tool generates the concrete implementations for the AoP and SoA layouts, based on the provided domain model. Figure 6 includes the generated classes for the SoA layout. The class *gCollectionParticleSoA* implements the interface *gCollection<Particle>* (note that the *gCollection* interface extends the *List* interface) with the SoA layout and the class *gIteratorParticleSoA* implements both *Particle* and *gIterator<Particle>* interfaces. In this implementation strategy the *gIterator* acts as a proxy to the actual *Particle* implementation, enabling the *gIterator* to also behave as a *Particle*, and to be used where an object of type *Particle* is expected. This allows the use of *Particle* entities in the base program, which could also include methods (e.g., *Particle move* method of Fig. 4) and the use of object references (a *gIterator* can be used as method parameter, copied, etc.). One feature of the developed tool is the ability "to flatten" aggregations defined in the domain model when generating the SoA representation. This enables the efficient support of composite objects in the stream API.

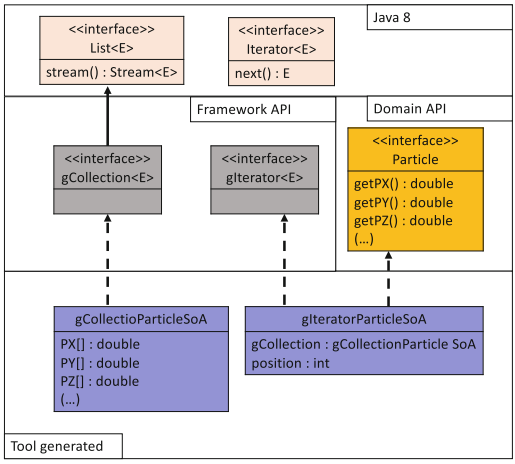

Fig. 6. Gaspar API and implementation overview

4 Evaluation

The benchmarks presented in this section were collected on a Linux machine with 24-cores (two Xeon E5-2695v2 processors) running Cent OS 6.3. The presented performance results are the median of 5 executions, after one warmup execution. The Java results use the OpenJDK 13.0.2 and C results use the GNU g++ 8.2.0.

4.1 Low Level Evaluation: DAXPY

The widely used DAXPY function adds two vectors: *Y += alfa * X*. This case study evaluates the overhead of accessing the elements of a collection, as well as the feasibility of the automatic JVM vectorisation. The DAXPY is an easy-to-vectorise case study, however, it requires two iterators, one for each vector, which might degrade the performance and disable the automatic vectorisation.

Figure 7 shows the pseudo-code of three coding alternatives: 1) a traditional index-based approach to access each vector; 2) Java iterators and a single collection to store both X and Y vectors (the *my2Double* object stores both x and y elements[1]) and 3) stream-based interface with a lambda function.

```
// 1) first approach: index based (two collections)
for (int i=0; i < vectorx.size() && i < vectory.size(); i++) {
  double aux = alpha * vectorx.get(i) + vectory.get(i);
  vectory.set(i, aux);
}

// 2) second approach: Java iterator (single collection)
Iterator<my2Double> itxy = vectorxy.iterator();
while(itxy.hasNext()){
  my2Double xyval = itxy.next();
  double aux = alpha * xyval.getX()+ xyval.getY();
  xyval.setY(aux);
}

// 3) third approach: stream based (single collection)
vectorxy.stream().forEach(
  (xyval) -> {
        double aux = alpha * xyval.getX() + xyval.getY();
        xyval.setY(aux);
   } );
```

Fig. 7. DAXPY implementation alternatives

Table 1 compares the JVM effectiveness to optimise alternatives in Fig. 7:

- The table rows are the three ways of accessing elements of the collection: i) index-based (first approach in Fig. 7); ii) external-iterator based (second approach); and iii) internal-iterator based (stream-based, third approach).
- The table columns are four alternatives to store X and Y vectors: i) using one or two collections (first and second columns); ii) using an AoP or a SoA layout; and iii) using the Gaspar Stream-compatible collections and iterators.
- The value in each cell is the number of instructions to compute each element of Y. The cells in bold are the cases where the JVM successfully vectorised the code[2], resulting in a lower number of instructions. The table also indicates the unrolling degree of the generated code (number in parenthesis).

[1] The use of a single collection for both x and y elements is mandatory, since streams require a single iterator for accessing all the elements (i.e., for internal iteration).

[2] Vectorisation was confirmed by inspection of the assembly generated.

Table 1. Instructions/element and vectorisation results on the DAXPY case study

	Java AoP (2 col)	Java AoP (1 col)	Gaspar AoP (1 col)	Gaspar SoA (1 col)
Index-based	20.0 (1×)	12.5 (4×)	8.3 (4×)	**1.3** (**4**×)
Iterators (external)	44.0 (1×)	14.5 (4×)	10.1 (4×)	**1.3** (**4**×)
Streams (internal)	N/A	12.3 (4×)	9.8 (4×)	**1.3** (**4**×)

Using two different collections (1st column) introduces high overhead, especially when using iterators. A single collection reduces the instruction count and makes it possible to use the Java stream API. The Gaspar AoP layout (3nd column) provides additional reductions on the number of instructions due to more efficient collection management (the compiler removes the type-checking of the elements in the collections, because those elements are created all at once). The Gaspar SoA implementations enable the vectorisation in all cases, including the implementation that uses the stream API. On the other hand, versions using AoP layouts are never vectorised, as expected, due to the usage of pointers.

The Fig. 8 shows the relative performance for vectors with 25M elements. Using two collections delivers the worst performance, due to less data locality (X and Y elements are different objects), resulting in more memory loads, misses, etc. The stream interface has a small performance penalty relative to index/iterator versions. Gaspar implementations provide a gain of around 2.5× when using the SoA layout: the Java compiler was able to optimise the stream-based code to the level of index-based/iterators. The graph on the right of Fig. 8 presents the relative performance of parallel streams and the most efficient C+OpenMP implementation (a parallel loop over raw arrays of X and Y elements, that is also automatically vectorised). The Java parallel streams implementation provides a 2× speed-up, a low gain for a 24-core machine. The Gaspar AoP implementation is slightly better (3× improvement), but the Gaspar SoA parallel stream provides a speed-up of 6× (note that part of this gain is due to better data locality), which is pretty close to the efficient C+OpenMP implementation (6.5× gain).

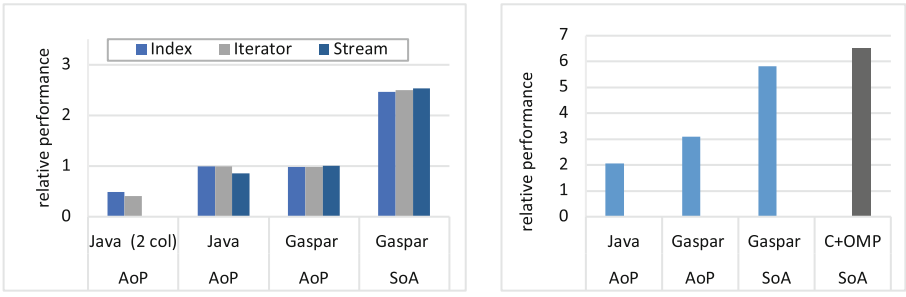

Fig. 8. DAXPY performance: sequential execution at the left and parallel at right. Performance results are relative to the base Java AoP layout with a single collection

4.2 Evaluation of Java Code: JECoLi

The Java Evolutionary Computation Library (JECoLi) [6,8], is a highly configurable Java framework, with a large number of classes that implement algorithms and data representation alternatives used in that domain. The JECoLi framework comes with a large set of case studies that were all updated to use Gaspar collections (e.g., using *gCollections* where a *List* is expected). This case study illustrates how the Gaspar framework avoided a huge refactoring work in order to take advantage of the SoA Layout in the large JECoLi code base.

```
int countOnes(ILinearRepresentation<gBoolean> genomeRep) {
  int countOneValues = 0;
  for(int i = 0;i < genomeRep.getNumberOfElements();i++)
    if ((genomeRep.getElementAt(i)).getValue()) countOneValues++;
  return countOneValues;
}
```

Fig. 9. Changes made to the CountOnes case study: Java Boolean was replaced by the Gaspar gBoolean interface and the introduction of the getValue getter method

The results presented in this section are for the CountOnesEATest problem that is the case study with the largest data size (and most time-consuming). The CountOnes optimisation example creates 10 random solutions, each one with 10000 gnomes that can be true (1) or false (0). The optimal solution is the one with all gnomes set to true.

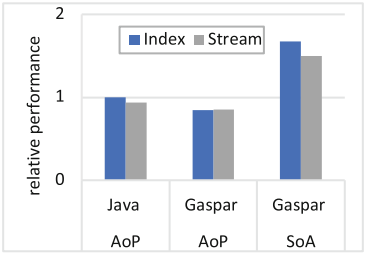

Fig. 10. JECoLi performance

The Fig. 9 illustrates the small impact of the changes required in the CountOnesEATest. The original ILinearRepresentation<Boolean> interface was internally implemented as an ArrayList, implying the usage of an inefficient AoP representation. To overcome that bottleneck the internal ArrayList<Boolean> was replaced by a gCollection<gBoolean> (gBoolean is the framework Boolean interface with the corresponding setter and getter methods). The code in Fig. 9 show the result of this change: the use of the gBoolean interface and the getter method. This quick change to use the Gaspar API allows generating a SoA representation. Figure 10 compares the performance of various implementations (the base line is the original JECoLi implementation, i.e., Java AoP with index-based iterations). The execution time slightly increases when using the framework data API with an AoP representation. This can be explained by an additional overhead of using gBoolean instead of the built-in Java Boolean, but it enables the usage of a SoA layout which improves the performance (a gain of around 1.7×).

4.3 High-Level Evaluation: MD

The third case study is a molecular dynamics simulation, based on the code from the MD benchmark from the JGF Suite [11], which performs a simulation of the behavior of Argon atoms (i.e., Argon particles). The JGF code uses an AoP of particle objects, where each particle has nine properties: position, velocity and a force in 3D space (see Fig. 2). The original benchmark was refactored to use stream-based processing, but the third newton law optimisation (symmetry of forces) was removed from the code to use a simple map parallelism pattern. This case study shows how the proposed approach can provide huge improvements delivering a highly scalable parallel application. A sketch of the code of this case study was already presented (i.e., Fig. 3 and 5). The domain model of this case study was developed using the Gaspar API, which enables the framework to generate the collection representation, making it easier to assess the performance implications of several implementation decisions. Figure 11 summarises the performance results for a problem size of 500000 particles.

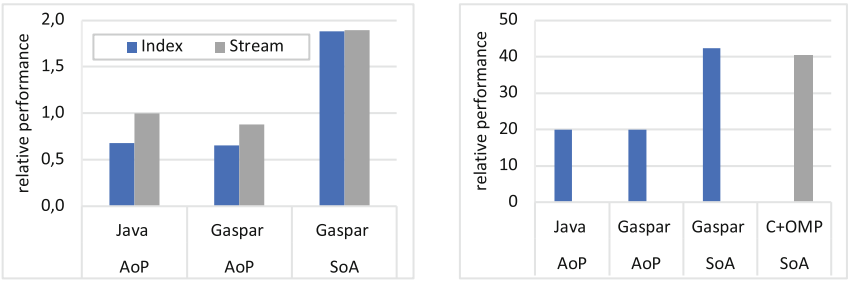

Fig. 11. MD performance: left: sequential execution time; right: parallel versions

In this case study, the stream AoP implementation is used as the baseline for performance comparison, instead of the original JGF AoP index-based implementation, since it is the fastest sequential version (it is faster due to additional virtual machine optimisations). The Gaspar AoP stream implementation introduces a slight overhead, but the Gaspar SoA implementation provides a speed-up of 1.8× over the base stream AoP implementation. The Java parallel stream execution provides a speedup of 20× and the Gaspar SoA parallel stream implementation provides a self-speed-up of 22×. This speed-up is closer to the ideal since the SoA version has better data locality. Overall, the Gaspar SoA implementation gets a performance gain of 42× over the base reference (stream AoP), which is ever faster than C+OpenMP (the C version uses the SoA layout and a parallel loop to compute the forces among particles). The slight advantage of Java is due to the NUMA-aware allocation which is used by default. A fine-tuned C implementation, using the thread binding feature of OpenMP 4.0 was able to get a speed-up of 43.2×, slightly faster than Java, but this fine-tuning does not introduce performance gains in other case studies.

5 Conclusion and Future Work

The Gaspar framework improves the performance of stream-based Java applications by transparently using SoA layouts for object collections. The framework supports the Java object model making it possible to use the more efficient SoA layout almost transparently, in way similar to the built-in AoP Java layout. This enables a more high-level, object-oriented, view of the data. The alternative would be to drop the high-level view and use arrays of object properties. Moreover, the Gaspar framework provides a cost-effective way of improving the data locality of existing Java applications, making those applications better suited for HPC. Performance results show that SoA layouts improve performance by using a better memory footprint and enables automatic vectorisation on modern JVM. Future work includes support for more advanced features of Java, such as polymorphism and support for irregular data structures (e.g., graphs).

Acknowledgements. This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020. The evaluation used the computing infra-structure of the project Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

References

1. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
2. https://docs.oracle.com/javase/tutorial/java/generics/erasure.html
3. https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide-703156.pdf
4. https://bitbucket.org/trove4j/trove/src/master/
5. https://labs.carrotsearch.com/hppc.html
6. https://github.com/jecoli
7. Costa, D., Andrzejak, A., Seboek, J., Lo, D.: Empirical study of usage and performance of Java collections. In: International Conference on Performance Engineering, ICPE 2017, pp. 389–400 (2017). https://doi.org/10.1145/3030207.3030221
8. Evangelista, P., Maia, P., Rocha, M.: Implementing metaheuristic optimization algorithms with JECoLi. In: International Conference on Intelligent Systems Design and Applications, pp. 505–510 (2009). https://doi.org/10.1109/ISDA.2009.161
9. Hirzel, M.: Data layouts for object-oriented programs. In: International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, pp. 265–276 (2007). https://doi.org/10.1145/1254882.1254915
10. Silva, R., Sobral, J.L.: Gaspar data-centric framework. In: Dutra, I., Camacho, R., Barbosa, J., Marques, O. (eds.) VECPAR 2016. LNCS, vol. 10150, pp. 234–247. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61982-8_21
11. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel Java Grande benchmark suite. In: Supercomputing, SC 2001 (2001). https://doi.org/10.1145/582034.582042
12. Wimmer, C., Mössenböck, H.: Automatic array inlining in Java virtual machines. In: International Symposium on Code Generation and Optimization, CGO 2008, pp. 14–23 (2008). https://doi.org/10.1145/1356058.1356061