Gaspar data-centric framework

Rui Silva and J. L. Sobral

Centro ALGORITMI, Braga, Portugal

Abstract. This paper presents the *Gaspar data-centric framework* to develop high performance parallel applications in Java based on a map pattern of computation. The framework provides an efficient data Application Programming Interface(API) that supports data tiling and flexible data layout. Data tiling and layout control enables the improvement of data locality, which is essential to foster application scalability in modern multi-core systems. This paper shows that the framework performance in modern multi-core systems is comparable with pure Java code.

Keywords: Java; locality optimisations; parallel application; pattern

1 Introduction

The high performance in modern computers is achieved by exploiting parallelism and accessing data efficiently. The memory hierarchy of multi-core systems is quite sophisticated whose a behaviour is hard to predict. Finding the best data locality optimisations is a arduous task as it usually requires testing different approaches and parameters. The effectiveness of each optimisation may depend on the particularities of a given platform, compiler and even the application input data.

This paper presents the *Gaspar data-centric framework* aiming to provide a system where data locality optimisations can be quickly implemented. The framework is based on a map pattern of computation which provides an uniform mechanism to express parallelism over data tiles and includes a set data locality optimisations that can be used in applications to improve performance. Furthermore, the user can develop application-specific locality optimisations by using the provided data API.

The next section describes the framework data API and the map pattern of computation and section 3 provides result of performance evaluation. Section 4 discusses related work and section 5 concludes the paper.

2 Data-Centric Framework

One challenge of the framework is to enabled the selection of the data layout without compromise the performance. The framework provides two generic approaches to enable data locality improvements/tuning: i) encapsulates the data

into framework provided collections supporting different data layouts and accessing the data using iterators that hide the concrete data layout; and ii) computation is expressed by a map & reduce pattern of computation over the framework data collections enabling changes to the order of accessing the data. The next two subsections describe these mechanisms.

2.1 Data Application Programming Interface

The data layout has extreme importance in modern computer architectures to deliver high performance. Common data layouts are *Array of Pointers (AoP)*, *Array of Structures (AoS)* and *Structure of Arrays (SoA)*. For illustrative purposes, this paper uses the implementation of a Molecular Dynamics simulation (MD) taken from the *Java Grande Forum(JGF)*[8]. The most computationally expensive part of the code is the computation of forces between all particles, which requires access to the particle position(x,y,z) (Figure 1).

```
//AoP layout
forceParticle(...){
  xi = p1.x;
  yi = p1.y;
  zi = p1.z;
  (...)
}
```

```
//SoA layout
forceParticle(..., int id){
  xi = p1.x[id];
  yi = p1.y[id];
  zi = p1.z[id];
  (...)
}
```

```
//Generic layout
forceParticle(...){
  xi = p1.getX();
  yi = p1.getY();
  zi = p1.getZ();
  (...)
}
```

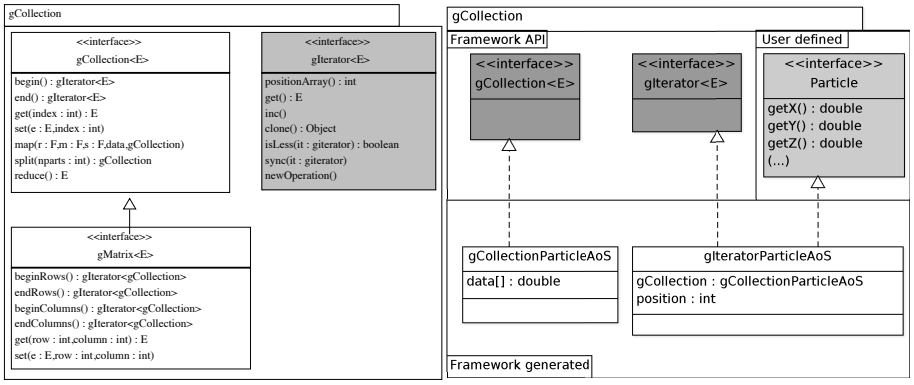
Fig. 1: Force computation using specific layouts and the generic layout supported by the framework.

The best data layout frequently depends on the platform and/or on the computational pattern [4]. Traditional approaches require changes to the base program in order to use a different data layout (e.g. see Figure 1, *AoP* and *SoA* layout). The framework data API provides a high level interface to access data while providing performance similar to tuned implementations. The API hides the concrete data layout (see Figure 1 *Generic layout*).

The framework currently supports two types of data structures (Figure 2a): i) the *gCollection* is a raw set of data; ii) the *gMatrix* is a set of data organised by rows and columns.

The framework provides a data layout generator tool that uses a data model (*Unified Modeling Language*) given by the user (e.g., *Particle* specification) and generates all required interfaces and classes. Thus, it is possible to develop a program without details about the data layout and later select a collection implementation with the most appropriate data layout (e.g., *AoP*, *AoS* or *SoA*). Figure 2b provides an example of two classes generated from the *Particle* interface defined in the MD case study.

In the framework data API, data access is performed using the interface *gIterator* (see Figure 3), which points to an element in a collection. Each collection implements the methods *begin()* and *end()*, that provide iterators to the first and last element. The iterator provides an *inc()* method to advance to the next



(a) Data API (b) Package for the MD *AoS*

Fig. 2: *Gaspar data centric framework* API

data element and an *isLess()* method that compares two iterators. Iterators also provide the *sync()* operation which synchronises the position of two iterators.

```
Iterator it = one.iterator();
while(it.hasNext()) {
  Particle p = it.next();
  xj = p.getX();
  yj = p.getY();
  zj = p.getZ();
  (...)
}
```

```
gIterator it = one.begin();
for(; it.isLess(one.end());
 it.inc()){
  xj = ((Particle) it).getX();
  yj = ((Particle) it).getY();
  zj = ((Particle) it).getZ();
  (...)
}
```

Fig. 3: Comparison between Java iterators and framework gIterator

2.2 Map Pattern

The most common data locality optimisation is the use of data tiles to improve temporal locality [2]. This optimisation requires the creation of additional loops to process data tiles. Moreover, multiple levels of tiling (to address multiple levels of cache) must be implemented by nesting multiple loops, which can be error-prone. Furthermore, it is common to introduce parallelism by running one or more of these loops in parallel [9].

The framework provides a map pattern that address both tiling and parallelism within a single mechanism. The map pattern divides a *gCollection* into multiple collections (multiple tiles), applies a *mapMethod* to each sub-collection (tile) and invokes the *reduceFunction* to join the generated/processed sub-collections (tiles). The *mapMethod* can be any method that iterates over a framework collection. The framework provides common implementations for split and reduce functions over *gCollections* (data tiles). Map operators can be

nested in order to implement multi-level tiling. In this case, different tiling orders can be quickly experimented by simply changing the nesting order of map operators.

Map operators can be executed in parallel by just changing the sequential map operator to a parallel map operator. In that case, the processing of sub-collections can be allocated to each thread in different ways, the framework currently offers block, cyclic and dynamic allocation.

The map pattern can generate additional data copies that can lead to inefficient implementations. The framework data API enables two mechanisms to improve performance of map operators: virtual collections and lazy copying. The framework offers several splitter and reducer functions in order to use these optimisations. One implementation creates a physical copy of the data, which can improve the spatial locality (an operation also known as packing). This implementation has the option to do the packing of all sub-collections when the split function is called or when the each sub-collection is accessed (i.e., lazy packing). Virtual collections avoid performing additional data copies by creating collections and iterators that are virtual views of the original data.

The map pattern is currently implemented by an high-order function introduced in Java 8 (i.e., a function that accepts pointers to methods). The map operator has the following interface: *MAP(splitFunc, mapMethod, reduceFunc, gCollection)*. Figure 4 illustrates the map pattern applied to the force method of our illustrative case. This case requires nesting to two map patterns. The *splitg2* divides the collection into virtual sub-collections. Each task calculates the interactions of all particles with first sub-collection, and repeats the process for other sub-collections. The *splitg1* divides again the collection and process the interaction of particles from sub-collections with the particles from another sub-collection.

```
parameters = gCollection.map(md::splitg1,
     (Object m) -> gCollection.map(
           md::splitg2, md:: force, md::reduceg2, m),
     md::reduceg1, parameters)
```

Fig. 4: Force computation applying tiling optimization

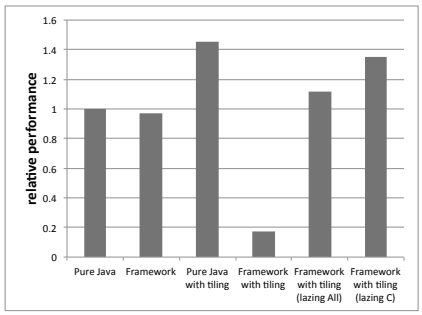
3 Performance evaluation

This section evaluates the framework performance on a server machine with two 12-core Xeon E5-2695v2 (NUMA memory), using OpenJDK 1.8.0_25.

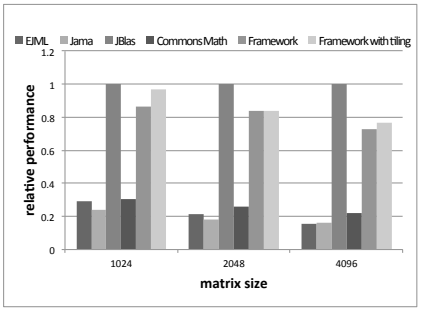
The first case study is a Matrix Multiplication, we compare code in plain Java with use of the framework. The kernel used is the same in both implementations, as well as the tiling optimisation. The Framework with tiling is based on the map operator described in section 2.2. Figure 5a shows the relative performance of tiling and lazy packing. The data API introduces a small overhead. In the plain Java implementation, tiling improves the performance, but with a traditional

map pattern implementation that advantage is lost. The introduction of lazy packing provides a performance comparable to a plain Java implementation. The figure also provides time for lazy packing of all matrices (A, B and C) and only for matrix C, which was tested by simply changing split functions.

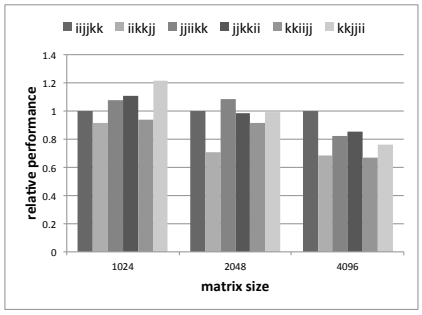
Figure 5b presents the relative performance of this version against other well-known pure Java implementations (the reference implementation is the JBLAS implementation). The framework provides the best pure Java implementation and up to 0.95 times the performance of the JBLAS implementation (JBLAS provides 0.70 times of the peak performance on this machine).



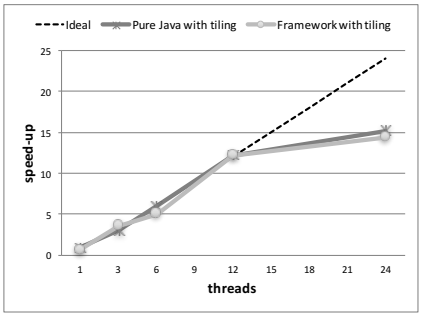
(a) Java vs Framework



(b) Performance of Java libraries



(c) Differents tile orders



(d) Parallel versions

Fig. 5: Matrix Multiplication benchmark

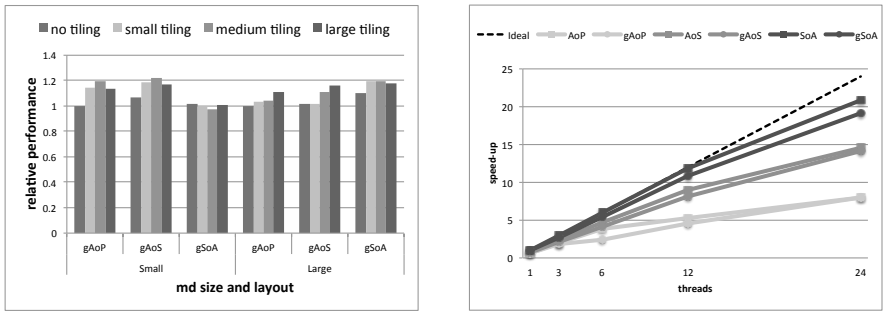
Tuning performance is a hard task and finding the best matrix implementation requires experimentation, since there are three nested loops in the Matrix Multiplication. In the framework, the experimentation can be quickly performed by adding a nested map. Figure 5c illustrates the relative performance using the *ti, tj, tk* order as reference, by changing the map nesting order. The best order also depends on the input size. In the framework this can be addressed by using different map nests for each input size.

A parallel version of the Matrix Multiplication is developed by replacing one map implementing the tiling with a parallel map. Figure 5d presents the speed-up obtained with this implementation and the comparison with an equivalent

implementation in plain Java. The performance of both implementations is very close and both scale linearly up to 12 processors. However, for 24 threads exists a performance penalty in both versions, caused by load unbalance (some threads process one more block than the others) and caused by the NUMA architecture..

The MD benchmark from the *JGF* (using an *AoP* layout) was implemented in the framework and tested with different data layouts (gXoX versions). The speed-up to the sequential *SoA* version (the more efficient) of each version is presented in Figure 6b. The *AoP* layout scales poorly due to the lack of data locality and because its sequential version is the slowest. The performance of the framework *AoP* implementation is similar to one of *JGF*. The *SoA* version provides the best performance.

The data layout can interplay with tiling. The framework provides a flexible mechanism to develop custom tiling approaches by implementing case-specific split/reduce functions for the map. A custom tiling approach is required for MD benchmark since there is a nested loop, where the inner loop depends on the current iteration of the outer loop. Figure 6a presents performance by composing the different data layouts of MD with tiling.



(a) Layout and tile (b) Parallel versions

Fig. 6: Molecular dynamic benchmark

For a small particle set, the *AoS* is the best layout and tiling improves the performance. Performance of *AoP* and *AoS* are very close for medium tiles. However, tiling does not improve the *SoA* version, which presents the worst performance. On the larger size, the *SoA* is the best version and, in this case, it benefits from applying tiling. Fortunately the framework provides an infrastructure where locality optimisations can be tested and implemented, which provides a fast mechanism to implement and study this kind of optimisations.

4 Discussion and Related work

There is a number of techniques to automatically improve locality by changing the *Java Virtual Machine(JVM)*. Hirzel et. al. [4] evaluated a technique based on sorting objects during garbage copying, which places objects in consecutive memory addresses to improve spatial locality. This technique still maintains the

AoP. Wimmer et. al. [11] propose an improvement to the *JVM* to automatically inline object fields by placing the parent and children objects in consecutive memory places and by replacing memory accesses by address arithmetic. Nie et. al [7] propose the Java vectorisation interface to explicitly expose data parallelism in programs enabling explicit vectorisation. These works require changes to the *JVM* implementation and there is no known system that supports data tiling.

OpenACC and Mint [10] are two programming frameworks that provide OpenMP like directives to support the loop tiling by a specific loop clause. The *Gaspar data-centric framework* provides a more flexible approach, for instance, it is easily change the tiling order or change the data layout.

The Java 8 parallel streams provide an API that resembles to map operators, but they are based on Java iterators and do not support data tiles and different layouts. The proposed framework uses a more sequential-like way of expressing map operators and is part of a larger effort to provide OpenMP-like programming in Java [6].

This paper shows that the map pattern is more suitable to express parallelism when the base program benefits from tiling [5]. The framework data API is similar to Standard Template Library(STL), but in STL there is a difference between a pointer (iterator) and the element pointed to. Thus, to access an element, pointer dereference must be used so it is not possible to automatically transform an iterator to the element pointed to. This will make it more difficult to encapsulate the data layout, without introducing a concept similar to interfaces in Java. Other approach is to hide the distribution and parallelism concepts in skeletons. STAPL[3, 12] and FastFlow[1] provide skeletons to simplify the code and improve performance. In STAPL, the skeletons express data distribution and parallelism, it is and based in STL iterators. Although, both frameworks do not support multiple data layout.

The map pattern of computation provides a safer way of iterating over data than using the traditional loop-based approach, since, in a map, the loop range is implicitly derived from the collection size. This avoids many potential errors of a loop-based approach, specially when multiple levels of tiling are required.

5 Conclusion

This paper presented a *Gaspar data-centric framework* and how the proposed data API efficiently supports multiple data layouts and tiling. The performance results show that the framework can provide implementations that compete with pure Java implementations. Thus, the framework provides improved trade-off between programmability and performance. The framework data *API* was also designed to make it easy to introduce locality improvements in loops that iterate over data in a collection. The provided infra-structure makes it easy to tests different data layouts and tiling, as well as to develop case-specific locality optimisations.

In future, the tool will provide a performance analyser helping the user to better layout choice. On the other side it will be included support to more platforms (example distribute memory).

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with fastflow. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011 Parallel Processing, Lecture Notes in Computer Science, vol. 6853, pp. 170–181. Springer Berlin Heidelberg (2011)
2. Anderson, J.M., Lam, M.S.: Global optimizations for parallelism and locality on scalable parallel machines. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. pp. 112–125. PLDI '93, ACM, New York, NY, USA (1993)
3. Buss, A., Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N.M., Rauchwerger, L., et al.: Stapl: standard template adaptive parallel library. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference. p. 14. ACM (2010)
4. Hirzel, M.: Data layouts for object-oriented programs. SIGMETRICS Perform. Eval. Rev. 35(1), 265–276 (Jun 2007)
5. Medeiros, B., Silva, R., Sobral, J.: Gaspar: a compositional aspect-oriented approach for cluster applications. Concurrency and Computation: Practice and Experience (2015)
6. Medeiros, B., Sobral, J.L.: Aomplib: An aspect library for large-scale multi-core parallel programming. In: Parallel Processing (ICPP), 2013 42nd International Conference on. pp. 270–279. IEEE (2013)
7. Nie, J., Cheng, B., Li, S., Wang, L., Li, X.F.: Vectorization for java. In: Network and Parallel Computing, pp. 3–17. Springer (2010)
8. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing. pp. 8–8. SC '01, ACM, New York, NY, USA (2001)
9. Smith, T.M., Van De Geijn, R., Smelyanskiy, M., Hammond, J.R., Van Zee, F.G.: Anatomy of high-performance many-threaded matrix multiplication. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International. pp. 1049–1059. IEEE (2014)
10. Unat, D., Cai, X., Baden, S.B.: Mint: realizing cuda performance in 3d stencil methods with annotated c. In: Proceedings of the international conference on Supercomputing. pp. 214–224. ACM (2011)
11. Wimmer, C., Mössenböck, H.: Automatic array inlining in java virtual machines. In: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization. pp. 14–23. ACM (2008)
12. Zandifar, M., Thomas, N., Amato, N., Rauchwerger, L.: The stapl skeleton framework. In: Brodman, J., Tu, P. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 8967, pp. 176–190. Springer International Publishing (2015)