Impact of Data Structure Layout on Performance

Nuno Faria
Universidade do Minho
CCTC
Braga, Portugal
Email: nfaria@di.uminho.pt

Rui Silva
Universidade do Minho
CCTC
Braga, Portugal
Email: rsilva@di.uminho.pt

João L. Sobral
Universidade do Minho
CCTC
Braga, Portugal
Email: jls@di.uminho.pt

***Abstract*—One key issue to design parallel applications that scale on multicore systems is how to overcome the memory bottleneck. This paper presents a study of the impact of data structure layouts in locality of memory references, providing insights on strategies to ameliorate the memory bottleneck. The paper compares the performance of Java and C++ STL collections and presents the impact of locality of reference optimisations in a molecular dynamics simulation case study. The case study shows that the selected data structure layout has impact on single core performance, becoming a critical factor in the application scalability on multicore systems. Moreover, data collections provided in the Java language compromise performance due to pointer chasing and lack of spatial locality of memory references.**

***Keywords*-locality; collections; multicore; Java;**

I. Introduction

The gap between CPU frequency and memory has increased over the last decades. Introducing multiple levels of memory hierarchy ameliorates the impact of this gap on performance. However, memory hierarchy is only effective if programs provide locality of reference in data access. Memory bottleneck is one main obstacle to performance scalability in many-core platforms since several cores share the available memory bandwidth. On the other hand, it is expected that the available bandwidth to access local caches will scale proportionally to the number of cores. Current platforms provide a fixed amount of L1 and L2 cache per core, independently from the total number of cores. Furthermore, there is a current trend to include a L3 cache shared among all cores, whose size and bandwidth scales proportionally to the number of cores.

Parallel programs should exhibit temporal and/or spatial locality of reference in data access in order to scale on many-core platforms. Data intensive applications are characterised by performing few operations per data item. Exploiting locally when these applications fall in the class of the so-called regular applications (e.g., matrix operations) is well known and usually resorts to partitioning data into blocks that can fit in cache [1]. Data intensive irregular applications that rely on pointer based data structures, such as graphs, are harder to optimise due to their intrinsic usage of pointers to access data and to their less-predictable pattern of data access.

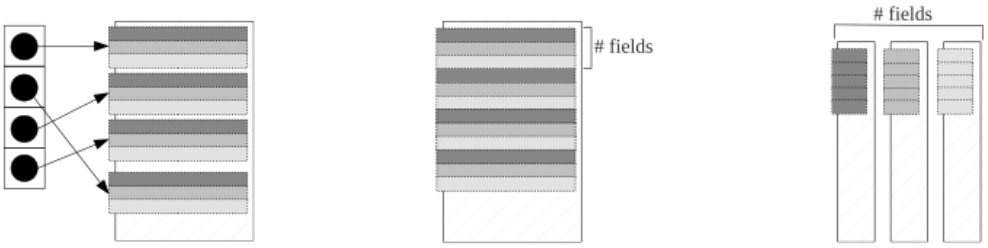
II. Data Structure Layout

An appropriate data structure layout can optimise the performance of a collection by improving locality of reference, for instance, by using knowledge about the access pattern made by an application. The Array of Pointers (AoP) layout (Figure 1a) is a popular layout due to its support for abstract data types. Memory references (or pointers to memory addresses) serve the purpose of being able to abstract the concrete type of the object pointed to. This kind of layout is usually adopted by systems with automatic memory management systems like garbage collectors in virtual machines (e.g., Java).

The AoS layout (Figure 1b) improves spatial locality by storing data fields continuously in memory, as in SoA, which stores fields into separate arrays. Choosing the best alternative between AoS and SoA depends on how the algorithm accesses the data. The SoA provides better locality if the algorithm does not require all fields of the original structure in the same time-frame. The AoS is the alternative used for problems that require all fields of the structure at once. The AoS layout is difficult to implement in Java since it is not possible to use explicit pointers to data. It is also more difficult to usse if the data fields are not of the same type. The AoP layout requires additional space to hold the array of pointers, when compared to AoS/SoA layouts, but provides more flexibility to manage the storage of data.

Several authors explored techniques to automatically improve locality in Java applications by relying solely on changes to the Java Virtual Machine (JVM). Hirzel et. al. [2] evaluate several improvements to data layouts in order to transparently improve spatial locality. The technique is based on sorting objects during garbage copying, which places objects in consecutive memory addresses. However, this technique still maintains the AoP layout and thus cannot avoid the overhead of pointer indirection.

Spatial locality in object collections can be improved by transforming an AoP implementation into an AoS or a SoA. In the latter case, the fields of the objects are converted into arrays, which normally involves removing the encapsulation of data. This provides better performance, but it might enforce significant restructuring of the code.



(a) Array of Pointers (AoP)

(b) Array of Structures (AoS)

(c) Struct of Arrays (SoA)

Figure 1. AoP, AoS and SoA views of attributes of structures in a collection

Wimmer et. al. [3] 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. The authors point out that using arrays as inlining parents is complex because the Java byte-codes for accessing array elements have no static type information. They claim that an automatic AoP to AoS transformation at JVM level is impossible without a global data flow analysis. Thus, automatically transforming AoP to SoA layouts seems not feasible at JVM level.

Automatically improving data layouts in languages with explicit pointers (e.g., C++) is not feasible due to the possibility of pointer arithmetic.

Temporal locality can be improved in all the presented layouts by tiling accesses to the object collection, dividing data objects into blocks (e.g., by performing a domain decomposition). If each block fits in cache and it is accessed multiple times, data will remain in cache and will be reused many times, decreasing the number of cache misses. In the case of AoP layout both the pointer array and data to must fit in the cache in order to exploit temporal locality.

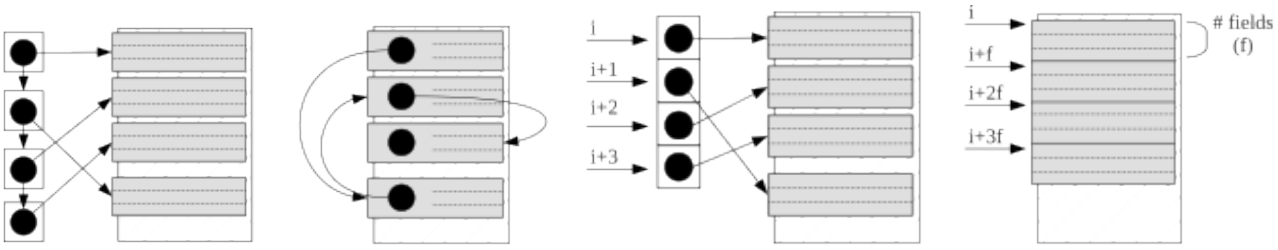
III. Java Generics and Boxing Problems

Java collections implementations add an implicit pointer-resolving layer to collections since they rely on AoP layouts. For instance, in array-based collections (e.g., ArrayList) elements are accessed via a memory reference and may not have the actual object elements in contiguous memory addresses. Moreover, primitive data types are boxed and accessed as an object.

The root of the problem is the Java designer's option to use a single collection implementation for all data types, enforcing the use of pointers to encapsulate the concrete type of the contained data. Thus, Java built-in collections use the AoP layout by default. The C++ Standard Template Library (STL) follows a different approach, since it relies on C++ template instantiation mechanism to generate a concrete implementation for each template instance (i.e., data type). Thus, the common case in STL is the use of AoS layouts. The drawback is the generation of bigger code since a different implementation is required for each data type. Note that STL collections also enforce considerable application restructuring to use a SoA layout.

IV. Benchmark methodology

Hardware performance counters available in modern processors provide valuable performance data for software optimisation. This paper assess the performance of data layouts using hardware performance counters gathered with the PAPI 4.2.1 library [4]. The ultimate metric is the number of clock cycles (#CC, measured with the PAPI_TOT_CYC event), since it is directly proportional to the execution time. The number of completed instructions (#I, gathered with PAPI_TOT_INS) is an estimation of the implementation complexity, whereas the number of references to the L1 data cache (L1.REFS, PAPI_L1_DCA) and L1 misses (L1.MISS, PAPI_L1_DCM) reflect the memory behaviour. Performance data was gathered on a machine with dual X5650 processor (i7, 6-core architecture, 2.66 GHz with 12 MB of shared L3 cache per processor), running CentOS 6.2. Results in Java where collected with the OpenJDK 1.6.0_22, in 64-Bit Server mode (it provides better results than the recent Oracle JRE 1.7.0_5). C++ results where collected with Intel compiler icpc 12.1. Presented results are the median of five executions, collected after an additional execution for warm-up (this is essential in Java due to the JIT approach). In general only misses in L1 data cache are presented since the second level and third level show similar trend-line. Most results are gathered in Java and C++. In this study it was surprising to discover that in most benchmarks the obtained metrics for #CC and L1.MISS are pretty close in Java and C++, even in the presence of huge differences in #I and L1.REFS (generally higher in Java). This is due to the memory-bound nature of our case studies, with instruction execution rates largely bellow the peek performance. Thus, inefficient code with more #I and L1.REFS are easily accommodated by the unused execution units. #CC and L1.MISS on these case studies are much more dependent on the effectiveness of the machine memory hierarchy subsystem, which is the same for C++ and Java.



(a) LLoP (b) LLoS (c) ALoP (d) ALoS

Figure 2. Array-List and Linked-List representations considered

V. Impact of Collections Data Layout

Java and C++ STL collections can be classified into four broad classes of data layouts: Vectors, Lists, Trees and Maps. This study focus on Lists and Vectors, by testing standard Linked-Lists (LL) and Array-Lists (AL) collections against optimised data structures (Figure 2). In Java, LinkedList and ArrayList collections enforce the storage of pointers to objects (e.g., LLoP and ALoP layouts). An optimised Linked-List (LLoS) which is similar to a list container in C++ STL was developed for this study. In C++ the LLoP and ALoP layouts are simulated by using collections of pointers (e.g., list and vector of pointers to data structures). In C++, a vector uses the ALoS layout, whereas in Java this layout is simulated by using arrays of primitive types. Note that Java native collections do not support the LLoS layout and the ALoS layout is only possible when using arrays of primitive data types.

Figure 2a (LLoP) shows why Java collections introduce "unnecessary" pointer-resolving operations: one pointer to the object and another pointer to the next element in the collection. In the optimised Linked-List implementation (Figure 2b, LLoS) the object in the collection contains the pointer to the next element in the list; this optimisation eliminates the redundant pointer between the list element and the data object. As for Array-Lists, in Figure 2c (ALoP), although there is contiguity in the memory addresses of the pointers to the objects, since it is an indexable structure, there is pointer-resolving operation in order to get the object data. The methodology applied in the last figure (ALoS) is the same as previously showed: elimination of the pointer-resolving operations to objects.

To asses the performance characteristics of those collections this study performed a benchmark that sums all elements in the collection. The main goal of the test is to verify the overhead of indirection levels in collections and to compare the performance of Java and C++ in data intensive applications. This benchmark is a data intensive test case since each element in the collection is accessed only once. On the other hand, the test case is amenable for the exploitation of spatial locality, since data elements in the collection are accessed by their logical order.

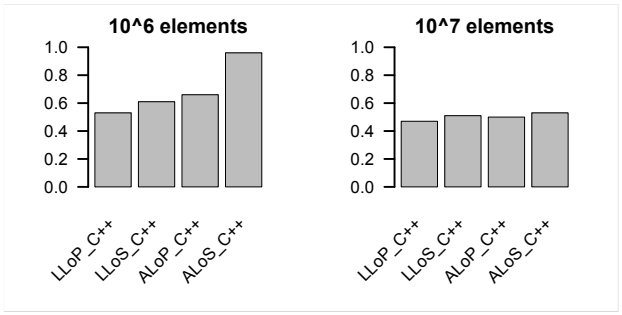


Figure 4. Sum bechmark weak scalability

Figure 3 shows benchmark results for the sum of 10^7 elements (integers):

- The optimised linked-list (LLoS) provides better performance (i.e., less clock cycles) when compared against Java's native linked-list (LLoP). This is due mainly to better spatial locality (lesser L1 data cache misses), resulting from a more effective usage of cache lines, since both the data and pointer to next datum can fit into the same cache line.
- LLoS and ALoP show similar memory behaviour in terms of cache misses since both involve a single pointer indirection, in LLoS to access the next element and in ALoP to access to the element pointed to. However, ALoP performs better in terms of clock cycles (almost half execution time) since there is no dependence among loop iterations as in any Linked List. This allows the processor to dynamically unroll the loop, partially hiding the latency of cache misses.
- The array version (ALoS) has lowest cache references as well as misses, indicating that one of the major overheads in previous implementations is due to pointer resolving operations - here the number of cache references approximates the number of elements in the list to be processed (10^7 elements) and the number of misses approximates the number of cache lines that must be fetched from lower memory levels (1/16 of memory references), meaning that there is little overhead in this representation.

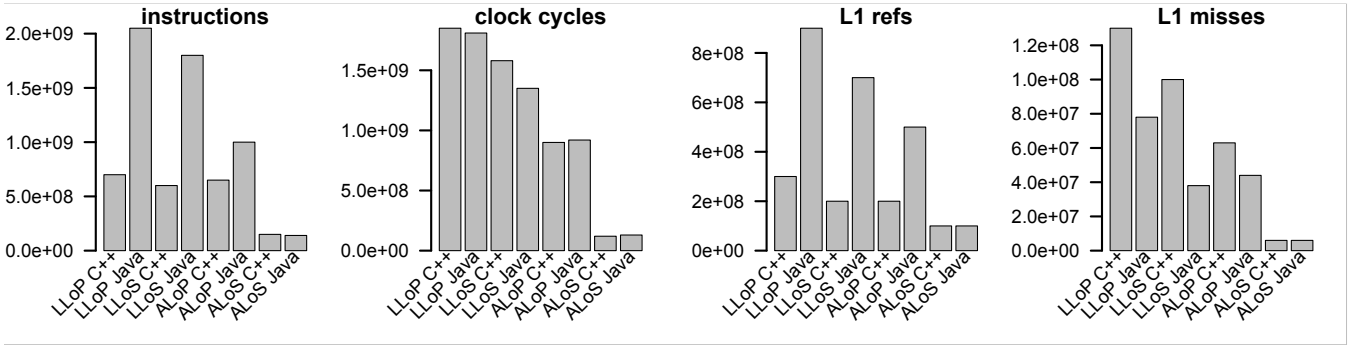


Figure 3. Performance of Array-List and Linked-List representations in the sum benchmark.

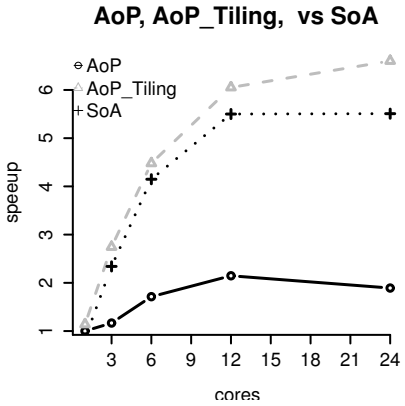


Figure 6. MD scalability.

- When comparing C++ and Java implementations we see that the actual performance is similar in terms of clock cycles and L1 misses despite the fact the the Java version has much higher number of instructions and L1 references. This is due to the data intensive nature of this case study as explained before.

One important question is how these layouts impact application scalability. Figure 4 shows the performance on 12 cores (maximum available on the test machine). The results present the performance by running the test case with a 12-fold increase in the data set (usually called weak scalability). The figure present results for summing 10^6 and 10^7 elements per core. The former data size is smaller than the L3 cache size. From these results it can be observed that the performance drop off is proportional to number of misses in L1 data cache. Thus, the alternatives consuming more cache/memory bandwidth also present the worst weak scalability (note that figures presents the speed-ups, which are less than 1). Actually only the ALoS layout scales reasonably well and only for the data set that fits in the L3 cache. Thus, in addition to presenting the lowest sequential execution time the ALoS is also the version which scales better due to better locality of memory references.

VI. Molecular Dynamics Simulations

Molecular dynamics (MD) is a technique of computer simulation where a set of particles (atoms) interact during a certain period of time. For each pair of particles there is an interaction (force). Initially, positions and velocities are assigned to the particles in the domain. Then, the forces acting on each particle are computed and the resulting particle acceleration. The new position for each particle is then computed based on the particle velocity and on the time step. The process is repeated for a given number simulation steps. The most time consuming part of the simulation process is the force computation [5] which involves a nested loop across all pairs of particles.

Figure 5 show a performance comparison of three different basic MD simulation versions using different data layouts. The first version is the one provided in the Java Grande Forum (JGF) MD benchmark [6] which implements the collection of particles with an array of pointers to objects (Particles in this case). The AoS version results from the adapation of the JGF MD benchmark to store particle information in a single array, where the information of the particle i is stored in the 9 ' i position of the array (each particle has nine fields: position in x, y and z axis, velocity/force in x, y and z directions). The SoA version stores each particle field in a different array.

The SoA version provides the shortest runtime. The improvement results from better locality of references (less L1 cache misses). The AoP version provides the fewer number of instructions since it avoids data copies required in other versions. Despite this, it performs worse than the SoA version, due to less locality of data accesses. The AoS version is not attractive, since particle fields are not used all at once in this case study. The graph also shows results by improving temporal locality using a traditional tilling approach (versions tagged with a M). It can be seen that only the AoP and AoS performance benefits from this improvement (clock cycles results), since SoA already has good locality (note that this performance is explained by better usage of L2 and L3 levels of cache).

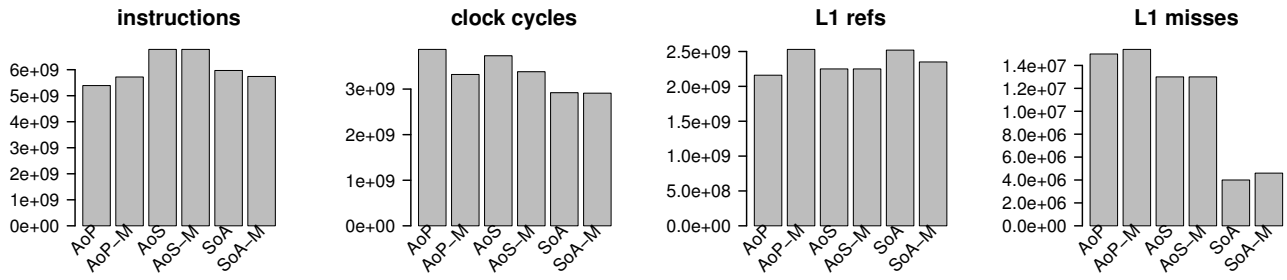


Figure 5. Performance of MD implementations.

Figure 6 compares the scalability of the SoA version against the base JGF AoP implementation and a version exploring temporal locality (Tiling). The better locality of the SoA version provides better scalability, but the AoP with Tiling also performs well. Note that this machine is a dual processor, each one with 6-cores and hyper-threading. This explains why the performance improvements in SoA do not increase linearly when using the second processor (i.e., move from 6 to 12 cores) and the lower improvement when using 24 cores (hyper threading provides 24 virtual cores mapped into 12 physical ones). Interestingly, the base JGF MD (AoP) benefits from using the second processor, since each processor provides a memory bank (i.e., NUMA configuration) and since this version lacks of locality it can slightly benefit from the additional memory bandwidth provided by the second processor. It is worth to note that the lack of scalability of the base JGF (AoP) implementation is only observed on modern multicore machines.

VII. Conclusion

This paper identified the systematic use of AoP layout as one main source of overhead in Java collections. This overhead is mainly due to pointer indirection and to the lack of spatial locality in data access. This overhead can be the main limitation to scalability on multicore systems, for both Java and C++ STL collections.

This paper showed that SoA and AoS data layouts are effective strategies to improve spatial locality, which provides better application scalability on multicore systems. AoS/SoA layouts may enforce an increase on the number of instructions and more lines of code (around 15% in our case studies).

The presented study focused mainly read-only data structures, but the proposed techniques can also be applied to mutable data structures by using hybrid layouts.

Acknowledgments

This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundac a o para a Ciencia e a Tecnologia (Portuguese Foundation for Science and Technology) within projects FCOMP-01-0124- FEDER-010152 and FCOMP-01-0124-FEDER-011413.

References

[1] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/1248377.1248394

[2] M. Hirzel, "Data layouts for object-oriented programs," in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '07. New York, NY, USA: ACM, 2007, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/1254882.1254915

[3] C. Wimmer and H. Mössenböck, "Automatic array inlining in java virtual machines," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 14–23. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356061

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000. [Online]. Available: http://dx.doi.org/10.1177/109434200001400303

[5] S. Kumar, C. Huang, G. Almasi, and L. V. Kalé, "Achieving strong scaling with namd on blue gene/l," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 61–61. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898953.1898995

[6] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, R. A. Davey, "A Benchmark Suite for High Performance Java," *Proceedings of the 199 ACM Java Grande conference*, pp. 81–88, 1999.