Enhancing Locality in Java based Irregular Applications

N. Faria, R. Silva and J. L. Sobral

{nfaria, ruisilva, jls}@di.uminho.pt
CCTC/Universidade do Minho

Abstract. Improving locality of memory accesses in current and future multi-core platforms is a key to efficiently exploit those platforms. Irregular applications, which operate on pointer-based data structures, are hard to optimize in modern computer architectures due to their intrinsic unpredictable patterns of memory accesses. In this paper we explore a memory locality-driven set of data-structures in order to attenuate the memory bandwidth limitations from typical irregular algorithms. We identify the inefficiencies in the standard Java implementation of a priority-queue as one of the main memory limitations in Prim's Minimal Spanning Tree algorithm. We also present a priority-queue using the data layout inspired in Van Emde Boas for ordering heaps. We also implement optimizations in the graph data-structure and explore ways to efficiently combine it with the memory-efficient priority-queue. In order to improve efficiency in both case studies we had to transform the data-structures in the form of array of pointer into arrays of structures or structure of arrays.

1 Introduction

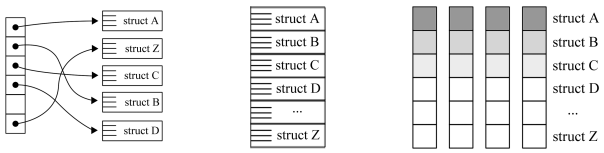
The gap between CPU frequency and memory bandwidth has been increasing over the last decades. Introducing multiple levels of memory hierarchy ameliorates the impact of this gap on application performance. Although, memory hierarchy is only effective when programs provide locality in data access. Memory bottleneck will also be a big hurdle in many-core platforms since several cores usually share the available memory bandwidth, limiting the application scalability. On the other hand, it is expected that the effective bandwidth in accessing local caches will scale proportionally to the number of cores. Current platforms provide same fixed amount of L1/L2 cache per core, independently from the total number of cores. Thus, to effectively use current and future many-core platforms programs should present temporal and/or spatial locality in data access. Exploiting locally in the so-called regular applications (e.g., matrix operations) is well known and usually resorts to partitioning data into blocks that can fit into the cache [1]. Irregular applications that rely on pointer based data structures, such as graphs, are harder to optimize due to their intrinsic usage of pointers to access data and to their less-predictable pattern of data access. One typical case is the Prim's algorithm to compute a Minimum Spanning Three of a

*This research was supported by the Fundação para a Ciência e a Tecnologia (project Parallel Programming Refinements for Irregular Applications, UTAustin/CA/0056/2008)

graph: the particular order of traversing the graph vertex depends on the weigh of the edge connecting vertexes. In generic frameworks, the effort to provide collections that can be used in multiple context leads to data accesses through additional pointer indirection. In this paper we identify this indirection as one main source of overhead both due to pointer indirection and lack of locality of memory accesses and explore strategies to remove this overhead.

2 Locality driven data layout

Many Java collections are implemented as Arrays of Pointers (AoP) to structures (Figure 1a). This data layout does not provide spatial locality, but it is still possible to exploit temporal locality. A common optimization for better temporal locality is the division of the array 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 can be reused many times lowering the cache miss rates. Optimizations to improve spatial locality need a different data layout, in which elements are contiguous and accessed in an orderly manner. In this paper we study three different data layouts (Figure 1): (*i*) arrays of pointers (AoP), (*ii*) arrays of structures (AoS) and (*iii*) structure of arrays (SoA). In the AoS layout, fields are stored continuously in memory, as in SoA, fields are stored into a separate array. Choosing the best alternative depends on how the algorithm accesses 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, although this choice is difficult to implement in Java since it is not possible to use explicit pointers to data. It is also more difficult to use if the fields are not of the same type. We may also use an hybrid alternative, in which the data fields are grouped according their time-frame usage. To improve the spatial locality in Java collections it is necessary to transform an AoP implementation into an AoS or a SoA. In the latter case, the fields of the objects are converted into arrays, which normally evolves removing the encapsulation of data. This provides better performance, but it might enforce significant restructuring of the code. In this work we intend to study the impact on performance of this transformation. Several authors propose techniques to automatically improve locality in Java applications by relying solely on changes to the Java Virtual Machine. Hirzel et.



(a) *AoP* (b) *AoS* (c) *SoA*

Fig. 1: *AoP*, *AoS* and *SoA* views of attributes of structures in a collection

al. [2] evaluates improvements in several data layouts, by sorting objects during garbage copying to improve locality, which can do so by placing target objects in consecutive memory addresses, but it still maintains the AoP layout. Wimmer et. al. [3] proposes an improvement to the JVM to automatically *inline* object fields by placing the parent and children in consecutive memory places and by replacing memory accesses by address arithmetic. It is argued that using arrays as inlining parents is complicated because the Java byte-codes for accessing array elements have no static type information. Thus they claim that automatic AoP to AoS transformation at JVM level is impossible without a global data flow analysis because of the structure of the array access byte-codes. Furthermore, transforming AoP to SoA layouts also seems not feasible at JVM level.

3 Case studies

To illustrate the impact of cache-efficiency in data-structures and algorithms in Java we study a few applications that would benefit from those optimizations. We show how we optimize the cache-hit behaviour for *priority-queues* (PQ) and graph structures for the Prim's Minimal Spanning Tree (MST) algorithm. For each PQ and graph implementation, were generated a random sequence of numbers (PQ) and a random graph, both stored in file so that every PQ and graph implementation uses the same memory access pattern. All results presented were measured with PAPI profiling tool[1], integrated using Java Native Interfaces. In order to only measure program sections we were interested in (for PQs, insertion and removal only, ignoring file computations), we start/stop PAPI measurements at the chosen routines. The tests were ran in a cache environment of L1 32 KB instruction caches + 32 KB write-back data caches and shared L2 of 4 MB; the CPU is an Intel Core 2 Duo Mobile; the installed JVM is Java HotSpot (TM) Server VM (version 1.6.0_24).

3.1 Cache efficient priority-queues

Priority queues (PQ) are built for fast retrieval of the highest/lowest element in a set of elements. Java's native priority queue is based on the structural layout of a binary heap [4]. The locality driven optimizations were applied on PQs avoiding use the *Comparable* interface. Comparing, for instance, Integer objects with explicit mathematical operators ($<$, $>$) and with the Comparable interface incurs in similar results, due to the auto-boxing/unboxing[2] - comparing reference data-types (e.g., Integer) with explicit comparators causes the auto-unboxing of

[1] A hardware counter-based profiling tool: http://icl.cs.utk.edu/papi/faq/index.html

[2] Boxing is the phenomena of converting a primitive data-type value able to be explicitly stored and represented in memory into a referenced data-type value, i.e., the value is no longer in raw format (*int*, *float*, *double*, etc.) since it is now a reference to a memory location where the value is stored. Unboxing is the opposite, the conversion of a reference data-type value into a primitive one. Most OO frameworks, like Java, do this automatically for primitive data-types.

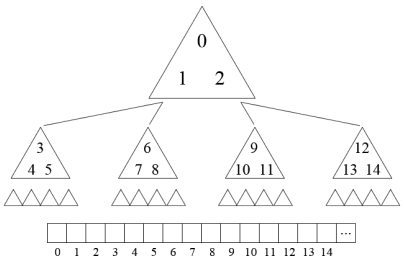


Fig. 2: The VEB data layout - numbers correspond to array indices

the referenced *int* value. However our main reason to use explicit primitive data types is due to the notion of boxing - creating references to memory addresses makes locality optimizations obsolete. After a quick analysis (see Table 1, first three columns) it is visible that forcing element adjacency in Java collections (*BH int*) substantially improves performance - less instructions were needed, cache miss[3] are also lower and the execution time is approximately three times shorter. Using primitive data-types *forces* heap elements to be directly accessible from the array (using a SoA layout), as for an array of *Objects* (i.e., AoP layout) we must first resolve the pointer to the *Object* in order to access it; so for each array access (in *int* version) pointer resolving is unnecessary - *BH int* runs in less than half the instructions of *BH Integer*[4]. Also, not resolving pointers ultimately results in less cache misses, which in its turn results in less *stall cycles*, thus despite a reduction in 2 times on the number of instructions we attain an overall improvement in execution time of 3.3 times, mainly due to a bigger improvement in locality of memory addresses (6 times less misses in L1).

Locality aimed heap. To improve spatial locality we implemented a variation of a *binary-heap* with a more cache-friendly data layout - a simplified layout version of Van Emde Boas (VEB) [5] (Figure 2). On Table 1 we compare *BH int* implementation against *VEB* for block sizes of 3 and 7 (i.e., block height of 1 and 2, respectively). We see a slight decrease in the execution time in *VEB 3 int* compared to *BH int*, we are able to achieve better cache performance. The increase in instructions for VEB is explained with the increase in complexity in children and parent index expressions.

3.2 Minimal Spanning Tree algorithm problem

The MST algorithm we focus on for this case study is Prim's MST algorithm [6] which uses a priority-queue to find the smallest weighted available edges to add

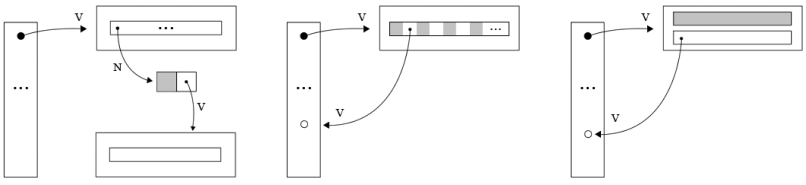
[3] We opted to use the number of cache misses since it is more independent from compiler optimizations than miss rates (e.g., poorly optimized code tends to show lower miss rates due to unessential memory loads, mainly due to register spilling).

[4] BH Integer is our PQ implementation (using Java's Integer class) similar to Java native PQ by using an AoP approach, it was developed for comparison purposes.

Table 1: Benchmarks comparing *binary-heap* to Java's native PQ (AoP), and to *VEB-heap*, storing single *ints*

	Java Integer	*BH Integer*	*BH int*	*VEB 3*	*VEB 7*
Instructions ($\times 10^8$)	132.02	153.06	73.29	74.78	96.42
Cycles ($\times 10^8$)	199.81	192.87	58.16	55.94	73.26
L1 accesses ($\times 10^8$)	69.22	79.68	28.14	35.62	48.82
L1 misses ($\times 10^6$)	234.65	232.42	39.51	28.50	22.76
L2 accesses ($\times 10^6$)	418.60	412.87	93.97	70.63	57.76
L2 misses ($\times 10^6$)	279.47	272.23	50.85	39.51	30.57
Time (s)	10.025	9.579	2.932	2.812	3.736

to the resulting MST graph. We studied the cache-hit/miss rates in the graph representation, a typically irregular structure. In Figure 3 we represent some of the graph representations studied. The encapsulation problem is more visible in this case for *weighted* graphs, when referring to the concept of *Neighbours*. Our implementations all follow a similar interface - (*i*) a set of *Vertices* (an Object), (*ii*) in which each *Vertex* has a set of neighbours (set of objects, *Neighbour*), (*iii*) each *Neighbour* harbouring the attributes: *weight* (primitive data-type), *neighbour-vertex* (pointer to *Vertex* object) and possibly other attributes, but for simplicity we consider only these. We distinguish the different graph implementations being AoP (Array of Pointers), AoS or SoA by looking at how the *Vertex* structure holds the *Neighbour* array. The process of decapsulating Neighbour attributes reduced redundant object instantiations between the graph and PQ structure, with the MST layer as an intermediate layer. We opened the implementation box, by changing the program's API in order to work directly with the attributes, instead of encapsulating them in objects. Thus, instead of returning from the graph data structure and adding to the PQ a Neighbour-object with the *weight* and *vertex* attributes, we return and add these attributes separately, to avoid redundant object instantiation, ultimately benefiting cache-hits. As stated in Table 2, the best execution time as well as, instruction, cycle and cache behaviour occur for the *GVG AoS* and *GVG SoA* implementations. The major jump in performance is mainly due to the redundant pointer resolving operations present in creating new object instances that has now been removed from the executions in AoS and SoA graph implementations. Although the cache



(a) Graph-Neighbour-Vertex (GNV_{AoP}) (b) Graph-Vertex-Graph (GVG_{AoS}) (c) Graph-Vertex-Graph (GVG_{SoA})

Fig. 3: Graph representations

Table 2: Benchmarks comparing all three graph implementations (*GNV AoP*, *GVG AoS* and *GVG SoA*), PQ implementation is *VEB 3*, ran with the cases AoS and SoA.

	$VEB\ 3_{AoS}$			$VEB\ 3_{SoA}$		
	GNV_{AoP}	GVG_{AoS}	GVG_{SoA}	GNV_{AoP}	GVG_{AoS}	GVG_{SoA}
Instructions ($\times 10^8$)	9.71	6.07	6.01	9.90	6.38	6.19
Cycles ($\times 10^8$)	15.40	4.87	4.78	15.52	5.08	5.00
L1 accesses ($\times 10^8$)	5.94	2.98	2.80	5.95	2.90	2.79
L1 misses ($\times 10^6$)	9.41	1.53	1.54	9.39	1.60	1.62
L2 accesses ($\times 10^6$)	12.99	3.96	3.95	12.84	3.94	3.92
L2 misses ($\times 10^6$)	3.14	1.74	1.73	3.15	1.88	1.87
Time (s)	0.776	0.299	0.302	0.784	0.297	0.299

miss behaviour in *GVG AoS* is slightly better than in *GVG SoA*, the later shows a little less running time, because it runs in less instructions and it consumes less cycles.

4 Conclusion

In this paper we presented the impact of data layout on application performance. We identify the use of array of pointers 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. We explore the use of two alternative data layouts - array of structures and structures of arrays - that showed performance improvements in the case studies. However, effective usage of these optimized data layouts in modern object-oriented frameworks require component decapsulation in order to avoid additional object creations and data copies. In the future we plan to investigate techniques to perform locality-driven optimizations to data layouts without compromising encapsulation.

References

1. Yotov, K., Roeder, T., Pingali, K., Gunnels, J., Gustavson, F.: An experimental comparison of cache-oblivious and cache-conscious programs. Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures SPAA 07 (2007) 93
2. Hirzel, M.: Data Layouts for Object-Oriented Programs. In: International Conference on Measurement and Modeling of Computer Systems SIGMETRICS. (2007)
3. Wimmer, C., Mössenböök, H.: Automatic array inlining in java virtual machines. Proceedings of the sixth annual IEEEACM international symposium on Code generation and optimization CGO 08 (2008) 14
4. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM **7** (1964) 347–348
5. Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory **10** (1976) 99–127
6. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technical Journal **36** (1957) 1389–1401