University of Texas
Institute for Computational Engineering and Sciences
Center for Distributed and Grid Computing

Summer Internship

Improving Locality on Traversal of Recursive Structures

Miguel Palhas mpalhas@ices.utexas.edu
Pedro Costa pcosta@ices.utexas.edu
Andrew Lenharth lenharth@ices.utexas.edu
Donald Nguyen donald.nguyen@gmail.com

Austin, Texas, U.S.A. — August 2012

Abstract

Locality in algorithms using irregular structures is naturally difficult to improve. This document reviews a new approach to these algorithms in the form of two optimizations, which change how the structures are traversed. Three implementations of example algorithms are also described in this document: Barnes-Hut, Point Correlation and Ray Tracing. These examples are parallelized using the Galois framework. Measurements showed that only the first two examples got speedups, but all reduced cache miss rates considerably for the first two levels.

1. Introduction

This document follows the work described in [1], where a framework called TreeTiler is presented, to optimize, at both compile time and run time, locality of algorithms relying on the traversal of recursive data structures.

In the following sections, the optimizations implemented in the TreeTiler framework are reviewed and applied to three test cases. These cases were selected to match the experiences performed in [1] and use the Galois framework[2] for parallelization. The goal of this document is to study the impact of these optimizations along with Galois.

Section 2 explains the original work and the *TreeTiler* framework, with a particular focus on the implemented optimizations in section 2.1. In section 3 the implemented test cases and each of their particularities are explained. Measurement results are all presented in section 4 with some final considerations and conclusions in section 5. Appendix A detail the testing environment used to achieve the results shown.

2. TreeTiler

The work showed on [1] presents a framework, *TreeTiler*, that can be used to refactor Java code at runtime. Given a Java application source code as input, *TreeTiler* will try to detect a recursive data structure. It will then look for a recursive method that performs a recursive traversal on that structure.

The JastAdd framework is employed to analyze and refactor the Java input at compile time, generating the output consisting of the same program with the applied optimizations, which are described here.

2.1. Optimizations

The techniques employed by *TreeTiler* deal with the way a recursive structure is traversed. Even though it is commonly an irregular structure, some techniques can be applied to exploit locality when multiple traversals are done. In this report, the techniques used were the Spatial Sort and Point Blocking. An Auto Tuner is also added to the Java code by *TreeTiler*. This Tuner tries to find the best parameters for the optimizations, which sometimes are input dependent and cannot be evaluated at compile time.

2.1.1. Sorting

Traversing an irregular structure requires loading from memory the elements of the structure which, due to their irregular nature, can not be obtained in an optimized way through prefetching. Also, the elements of this structure which are consecutively accessed (especially in parallel) will most probably lie in distinct cache lines, hurting locality.

By changing the order of accesses in these structures, so that domain objects with similar traversal patterns become consecutive, temporal locality is improved as the elements a given object will touch are already likely to have been touched by the previous one.

Sorting these objects in applications where the domain refers to a notion of space can be performed using space-filling curves[1] (for algorithms in the space domain like Barnes-hut). The examples described in this document use the CGAL library, implementing the traits required for spatial sorting in 3D.

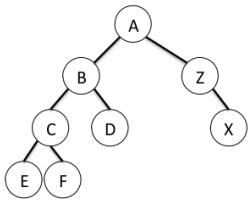

Figure 1: A sample tree

Limitations: Processing objects in an order which brings similar traversals closer in time only brings improvements when the traversal itself fits in cache. This will allow subsequent objects to take advantage of the acceleration structure elements already visited by previous objects. For sufficiently deep traversals, when the subsequent objects traverse the structure, the first elements have already been evicted from cache. [1] already shows that as the traversal size increases, the locality improvements gained from the Sorting optimization decrease drastically.

[1] A curve which touches every object in the domain only once. Examples of these curves are the Peano and Hilbert curves.

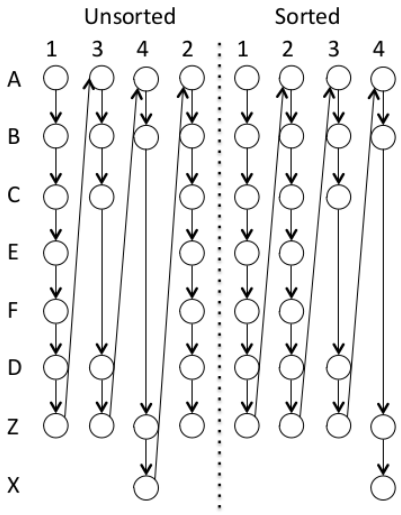

Figure 2: Spatial Sort transformation applied to traversal of tree shown in fig. 1

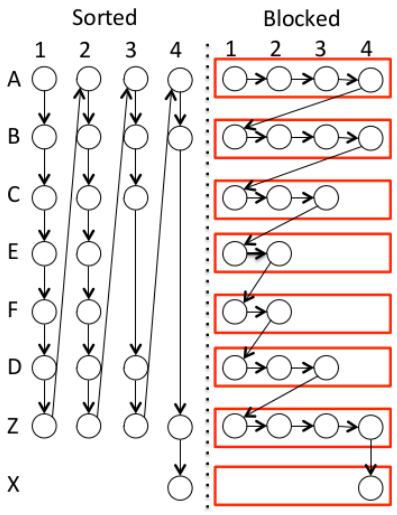

Figure 3: Point Blocking transformation

2.1.2. Blocking

Since it is likely that geometrically closer points also follow a similar traversal, which is the basis for the Spatial Sort optimization, it is also likely that those points can be processed in blocks, rather than one at a time, thus taking advantage of locality, and preventing the traversal from being evicted too soon from cache.

Instead of traversing the structure for a single point, and iteratively switching points, the new strategy takes advantage of the premise that Spatial Sort rearranges points so that a small subset of points is likely to be geometrically close to each other. This subset is grouped into a block, and the tree is traversed for that block at the same time.

The root node of the structure is processed for all points of the block. If necessary (depending on the algorithm), if the paths of each point in the block diverge at a given node, it may be necessary to split the block into smaller blocks, each one going to a different node. Given the geometrical proximity, this is only more likely to occur at the last levels of the structure, allowing for some locality to still be gained in the previous levels.

Figure 3 illustrates the execution order of a point blocked implementation, compared to the non blocked, sorted implementation.

It is intuitive to notice that this strategy is only good if the points within a block are likely to follow a similar traversal path, otherwise too much divergence will hinder performance much like the original strategy.

2.1.3. Auto Tuner

The optimal block size is dependent on both hardware and algorithm details, but it may also be input dependent. The optimal block size for a Barnes-Hut implementation might not be the same as the ideal block size for a Ray Tracing application. Given that, a generic framework like *TreeTiler* must not make assumptions about the algorithm and the input only at compile time.

For that reason, *TreeTiler* installs an AutoTuner in the output code. This AutoTuner will perform a small sample (possibly selected randomly) of block iterations, and evaluate their performance at runtime. Typically, it is expected that a block size too small will yield bad results, possibly even worse than the original version, due to large overhead of the additional instructions for the point blocking, and because of the fact that misses in the tree occur between each block. In contrast, a block size too big will not fit in cache, and misses will occur in the block items instead.

A sweet spot is expected to occur where the block size is near ideal value, meaning the block size is large enough to avoid cache misses in the tree, but still small enough to fit in cache.

The AutoTuner might then start at a small block size, and process the chosen samples, measuring the average traversal time for each subsequent block size. The time is expected to reduce until the sweet spot is reached, at which point the time will start to increase again. When this happens, the best block size is the one that yielded the lowest runtime.

3. Case Studies

The studied optimizations, previously described in section 2.1 were applied to three different algorithms. The work focused mostly on the Spatial Sort and Point Blocking techniques. AutoTuner was not implemented, as this was not a generic framework like *TreeTiler*, and as such there was no need for a tool to automatically determine the ideal parameters. The implemented test cases were:

Barnes-Hut This was the first test case, based on the implementation provided on the sample applications from the Galois framework. The acceleration structure used here was the already provided Octree.

Point Correlation Implemented from scratch. In terms of implementation, it is mostly equivalent to that of the Barnes-Hut example, with the biggest difference being in the acceleration structure used, which is a kD-Tree rather than an Octree;

Ray Tracer Based on a simple unoptimized implementation in OpenMP. A Bounding Volume Hierarchy implementation was designed for the acceleration structure.

Sections 3.1 to 3.3 explain with some more detail each test case.

3.1. Barnes-Hut (BH)

For the first test case, the Barnes-Hut implementation provided in the sample applications of the Galois framework was used.

This implementation used an Octree as the acceleration structure, and served as a starting point to the usage of Galois. Only the optimizations were left to implement.

An Octree structure divides the space equally in eight subspaces recursively. In the Barnes-Hut algorithm, it allows the contribution of a set points to be estimated by the contribution of the subspace they belong to, given that the distance between the point and the subspace is enough to make the contribution of each individual body almost neglectable.

As stated in section 2.1.1, the CGAL library was used to implement the spatial sorting of the bodies according to their coordinates in the space.

To apply Point Blocking, the traversal of the Octree was changed in order to accept a block of bodies (in contrast to a single body in the original implementation). During the traversal, each node of the tree would evaluate which bodies would need to descend further in the structure, and a new block containing only those proceeds.

3.2. Two Point Correlation (PC)

The Two Point Correlation algorithm is in many ways similar to the Barnes-Hut. This application counts the pairs of points in a space which rely within a given radius of each other. It differs from the previous example in the acceleration structure – a kD-Tree – and in the need for a reduction to sum the pairs counted for each point.

Since this algorithm does not require any special data to be stored in the acceleration structure (the Barnes-Hut requires that each node gives an estimation of its internal data), a simpler structure may be used. A kD-tree is a binary tree: in each level, the points are ordered by one of the coordinates, and the median is placed as a divider. In the next level, another coordinate is used. This algorithm uses a kD-Tree to automatically exclude all the points in a given subspace if the closest point in the subspace is already too far away.

Like with the Barnes-Hut, the sorting optimization was implemented using CGAL, and Point Blocking required a change in the tree traversal to accept a block of points at a time. At any node of the tree, if a point is too far way, it is excluded from the block.

The total count of correlated pairs is obtained using a thread local accumulator (available in Galois), where each thread adds the count of the processed blocks.

3.3. Ray Tracer (RT)

This was the most different and challenging test case to implement. The original source code, "*smallpt: Global Illumination in 99 lines of C++*", is explained and available in [3].

The original implementation of `smallpt` relied on OpenMP to distribute workload accross threads. The workload consisted in the pixels, meaning that each thread would be responsible for a subset of the total pixels of the image. For each pixel, the corresponding thread would launch a given number of rays (given as a parameter) using Monte Carlo to add some randomness to their direction. Each ray was then checked against the entire list of objects composing the scene, which was implemented as a naive vector of spheres [2].

After each collision, one or more subrays might be generated, depending on the properties of the object with which the ray collided. The ray might generate a reflected or refracted ray, or a combination of both. After the path of rays reaches a given depth (also given as a parameter), Russian Roulette was employed to decide whether to stop the path. Each ray computes a contribution to the pixel it belongs to, which is inversely proportional to the depth of the ray.

3.3.1. New Parallelization Strategy

To use the ray tracer in a way that could favor the locality improvements shown here, a different parallelization strategy was required. The main reason for this is that, to better exploit geometrical proximity, it is better for all threads to share the processing of the same pixel, instead of processing rays from different pixels, that consequently are more apart from each other.

Also, since at each new depth level, rays are completely unsorted due to the high divergence that is natural from Ray Tracing algorithms, it is useful to allow the rays to be sorted again and the blocks reorganized. This could also be done with the initial parallelization approach, however, it would require each ray to keep an even larger amount of auxiliary data, such as the pixel it belongs to, increasing memory footprint, and hurting locality. This new strategy is only possible if we assume that the implementation only provides acceptable results with a very high number of samples per pixel (values lower than 10000 result in to much noise in the final result), so that they can be split across enough blocks and threads.

To account for the high divergence when compared to the more regular N-Body problems of the previous test cases, a two step sort is done here. First, the entire set of rays is spatially sorted by their origin point. This ensures that all rays within a block are as much close as possible. But this won't prove useful for cases where two rays start roughly at the same origin point, but have completely different directions, since their traversal path will still be different. For this, a second spatial sort is done, now for each block (blocks are nothing more than pointers to a subset of the global ray list), and this time using ray direction as the key. This way, when a block diverges during the traversal, it is more likely that there will only be a small number of breaking points in the block, rather than having random block elements to go to each traversal.

Upon a collision, the ray will update its contribution and weight to the final pixel result, and its position and direction will be updated to reflect the newly created ray, if necessary. A final reduction is done on the entire set of rays to compute the final value for the pixel.

3.3.2. Expectations

During the development of this test case, it was noted that not only the irregularity was likely much higher than in the other samples, hindering improvements. The amount of extra auxiliary data required to keep track of all collision information during the traversal is also much higher.

Another fact that limits improvements is that this problem deals with two particular items: rays and objects on the scene, where only the objects are indexed into an acceleration structure. In both Barnes-Hut and Point Correlation, the items traversing the structure were the points themselves, which reduces memory footprint. Here, a list of rays is accessing a different set of items, the objects in the scene.

This should probably translate to a smaller ideal block size for this particular problem, but that may itself be a problem, as that block size may also be too small for any substantial gain.

4. Results

Figure 4 shows the improvements in execution time obtained

[2] `smallpt` was optimized for code size, not speed. All objects were simply spheres (even the walls), and no acceleration structure was provided, minimizing the amount of code

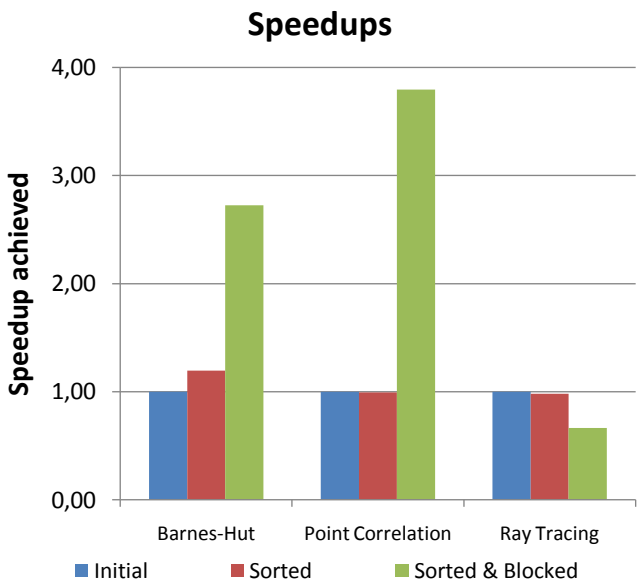

Figure 4: Speedups attained using Sorting and Point Blocking for each of the three implemented examples.

with the optimizations described in this document when applied to the three implemented examples. PC got the largest speedup of 3.79 from using Sorting and Point Blocking, followed by BH with a speedup of 2.7, while RT did not achieve better execution times. To notice that the PC algorithm did not benefit at all from using only Sorting. This can be easily explained by the input generator, which was copied from the BH original implementation in Galois, and may not be suitable for the PC problem. The lack of speedup in the RT example can be easily explained by the increased complexity behind managing the blocks.

Despite the RT not achieving a better execution time, fig. 5 shows that it benefited from the described optimizations in cache levels 1 and 2. The reported speedups can also be justified by a significant decrease in cache miss rates in the same two levels due to the increase of locality. The figure shows decreases around 5% in the first level of cache for both BH and PC, but only a neglectable 1% for RT. The best improvements are shown in L2, where BH decreased the miss rate by almost 80%, PC decreased by more than 50% and RT decreased the miss rate by 25%. In the last level of cache, no improvements are shown for any of the examples.

5. Conclusions

In this document, two optimizations performed by the *TreeTiler* framework to increase locality in irregular applications with acceleration structures were reviewed and applied to three distinct examples (Barnes-Hut, Two Point Correlation and a Ray Tracer) using the Galois framework.

The first optimization consisted in sorting the domain objects in such a way that similar patterns traversing the acceleration structure would be processed consecutively. This allows an increase in temporal locality, but fails when the traversal is too deep to fit in cache.

Blocking (the second optimization) applies a tilling approach to group objects during the traversal of the tree. Applied together with Sorting this amplifies temporal locality, allowing multiple objects with similar patterns to traverse the tree together, thus decreasing cache misses for each level of the acceleration structure.

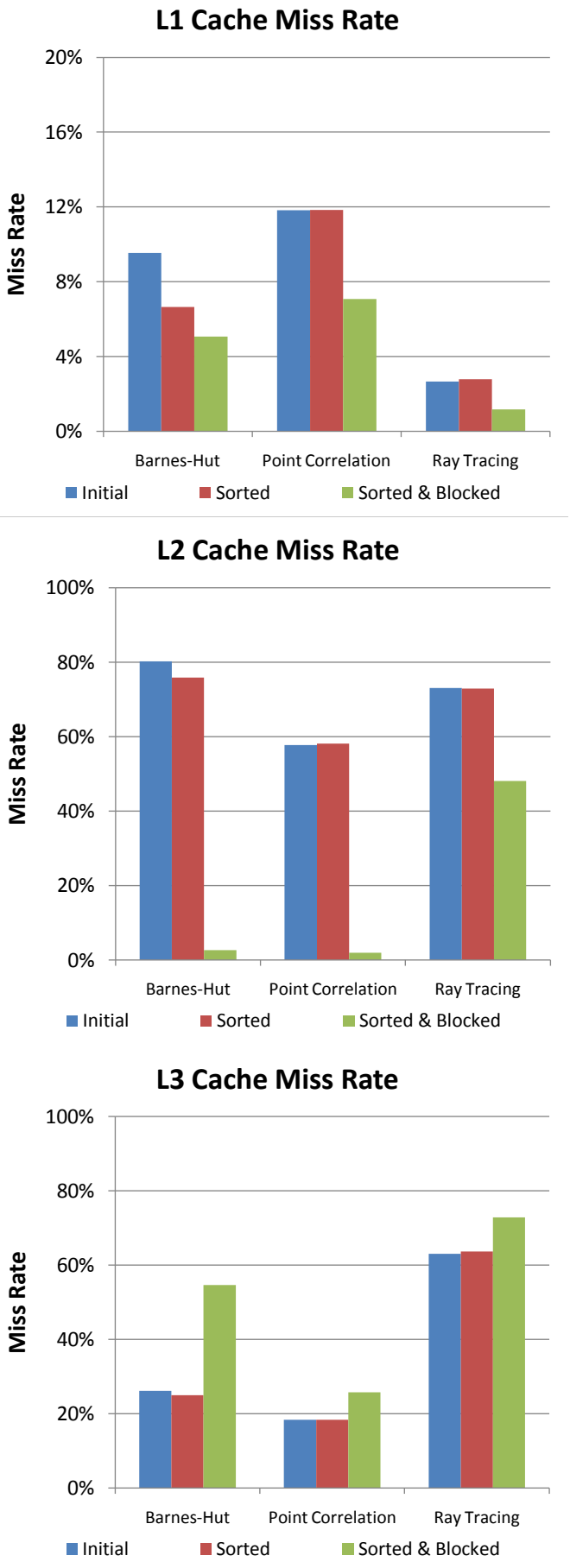

Figure 5: Cache miss rates in each level for the base version and using the two optimizations for each of the described examples.

For Barnes-Hut, Galois already presented a functional implementation, using an Octree. The implementation of the Sorting optimization was performed using the CGAL library to spatially sort the bodies. Blocking required a change in the Octree traversal so that groups of bodies would be considered together for each node.

The Two Point Correlation example was implemented entirely from scratch using a kD-Tree as an acceleration structure. Also being an N-Body problem, it differs from the Barnes-Hut example in both the acceleration structure and the need for a reduction to compute the final result in a parallel execution. Despite that, the implementation of both optimizations is similar.

As for the Ray Tracer, an implementation already available online was used as the base. The final implementation processes one pixel at a time parallelizing the computation of the rays. Due to the increased complexity of this problem, where both the origin and the direction of the rays must be taken into account for Sorting, rays are resorted and blocks rebuilt after each level of rays has been computed.

Results showed significant improvements in speedups due to the decrease of cache miss rates in both the Barnes-Hut and the Two Point Correlation example. While improvements regarding miss rates also exist in the Ray Tracer example, the highly divergent characteristics of the algorithm make the management of the blocks too costfull for any speedups to exist.

Proven the effects of the described optimizations in the temporal locality of irregular structures, future work can focus in implementing a generic way for Galois to apply these optimizations at compile time, as described in [1]. After this, an automatic tuning tool similar to the AutoTuner in *TreeTiler* could be implemented to optimize the parameters for a given implementation in runtime.

References

[1] Y. Jo and M. Kulkarni, "Enhancing locality for recursive traversals of recursive structures", in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. ACM, 2011, pp. 463–482.

[2] ISS Group at the University of Texas, "Galois", http://iss.ices.utexas.edu/?p=projects/galois.

[3] Kevin Beason, "smallpt: Global illumination in 99 lines of C++", http://www.kevinbeason.com/smallpt/.

[4] Intel Corporation, *Intel® Xeon® Processor 5500 Series Datasheet, Volume 1*, June 2011.

Appendix A. Environmental Setup

All the results retrieved for this document were obtained using tests performed in the Maxwell machine[3]. Table 1 shows the hardware description of this machine.

Processors:	4
Processor model:	Intel® Xeon® X5570
Cores per processor:	6
Threads per core:	2
Clock frequency:	2.93 GHz
L1 cache:	32 KB + 32 KB per core
L2 cache:	256 KB per core
L3 cache:	8 MB shared
RAM:	23 GB

Table 1: Maxwell machine hardware description. See [4] for further detail about this processor.

[3] maxwell.ices.utexas.edu