Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Cache-Efficient Parallel Isosurface Extraction for Shared Cache Multicores

M. Tchiboukdjian[†1] and V. Danjean[‡2] and B. Raffin[§3]

[1] CNRS - CEA/DAM,DIF
[2] Grenoble Universites
[3] INRIA

Abstract

This paper proposes to revisit isosurface extraction algorithms taking into consideration two specific aspects of recent multicore architectures: their intrinsic parallelism associated with the presence of multiple computing cores and their cache hierarchy that often includes private caches as well as caches shared between all cores. Taking advantage of these shared caches require adapting the parallelization scheme to make the core collaborate on cache usage and not compete for it, which can impair performance. We propose to have cores working on independent but close data sets that can all fit in the shared cache. We propose two shared cache aware parallel isosurface algorithms, one based on marching tetrahedra, and one using a min-max tree as acceleration data structure. We theoretically prove that in both cases the number of cache misses is the same as for the sequential algorithm for the same cache size. The algorithms are based on the FastCOL cache-oblivious data layout for irregular meshes. The CO layout also enables to build a very compact min-max tree that leads to a reduced number of cache misses. Experiments confirm the interest of these shared cache aware isosurface algorithms, the performance gain increasing as the shared cache size to core number ratio decreases.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Software]: Concurrent Programming—Parallel Programming I.3.3 [Computer Graphics]: Picture/Image Generation—Isosurface computation I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types

1. Introduction

Isosurface extraction is one of the most classical filters for scientific visualization. It has been intensively studied and various algorithms exist with different acceleration data structures and parallelizations.

In this paper, we focus on one specific aspect of its parallelization that has not been addressed so far: how to adapt the algorithm to take advantage of the shared caches often present on multicore processors. The goal is to propose an algorithm that saves cache misses, thus improving performance, compared to parallel algorithms that do not take into account the shared cache amongst several cores.

Multicore architectures usually have their last cache level shared between cores. For instance the L3 cache of the Intel Nehalem, the L2 cache of the Intel Larrabee or the L1 cache of NVIDIA Fermi processors are shared. Compared to private caches, this shared cache architecture can bring performance benefits if managed adequately. It allows fast communication between cores. If some cores work on the same data, these data are not duplicated into several caches. A core can potentially use more than its fraction of the cache if necessary. But this requires the algorithms to be adapted to make the cores collaborate on cache usage. Classical parallelization approaches usually favor tasks working on independent data sets to reduce communication and synchronization overhead. It results in competition rather than collaboration between cores for shared cache usage. Performance is at most equivalent to a private cache configuration. Indeed, [Has10] shows that this is actually worse than with pri-

† marc.tchiboukdjian@imag.fr
‡ vincent.danjean@imag.fr
§ bruno.raffin@imag.fr

DOI: 10.2312/EGPGV/EGPGV10/081-090

vate caches as the LRU replacement policy performs poorly in this context.

Many scientific visualization filters, like isosurface extraction, are memory bounded. Favoring the locality of access patterns through an adapted data layout can bring significant performance benefits. In this paper we propose to have cores working on independent but close (regarding the memory layout and spatial locality) data sets that can all fit in the shared cache. If a core needs a data that is not in its data set, there is a good chance it will find it in the data set loaded in the cache by one of its neighbors, thus saving cache misses. We propose two versions of this isosurface parallel algorithm, one based on the marching tetrahedra (MT), and one using a min-max tree as acceleration data structure. We theoretically prove that in both cases the number of cache misses is the same as for the sequential algorithm using a cache of the same size. This is like if each core would benefit from a full size private cache, at the price of a few extra synchronizations required to ensure a proper collaboration between cores. The algorithm is based on the cache oblivious (CO) data layout for irregular meshes proposed in [TDR10]. Not only it ensures a strong data locality, but, in opposite to other layouts, it also provides a theoretical bound on the number of cache misses. Our proof relies on an extension of this result to our parallel isosurface extraction algorithms.

Experiments confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs. It also show that a CO layout is not necessarily required, as other classical layouts can lead to a high enough data locality given the high cache to core ratio available on the tested processors.

We also detail a compact and cache-efficient tree structure for accelerating the MT algorithm. This tree is a min-max tree using a decomposition of the mesh in regions adapted to the CO mesh layout. It allows a compact storage as regions correspond to intervals of the cell array. It also induces very few caches misses as the active cells respect the order of the CO layout.

The paper is organized as follow. The MT algorithm is reviewed in section 2. The sequential CO algorithm is presented and proved in section 3 before to be extended to the parallel context in section 4. Experiments are detailed in section 5. Related works are discussed in section 6 before the conclusions.

2. Marching Tetrahedra Review

We review the data access patterns associated with the MT isosurface extraction algorithm and its tree accelerated versions.

Mesh Data Structure. A mesh data structure usually consists of two multidimensional arrays: an array storing point attributes (e.g. coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g. type of the cell, scalar values, etc.). When the mesh is composed of cells of different types (using various number of points), an additional array allows random access to cells (Fig. 1). As the cache performance for meshes having identical or not type cells is similar, we focus on homogeneous meshes in this paper.

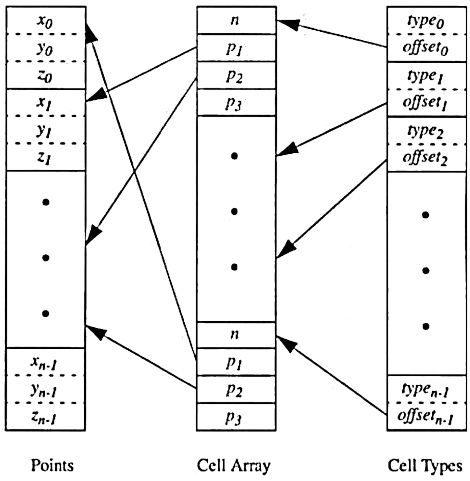

Figure 1: *The vtkUnstructuredGrid data structure (from the VTK Textbook [SML04]). The Points array contains point coordinates and the Cells array contains the indices of cell points. The Cell Types array contains the type of each cell and provides $O(1)$ random access to cells.*

MT Algorithm. For one cell of a mesh, the MT algorithm reads the point coordinates and scalar values and computes a linear approximation of the isosurface going through this cell. Applied on all mesh cells sequentially, it leads to a cost linear in the number of cells.

Tree Accelerated MT. The MT algorithm can be accelerated with various data structures allowing to efficiently search for the cells intersected by the isosurface. One such data structure is the min-max tree [WVG92]. An octree where each node stores the minimum and maximum values of its subtrees allows to quickly discard parts of the mesh that do not contain any intersected cell. The search is thus improved from $O(n)$ to $O(k+k\log n/k)$ where n is the number of cells and k the size of the isosurface (usually $k \ll n$). If the scalar field is spatially coherent, the performance is actually improved over this theoretical bound as large subtrees can be pruned.

Several kinds of min-max trees can be used. Octrees, kd-trees or more generally Binary Space Partitioning (BSP) trees recursively decompose the mesh into regions. The idea is that the scalar field does not vary too much in each re-

gion and thus the extreme of the scalar field form a small interval, less likely to contain the isovalue. Contrary to these geometric decompositions, the vtkSimpleTree of the VTK library [SML04] uses a layout decomposition. The regions consist of intervals of indices in the cell array. This tree is faster to compute and less memory consuming as regions are implicitly defined. However, depending on the cell layout, the scalar field may vary a lot in each region as they are not based on the geometry.

There exists an optimal data structure which is not based on the min-max tree. The interval tree [CMPS96] stores for each cell c the interval whose extremes are the minimum and maximum value of the points of c. The query time is improved to $O(\log n + k)$ whatever the spatial repartition of the scalar field is. The interval tree has been made I/O-efficient allowing a query with complexity $O(\log_B n + k/B)$, where B is the block size. This is optimal [CS97]. However this approach is not space-efficient since vertex information is duplicated many times. The 2-level indexing scheme based on the meta-cells technique introduced in [CSS98, CS99] improves over the interval tree in term of space usage but some data remain duplicated. Spatially close cells are grouped into meta-cells, which are then used in the I/O-efficient interval tree.

The tree accelerated MT, based on the min-max tree or the interval tree, improves performance over the regular MT at the cost of a higher memory requirement [SHwSS00]. Moreover, if the tree structure depends on the scalar field, the tree has to be stored for each scalar field. In this paper, we propose a min-max tree that is very compact and whose structure is the same for all scalar fields.

3. Cache-Efficient Isosurface Extraction

We now look at cache misses induced by sequential MT algorithms. After a general discussion, we focus on meshes stored according to the CO layout introduced in [TDR10] and give a theoretical guarantee of cache performance for a MT algorithm and a min-max tree accelerated.

3.1. Source of Cache Misses in MT

The cache misses in the MT algorithm come from accessing the cell array, accessing the point array, and accessing the min-max tree for the tree accelerated variant. In the regular MT algorithm, the cell array is traversed once in order, i.e. from low indices to high indices. Thus it induces only compulsory cache misses, corresponding to a first access. The point array is not accessed in a regular order like the cell array. Points are accessed by following a reference from the cell array, e.g. read coordinates of a point. This can induce capacity misses if the same point is needed by cells far away in the cell array. It is likely that the cache line containing the point will be evicted from the cache between these two accesses. In this case, the layout does not exhibit a good temporal locality. Moreover, not all data stored in a cache line will be use before being evicted; for example when a cache line stores data for two different points that are needed by cells far away in the cell array. In this case the layout lacks of spatial locality. For the tree accelerated MT, the cell array is not necessarily accessed in order like for the regular MT, which could lead to extra cache misses.

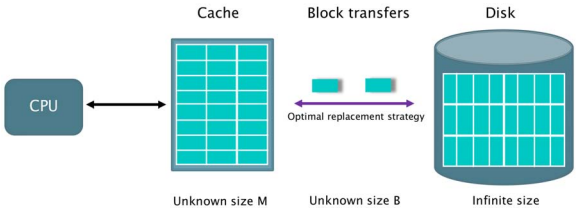

Figure 2: *The cache-oblivious memory model. The data are transferred by block of B consecutive elements into a cache of size M. Both parameters are unknown to the algorithm.*

The number of cache misses heavily depends on the mesh layout, i.e. how cells and points are sorted and stored in memory. For example, a point layout can improve cache performances if points corresponding to the same cell are stored nearby. These points may share cache lines, which increases spatial locality. Also, a cell layout can improve cache performance if the cells that are accessed consecutively by the min-max tree are stored nearby. Two consecutive active cells could be in the same cache line, increasing spatial locality.

As a mesh often includes several scalar fields, we consider here only optimizations of the layout that do not depend on the scalar field value. Thus these optimizations are efficient for all scalar fields.

3.2. Cache-Oblivious Model (CO)

We now introduce the cache-oblivious model from [FLPR99] we rely on to theoretically measure the number of cache misses. The memory hierarchy consists of two levels, a fast memory of size M called cache and an infinite size slow memory. The data are transferred between these two levels in blocks of B consecutive elements (Fig. 2). The cache performance of an algorithm is the number of block transfers needed to complete the computation. Parameters B and M are unknown to the algorithm to forbid tuning for a specific architecture. A good CO algorithm, i.e. one that performs well in the CO model, is thus expected to be cache-efficient whatever the cache and block size are.

3.3. CO Mesh Layout

In [TDR10], the authors introduce a CO layout algorithm for irregular meshes with a theoretical performance guarantee. It relies on a recursive mesh partitioning using a specific

BSP algorithm. This algorithm cuts the mesh guaranteeing a good tradeoff between minimizing the number of cut elements and having two partitions of similar size. When applied recursively, it ensures that spatially close and strongly connected data tend to be partitioned deeper in the BSP tree. The CO layout is obtained by storing the data linearly in memory from the first leaf of the BSP tree to the last one. The data loaded in a cache block are thus contiguous leaves of the BSP tree. It is cache-oblivious as to any block and cache size corresponds a BSP tree depth level. This ensures a strong locality and connectivity.

This CO layout algorithm has several benefits. Computing the layout is fast (complexity of $O(n \log n)$). When traversing the layout the cache-complexity is guaranteed not only for a strict layout consistent access order but also for a chunk based access. Data can be accessed by chunks of m consecutive elements in the layout, to be processed (in any order) before accessing another chunk anywhere in the layout.

Theorem 1 (Chunk traversal from [TDR10])
The CO layout guarantees that a traversal by chunks of size $m \leq M$ of an N-size mesh induces less than $N/B + O(N/m^{1/3})$ cache misses where B and M are the block and cache size, respectively.

3.4. MT Cache Performance

Using the cache-oblivious layout of the previous section, one can guarantee that the marching tetrahedra algorithm induces less that $O(n/B + n/M^{1/3})$ cache misses where B and M are the block size and the cache size. Indeed, the MT algorithm processes the mesh in order and thus by chunks of size M.

3.5. Tree Accelerated MT Cache Performance

We consider here a specific min-max tree, the one based on the BSP tree partitioning the mesh for computing the CO layout. We use this BSP tree because each node corresponds to a sub part of the mesh stored sequentially in memory. We thus get a min-max tree that is layout friendly. When traversing the BSP tree in prefix order and examining the mesh cells that might contain a part of the isosurface, mesh cells are accessed sequentially. Contiguous mesh cells can be skipped (pruned by the min-max tree), but we will never go backward. We can expect to save cache misses. This tree accelerated MT algorithm has been introduced in [TDR10]. Here, we also state its performance given the additional hypothesis that the scalar field is spatially coherent, i.e. for regions of n cells in the mesh the isosurface intersects on average at least $n^{2/3}$ cells for each region. A similar hypothesis is used in [CS99]. Using theorem 1, one can show that the cache performance of this tree accelerated MT algorithm is $O(k/B^{2/3} + k/M^{1/3})$ where k is the number of active cells.

4. Parallel Cache-Efficiency

We now introduce a parallel MT algorithm that is equivalent to the sequential one regarding cache performance.

4.1. Shared Cache Multicore

Most last generation multicores share a similar design for the cache hierarchy. Each core has its own private caches while the last cache level is shared between all cores. For instance the Intel Nehalem, the AMD Phenom and the IBM Power7 all have a shared L3 cache. Coming GPU architectures also adopt this cache design. The Intel Larrabee has a shared L2 cache. The NVIDIA Fermi has a L1 cache that is shared for all stream processors in the same multiprocessors and a L2 cache that is shared across all multiprocessors. They are many advantages of the shared cache compared to private caches. It allows fast communication between cores. If some cores work on the same data, these data are not duplicated into several caches. A core can potentially use more than its fraction of the cache if necessary. But this requires the algorithms to be adapted to make the cores collaborate on cache usage. Classical parallelization approaches that are not shared cache aware lead to competition for shared cache. Performance, at most equivalent to a private cache configuration, is actually impaired as the LRU replacement policy performs poorly in this context [Has10].

4.2. Shared Cache Aware Parallelization

We now present a new parallelization scheme that guarantees that the cache performance of the underlying sequential application is not reduced. The idea is to have cores working on close data so that each core has the impression to own the totality of the shared cache. Cache misses of one core profit to other cores as they are likely to also need these data in a near future. But cores should not work on data that are too close either because this could cause bad private cache behaviors.

The parallel algorithm is based on the sequential execution order $i_0, i_1, \ldots, i_p$. We assume that the sequential algorithm has good locality and thus data that are processed closely in the sequential execution are also close in memory. Informally let i_m be the first instruction whose processing would need to evict from the shared cache data needed by instruction i_0. To keep the cache performance of the sequential algorithm, the parallel scheme will deviate from the sequential order at most for m instructions. That is, instruction i_k can be processed only when instructions i_1 to i_{k-m} have been completed. This way, data evicted when processing i_k do not affect the processing of the other instructions. Moreover, as the cores work on instructions close in the sequential order, they work on close data and thus can profit of other cores cache misses.

4.3. Parallel MT

We now apply this parallel scheme to the MT algorithm. Let m be the maximal size, in number of cells, such that the corresponding region of the mesh fits entirely in the shared cache. It corresponds to the cells of the largest subtree of the CO layout BSP tree that fit in the last level of cache. To process all cells in parallel, we divide the n cells into n/m chunks of size m and then process each chunk in parallel. Processing a new chunk is started only when the previous one has been entirely processed. To process one chunk, we divide the m cells into m/p groups, one for each core. From theorem 1, we know that a chunk can be processed in any order without affecting the cache performance, thus the number of cache misses of the sequential algorithm 3.4 is still valid for this parallel algorithm.

Let compare this shared cache aware parallelization with a standard parallelization. The trivial way to process n cells in parallel is to divide the cells in groups of n/p, one for each core. Let assume now that the shared cache behaves as well as p private caches (it should be worse in practice). This parallelization yields at best on each core the performance of the sequential algorithm on a cache of size M/p, i.e.

$$O\left(\frac{n/p}{B}+\frac{n/p}{(M/p)^{1/3}}\right).$$

Thus, the total number of cache misses is

$$O\left(\frac{n}{B}+p^{1/3}\cdot\frac{n}{M^{1/3}}\right).$$

This is a factor of $p^{1/3}$ worse than the shared cache aware parallelization, which induces the same number of cache misses as the sequential algorithm, i.e. $O(n/B+n/M^{1/3})$.

However, the shared cache aware parallelization has much more global synchronizations, where all cores wait for the last one to finish. There is one synchronization per chunk, n/m compared to only 1 for the standard parallelization. We show in the experiments that this additional cost of synchronization does not impair too much the performance of the shared cache aware parallelization.

4.4. Parallel Tree Accelerated MT

We now apply the shared cache aware parallelization to the MT accelerated with a min-max tree. We use as a min-max tree the BSP tree built with the layout. Let consider nodes of the BSP tree that correspond to a region of the mesh fitting in the shared cache. By theorem 1, we know that each of these nodes can be processed in any order without affecting the cache performance. Thus each node of the BSP tree can be processed in parallel by the p processors. A new node starts being processed only when the previous one is terminated. The cache complexity of the sequential algorithm of section 3.5 is still valid for this parallel algorithm. The overhead is the same as for the classic parallel version, i.e. $O(p^{1/3})$. However the shared cache aware parallelization also has more synchronizations, one per node of the BSP tree.

5. Implementation and Experiments

5.1. Architectures and Meshes

We took 4 different meshes (Blunt fin, buckyball, liquid oxygen post, plasma64 from the AIM@SHAPE Shape Repository (`http://shapes.aim-at-shape.net/`), processed to generate instances of 150M cells. We used tetgen (available at `http://tetgen.berlios.de/`) to refine the meshes by adding volume constraints to each tetrahedron (we used the command `tetgen -raq`). For each mesh, we generated two finer meshes. In the first one, all tetrahedra have approximately the same volume. In the second one, we used a volume constraint proportional to the inverse of the gradient of the scalar field to mimic adaptive mesh refinement. It leads to a set of 8 meshes with approximately 150M cells each.

For each mesh, 3 layouts are compared. The original layout as downloaded from the web, the geometric layout where points and cells are sorted according to x,y,z coordinates using lexicographic order, and the CO layout generated by FastCOL [TDR10].

The experiments were conducted on three different architectures. Two multicores (Intel Core2 E6750 @ 2.66Ghz, dual core, private cache L1 32KB, shared cache L2 4MB and Intel Xeon E5530 @ 2.4Ghz, quad core, private cache L1 32KB and L2 256K, shared cache L3 8MB) with a shared last level of cache and one multicore with only private caches (AMD Opteron 875 @ 2.2Ghz, dual core, private cache L1 8KB, private cache L2 1MB).

We measured the execution time and the number of L1, L2 and L3 cache misses using the PAPI software [BDG*00]. For each experiment (architecture, layout and algorithm fixed), the execution time and the numbers of cache misses are very stable. Results of the experiments are the median of 10 runs.

5.2. Min-Max Tree Implementation

We first describe the data structure used to store a min-max tree, whatever the cell layout is. Each node of the tree corresponds to a region of the mesh and contains the minimum and maximum value of the scalar field in this region. Those nodes are stored linearly in memory like a pointerless binary tree. A more efficient layout could be used, like the van Emde Boas layout of the cache-oblivious B-tree [ABF04]. We used a simple layout as the tree is small and the tree traversal is only a small part of the computation (less than 10%). For each leaf of the tree, we need the cells lying in the corresponding region of the mesh. A simple way is to store in each leaf node a list of cells, which results in a complex and inefficient storage. Instead we store a permutation

π of cells indices such that to each node of the tree corresponds an interval of cells. This way, each leaf contains only two cell indices i and j. To find the cells laying in a region, we apply the permutation to each cell indices in the interval: $\pi(i), \ldots, \pi(j)$. Thus the data structure contains the tree and the permutation.

In the case of a min-max tree based on the CO layout and its associated BSP tree (Section 3.5), the cell permutation is simply the identity. Storing the permutation is no longer needed, leading to a very compact data structure.

The VTK binary tree vtkSimpleTree [SML04] can be used to accelerate a MT algorithm. It is based on a recursive decomposition of the layout of the mesh. This tree does not need to store a cell permutation either but the regions it defines are not based on geometry and thus could map together very distant cells. It may not be as efficient when pruning active regions.

The BSP tree associated with the CO layout is both based on the layout and on the geometry. It is compact, can efficiently discard inactive regions and induces a mesh traversal with few cache misses.

We compare our min-max tree to the compact interval tree of [WJV07]. To our knowledge, it is the most compact implementation of the I/O efficient interval tree for isosurface extraction. To index a 25M cells mesh decomposed into 9x9x9 metacells, they need between 200MB and 250MB, a factor 2 improvement over the standard I/O efficient interval tree of [CS97]. In comparison, for a mesh of 150M cells, a min-max tree using our permutation based storage needs 579MB, only 3 times bigger for a 6 times bigger mesh. With the CO layout the permutation is not stored and the tree only requires 6MB, which is several orders of magnitude smaller than trees reported in the literature. The smallest tree reported in [SHwSS00], the branch-on-need octree, requires roughly 100MB for a 16M cell mesh. In the experiments we used 2x2x2 metacells (8 cells per leaf) leading to a space requirement of 958MB for the standard min-max tree and 385MB for the CO version, 36% and 14%, respectively, of the size of the considered 150M cells mesh (one scalar field).

5.3. Sequential Performance

Table 1 shows the sequential performance of the MT algorithm and the tree accelerated MT algorithm on various layouts and architectures.

For the MT algorithm, the geometric layout outperforms both in time and cache misses the original layout. The cache oblivious layout shows the best performance with an improvement of a factor between 1.5 and 2 over the original layout on all architectures.

For each layout, the tree accelerated MT algorithm is always faster than the regular MT algorithm. However, for the tree accelerated MT algorithm, the geometric layout does not improve performance over the original layout. The geometric and the original layout use the same kd-tree so the difference is not due to a smallest number of active cells but only to a better cache behavior. As both the layout and the kd-tree are based on the geometry, it is possible that the tree accelerated algorithm accesses the mesh less efficiently. The CO layout with the adapted min-max tree shows the best performance with very few cache misses. The reduction in cache misses over the original layout with the kd-tree is between 3 and 5. This is impressive as the tree accelerated algorithm already induces few cache misses. This results in time speedup of 2.

5.4. Parallel MT Implementation

We present here the implementation of the two parallel MT algorithms. The standard parallelization that acts as if the shared cache was split into p private caches is denoted *split cache*, while the shared cache aware parallelization is called *shared cache*. We used `pthread` to parallelize all algorithms as this allows a fine grain control on synchronizations and to reduce parallelism overhead over high level parallel libraries like OpenMP.

For the split cache parallel MT algorithm, the array of cells is statically divided into p groups. Each group is assigned to one thread and all threads synchronize at the end of the computation.

For the shared cache version, the array of cells is first divided into chunks as large as possible but that can fit in the shared cache. Then each chunk is statically divided into p groups and all threads synchronize at the end of each chunk before starting to compute the next one. Each synchronization is implemented with a `pthread_barrier`. Threads wait at the barrier and are released when all of them have reached the barrier. Compared to the split cache version where there is only one synchronization at the end of the computation, this version has n/m synchronizations, one per chunk. So we expect the threads to spend more time waiting for other threads to finish their work. Nevertheless, the shared cache version keeps all threads working on data close in mesh space and thus in memory, which should result in less cache misses. We show in the next section that the shared cache version outperforms the split cache version. The reduction of cache misses more than counterbalance the augmentation of waiting times due to an increased number of synchronizations.

5.5. Parallel MT Performance

Table 2 (top) compares the performance of the two parallel MT algorithms on a multicore with only private caches and two multicores with a shared last level of cache. Both algorithms execute the same number of instructions. The very good speedups on the Opteron suggest that the work load is well balanced. We also examined the number of instructions

Table 1: *Sequential performance of the MT algorithm and the tree accelerated MT (TA) on Opteron, Core2 and Nehalem. Only one core is used. For the original and geometric layout, the min-max tree is a kd-tree. For the CO layout the associated BSP tree is used. Times are in ms. Cache misses for all levels of caches (L1, L2 and L3) are in millions.*

		Opteron			Core2			Nehalem		
		Time	L1	L2	Time	L1	L2	Time	L2	L3
MT	Original	10800	155.2	48.2	2940	190.2	59.4	3600	190.2	54.0
	Geometric	6200	122.8	25.9	2120	141.6	49.8	2600	216.4	53.2
	CO	5250	45.5	14.7	1935	47.8	41.2	2280	91.5	7.6
TA	Original	1650	16.8	13.0	970	29.3	21.3	855	23.8	16.8
	Geometric	2890	25.9	22.0	1385	52.6	40.4	1300	44.4	23.8
	CO	690	5.0	4.0	565	5.4	4.4	415	8.2	3.5

executed by each thread and they are very close. Thus the performance difference observed between both algorithms is mainly based on the cache behavior.

Both algorithms scale very well on the Opteron. We obtain a speedup close to 2 for the original and geometric layouts. The CO layout shows a super linear speedup due to a very low number of cache misses for the parallel algorithms, 2 times less cache misses than the sequential algorithm. However we are not able to explain such a reduction of cache misses. The two parallel schemes perform equally well with a very close number of cache misses for both levels of cache L1 and L2. This was expected as there is no shared level of cache.

For the Core2 and Nehalem processors, the parallel algorithms do not scale as well as on the Opteron (2 threads run on the Core2 and 4 on the Nehalem). Indeed, the parallel algorithms do not have more cache than the sequential algorithm. Similarly to the Opteron, the number of private cache misses are very close for the two parallel schemes. As expected, the split cache version induces more cache misses than the sequential algorithm on the shared cache. The shared cache version keeps a number of cache misses very close to the one of the sequential version and thus is faster than the split cache version almost all the time for the original and geometric layout. Performances are similar for the CO layout due to the very low number of cache misses of the sequential algorithm (behavior analyzed in section 5.8) .

5.6. Parallel Min-Max Tree Implementation

In our implementation of the parallel min-max tree algorithm, only the generation of triangles is performed in parallel and not the tree traversal to select active cells. As the active cells selection phase represents only 7% of the sequential time, we expect the algorithms to still scale well. In the split cache version, the active leaves are statically divided into p groups, one per thread. Each thread processes its group of cells and all of them synchronize at the end. In the shared cache version, the min-max tree is first divided into nodes that correspond to regions of the mesh that fit in cache. Then actives leaves of each region are distributed to the threads and processed in parallel. Threads synchronize at the end of each region using a `pthread_barrier` before starting processing the next region. Like for the MT algorithm, the shared cache aware version uses more synchronizations to stay close to the sequential order but it leads to less cache misses compared to the split cache version.

5.7. Parallel Min-Max Tree Performance

Table 2 (bottom) compares the performance of the two parallel tree accelerated MTs. The behavior of the tree accelerated version is similar to the regular MT. Both schemes perform equally well on Opteron and the shared cache version performs better on the Core2 and the Nehalem. However in this case, the shared cache version still offers some improvement over the split cache version with the CO layout.

5.8. Measure of Locality

To better analyze the properties of the different layouts, we analytically relate the performance improvements to the better data locality in memory. We call "edge length" the memory gap between two vertices of the same edge in the vertex array loaded in memory. If a mesh has shorter edges, more of them will fit in cache and a better performance should be observed. Figure 3 shows that the CO layout favors smaller edge lengths than the two other layouts.

We now estimate the number of cache misses using an edge length based metric. Let N be the size of a mesh (in bytes), E the set of all edges of the mesh, B the cache line size and M the cache size, we estimate the number of cache misses by:

$$CM_{\text{seq}} \approx \frac{N}{B} + \sum_{e \in E} \mathbb{1}_{\lambda_e > M}$$

where λ_e is the length of the edge e. We count the number of cache misses for a linear full read of the data arrays and we add one cache miss per edge whose length is bigger than the cache size M.

Table 2: *Performance comparisons on three different processors, for three different layouts original (Ori.), geometric (Geo.) and cache-oblivious (CO) of the sequential MT algorithm, the split cache and shared cache parallel MT algorithms (top), or the tree accelerated sequential MT algorithm, the split cache and shared cache tree accelerated parallel MT algorithms (bottom). Hyperthreading is disabled on Intel processors. The parallel algorithms are executed with 2 threads on the Opteron and Core2 processors, and with 4 threads on the Nehalem. Cache misses are in millions. Speedups are relative to the sequential algorithm on the same layout (Seq. speedup) or relative to the sequential algorithm on the original layout (Orig. speedup).*

			Opteron				Core2				Nehalem			
			Speedup		Cache Misses		Speedup		Cache Misses		Speedup		Cache Misses	
			Seq.	Orig.	L1	L2	Seq.	Orig.	L1	L2	Seq.	Orig.	L2	L3
MT	Orig.	Sequential	1.00		155.2	48.2	1.00		190.2	59.4	1.00		190.2	54.0
		Split Cache	2.07		155.2	42.1	1.43		189.9	75.0	2.83		190.1	70.0
		Shared Cache	1.99		155.3	44.0	1.72		188.3	57.8	3.36		191.1	55.2
	Geo.	Sequential	1.00	1.74	122.8	25.9	1.00	1.39	141.6	49.8	1.00	1.38	216.4	53.2
		Split Cache	1.98	3.45	122.8	21.1	1.59	2.21	140.9	59.0	3.02	4.19	217.8	68.5
		Shared Cache	1.98	3.45	122.8	21.4	1.78	2.47	140.2	48.6	3.31	4.59	220.2	56.9
	CO	Sequential	1.00	2.06	45.5	14.7	1.00	1.52	47.8	41.2	1.00	1.58	91.5	7.6
		Split Cache	2.69	5.53	45.5	7.0	1.74	2.65	47.6	41.3	3.10	4.90	92.3	8.9
		Shared Cache	2.56	5.27	45.6	6.6	1.71	2.60	47.6	41.1	3.06	4.83	93.5	8.7
Tree Accelerated MT	Orig.	Sequential	1.00		16.8	13.1	1.00		29.3	21.3	1.00		23.8	16.8
		Split Cache	1.55		16.6	13.1	1.42		28.9	22.1	2.59		22.9	17.9
		Shared Cache	1.54		16.6	13.1	1.60		29.1	21.1	2.85		22.8	16.7
	Geo.	Sequential	1.00	0.57	25.9	22.0	1.00	0.70	52.6	40.4	1.00	0.66	44.4	23.8
		Split Cache	1.68	0.96	25.8	22.0	1.56	1.09	52.5	41.6	2.89	1.90	43.8	25.9
		Shared Cache	1.64	0.93	25.6	22.0	1.73	1.21	52.5	40.2	3.02	1.99	43.9	24.6
	CO	Sequential	1.00	2.39	5.0	4.4	1.00	1.72	5.4	4.4	1.00	2.06	8.2	3.5
		Split Cache	1.31	3.14	4.5	3.8	1.43	2.46	5.1	5.1	2.08	4.28	7.3	3.7
		Shared Cache	1.33	3.17	4.5	3.7	1.53	2.62	5.3	3.6	2.18	4.50	7.3	3.4

For the split cache version, we could think of each core having a private cache of size M/p instead of a shared cache of size M, which gives an expected number of cache misses of

$$CM_{\text{split}} \approx \frac{N}{B} + \sum_{e \in E} \mathbb{1}_{\lambda_e > M/p}$$

As the shared cache version has the same cache performance as the sequential algorithm, we have

$$\frac{CM_{\text{shared}} - \frac{N}{B}}{CM_{\text{split}} - \frac{N}{B}} \approx \frac{\sum_{e \in E} \mathbb{1}_{\lambda_e > M}}{\sum_{e \in E} \mathbb{1}_{\lambda_e > M/p}}$$

Let check this formula for Core2 (Table 2). In our experiments, we measured that a linear read of the mesh induces 37.6 millions of L2 cache misses so we have $N/B = 37.6$. On the original layout, the experiments give

$$\frac{CM_{\text{shared}} - \frac{N}{B}}{CM_{\text{split}} - \frac{N}{B}} = \frac{75.0 - 37.6}{57.8 - 37.6} \approx 1.85\,,$$

which is close to the value of 1.90 found with the edge length metric. A similar calculation using the experiments and the geometric layout gives 1.95, with an edge length metric of 1.95. On Nehalem we measured 2.45 for the original layout and 2.00 for the geometric layout, close to the edge length metric values of 2.30 and 2.35. For the CO layout, both the measured cache misses and the edge length values are close to 1. This is consistent with the split cache and shared cache algorithms inducing approximately the same number of cache misses.

This reasoning allows to extrapolate the gain in cache misses of the shared cache parallelization over the split cache parallelization for various cache sizes. Figure 4 presents expected gain for all three layouts on various shared cache sizes. One can remark 3 things. The more cache is available, the less speedup is to be expected. The more cores share the same cache, the more speedup is to be expected. If the sequential application has already very few cache misses, then it performs well under both parallelization schemes (CO case). However the shared cache approach has still an interest for well optimized layouts if the cache is small and shared by a lot of cores. The L1 cache of the Fermi architecture can hold 48KB of data and is shared amongst 32 processing units. Using the same edge length prediction, the shared cache scheme should reduce the number of cache

misses by 66% compared to the split cache scheme for the CO layout.

6. Related Work

There is another method for efficient isosurface extraction not studied in this article: the seed set and propagation algorithm [BPS96] that proposes to find cells intersected by the isosurface using a graph search on the topology of the mesh. We believe such an approach can benefit for our shared cache scheme. There is also an entirely different approach, not based on marching tetrahedra but on ray tracing [WFM*05]. We plan to further study the impact on the shared cache on ray tracing applications.

Many parallel schemes have been proposed to achieve good load balancing for isosurface extraction [ZNZ04]. Those techniques could be coupled with our shared cache scheme to efficiently balance the load inside a chunk. However those techniques only take into account the number of instructions and not the cache misses. To overcome this problem, we plan to use a work stealing load balancing scheme.

The closest work is the out-of-core parallel interval tree of [WJV07]. Authors provide a provably efficient load balancing scheme. However the technique is cache aware. Moreover, they focus on distributed processors that do not share caches.

Other layouts exist for efficient mesh traversal. Space filling curves have been used for regular meshes in [PF01]. For unstructured meshes, OpenCCL [YLPM05] provides an heuristic for building efficient layouts but without any performance guarantee. Using the algorithm of OpenCCL, [YM06] proposes an efficient layout for both a mesh and a bounding volume hierarchy tree. This is similar to our adapted min-max tree as both trees are tailored to efficiently access the mesh.

To our knowledge, the first work on shared cache is [BG04, CGK*07] that presents a scheduler that follows as much as possible the sequential execution order. However, threads working on too close data can impair the performance of private caches. By using a suitable chunk size, our shared cache algorithm benefits from the shared level of cache while still using private caches efficiently.

7. Conclusion

This paper focused on cache efficiency for isosurface extraction. We theoretically guarantee the performance of the sequential MT algorithm relying on the CO layout introduced in [TDR10], as well as a tree accelerated version taking advantage of the BSP tree built to compute the layout. Then, we consider a parallelization scheme that takes into account that caches may be shared on multicore processors. We prove that this parallelization leads to the same number of cache misses than the sequential algorithm, less than a traditional parallelization assuming caches are all privates. Experiments confirm the benefits of shared cache aware approaches and of CO based algorithms. We expect these techniques to be even more effective on architectures with small caches like the Fermi GPU. Future work will try to combine shared cache aware access patterns with advanced load balancing schemes.

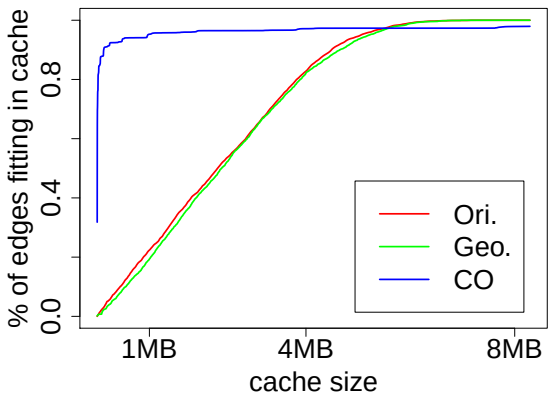

Figure 3: *Cumulative distribution function of edge lengths for various layouts applied to the 150M plasma mesh (the other meshes produce similar graphs).*

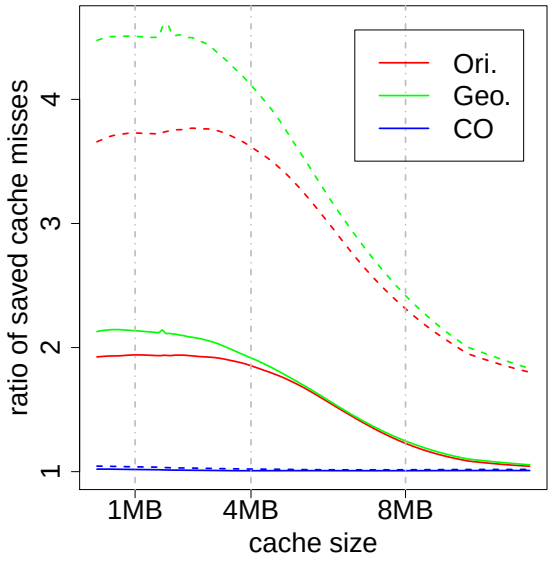

Figure 4: *Gain in cache misses of shared cache over split cache for the* 150*M plasma mesh. Solid lines assume a cache shared between 2 cores, dashed lines assume a cache shared between 4 cores.*

References

[ABF04] ARGE L., BRODAL G., FAGERBERG R.: Cache oblivious data structures. *Handbook on Data Structures and Applications* (2004). 5

[BDG*00] BROWNE S., DONGARRA J., GARNER N., HO G., MUCCI P.: A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications 14* (2000), 189–204. 5

[BG04] BLELLOCH G. E., GIBBONS P. B.: Effectively sharing a cache among threads. In *Proceedings of SPAA '04* (2004), pp. 235–244. 9

[BPS96] BAJAJ C. L., PASCUCCI V., SCHIKORE D. R.: Fast isocontouring for improved interactivity. In *Proceedings of VVS '96* (1996), p. 39. 9

[CGK*07] CHEN S., GIBBONS P. B., KOZUCH M., ILEIOS LIASKOVITIS V., AILAMAKI A., BLELLOCH G. E., FALSAFI B., FIX L., HARDAVELLAS N., MOWRY T. C., WILKERSON C.: Scheduling threads for constructive cache sharing on cmps. In *Proceedings of SPAA '07* (2007), pp. 105–115. 9

[CMPS96] CIGNONI P., MONTANI C., PUPPO E., SCOPIGNO R.: Optimal isosurface extraction from irregular volume data. In *Proceedings of VVS '96* (1996), pp. 31–38. 3

[CS97] CHIANG Y.-J., SILVA C.: I/O optimal isosurface extraction. In *Proceedings of Visualization '97* (1997), pp. 293–300. 3, 6

[CS99] CHIANG Y.-J., SILVA C. T.: External memory techniques for isosurface extraction in scientific visualization. In *External memory algorithms* (Boston, MA, USA, 1999), American Mathematical Society, pp. 247–277. 3, 4

[CSS98] CHIANG Y.-J., SILVA C., SCHROEDER W.: Interactive out-of-core isosurface extraction. In *Proceedings of Visualization '98* (1998), pp. 167–174. 3

[FLPR99] FRIGO M., LEISERSON C. E., PROKOP H., RAMACHANDRAN S.: Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (1999), p. 285. 3

[Has10] HASSIDIM A.: Cache replacement policies for multicore processors. In *Proceedings of Innovations in Computer Science* (2010). 1, 4

[PF01] PASCUCCI V., FRANK R.: Global Static Indexing for Real-Time Exploration of Very Large Regular Grids. In *Proceedings of Supercomputing '01* (2001), pp. 45–45. 9

[SHwSS00] SUTTON P. M., HANSEN C. D., WEI SHEN H., SCHIKORE D.: A case study of isosurface extraction algorithm performance. In *Data Visualization 2000* (2000), Springer, pp. 259–268. 3, 6

[SML04] SCHROEDER W., MARTIN K., LORENSEN B.: *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed.* Kitware Inc., 2004. 2, 3, 6

[TDR10] TCHIBOUKDJIAN M., DANJEAN V., RAFFIN B.: Binary mesh partitioning for cache-efficient visualization. *IEEE Transactions on Visualization and Computer Graphics 99*, PrePrints (2010). http://moais.imag.fr/membres/marc.tchiboukdjian/pub/tvcg10.pdf. 2, 3, 4, 5, 9

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11* (2005), 562–572. 9

[WJV07] WANG Q., JAJA J., VARSHNEY A.: An efficient and scalable parallel algorithm for out-of-core isosurface extraction and rendering. *J. Parallel Distrib. Comput. 67*, 5 (2007), 592–603. 6, 9

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3 (1992), 201–227. 2

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. In *ACM SIGGRAPH* (2005), pp. 886–893. 9

[YM06] YOON S.-E., MANOCHA D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. *Computer Graphics Forum 25*, 3 (2006), 507–516. 9

[ZNZ04] ZHANG H., NEWMAN T. S., ZHANG X.: Case study of multithreaded in-core isosurface extraction algorithms. In *EGPGV* (2004), pp. 83–92. 9