

# Optimizing Memory Synchronization for the Parallel Construction of Recursive Tree Hierarchies

Dirk Bartz  
WSI/GRIS, University of Tübingen\*

## Abstract

Multi-resolution methods are widely used in scientific visualization, image processing, and computer graphics. While many applications only require a one-time construction of multi-resolution data-structures, which can be done in a pre-process, this pre-process can take a significant amount of time. Considering large datasets, this time consumption can range from several minutes up to several hours, especially if this pre-process is frequently needed. Furthermore, numerous new applications, such as visibility queries, often need a dynamic reconstruction of a scene database, coded in a multi-resolution data-structure.

In this paper, we address the construction or reconstruction process of recursive tree hierarchies in parallel. In particular, we look into parallel dynamic memory allocation, the associated synchronization overhead, and memory access latency.

**Keywords:** Parallel hierarchies, recursive tree structures, octrees, memory synchronization, memory access latency on NUMA systems, shared memory, thread model.

## 1 Introduction

In computer graphics, hierarchical methods are widely used to reduce the complexity of common problems. Specifically, multi-resolution methods are used to reduce the polygon count of large models [8], to reduce the light interaction between different parts of a scene [10], and so forth. Among the most popular spatial multi-resolution representations are recursive tree structures like quadtrees and octrees [20], k-D-trees [3], and BSP-trees [6]. Unfortunately, the construction of these multi-resolution representations can be very time consuming. This is especially true if we consider large datasets. If this construction is needed frequently, the parallelization of this process quickly becomes worthwhile, even if it is considered as a pre-processing step. Furthermore, several applications require fast reconstructions (or re-evaluation) of octrees (or other recursive tree structures), such as changes of the color table, transfer functions, or isovalues in volume rendering applications [9, 17, 25]. In occlusion culling applications, animated objects cause a partial reconstruction of the scene representation [23].

To overcome the high construction time, we proposed to perform this process in parallel [2] by decoupling the recursive parent/children relationship of the tree elements. However, dynamic memory allocation was limiting the scalability of our algorithm by introducing additional synchronization overhead. In this paper, we discuss three approaches of dynamic memory allocation in parallel – where two approaches address this bottleneck – on three different shared-memory (“share everything” or tightly coupled) MIMD architectures.

Our paper is organized as follows: We first outline the background and the related work in Section 2. Next, we briefly review

the basic algorithm for the parallel and balanced tree (re-) construction and isosurface extraction (Section 3). In Section 4, we discuss three approaches for the reduction of the overhead during the dynamic allocation of memory. Finally, we draw a conclusion and give perspectives to future work.

## 2 Background and Related Work

Tree-based spatial subdivision schemes are widely used in scientific visualization, computer graphics, and image processing. Of special importance are schemes like quadtrees – which represent a regular two-dimensional subdivision –, octrees (the three-dimensional counterparts of quadtrees) [20], k-D-trees [3], and BSP-trees [6] – representing regular and irregular binary subdivisions of arbitrary dimensions. Although our results are generally valid for all recursive tree structures, we only focus on octrees for volume data.

An octree is a hierarchical spatial data-structure to represent three-dimensional volumetric data at different levels of details [20]. Starting with the superblock – representing the whole dataset – each octant is subdivided into eight child blocks. Each of these child blocks has half the size of the parent in each dimension (Fig. 1). This subdivision is performed until the lowest level is reached, where each block represents eight volumetric sample values (voxels). These bottom level blocks are called cells and they are identical to Marching Cubes cells, or cells in structured datasets with a rectilinear grid topology.

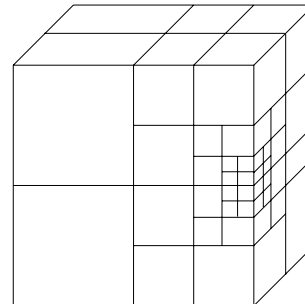


Figure 1: Octree of a volume dataset.

Due to the subdivision, the size of each octant is a power of two. Unfortunately, datasets usually do not have the exact size for this scheme (see Table 1). Therefore, some octants are “empty” – they do not intersect with the dataset – according to the alignment of the dataset within the octree. These “full octrees” result in wasted space allocated for empty octants. In order to save this space, we use a branch on demand octree (BONO) [25]. The BONO approach enumerates only the octants – and their children – that are not empty. Furthermore, BONO always tries to maximize the size of at least one octant by aligning the dataset to the lower-left-front corner of the octree [25].

\*WSI/GRIS, University of Tübingen, Auf der Morgenstelle 10/C9, D-72076 Tübingen, Germany, Email: bartz@gris.uni-tuebingen.de

Octrees are used in several applications to provide a multi-resolution representation. Laur and Hanrahan presented an octree-based scheme for hierarchical splatting [16]. Splats of different size and shape were used, according to the standard deviation of the color values of the different octree blocks. Grosso et al. presented a parallel implementation of this algorithm [9]. In their approach, a static parallelization of the octree construction was used, where up to eight threads were processing up to eight child blocks of the superblock. Greene et al. used an octree and an image pyramid for visibility queries in large polygonal environments [8]. Shekhar et al. used an octree representation of a volumetric dataset to generate a block-oriented polygon reduction scheme of its isosurface [21]. Levoy presented an approach to accelerate ray casting by using octrees [17], where coherent (non-contributing) data can be rapidly skipped. A hierarchical approach for cell-projection based volume rendering using a k-D-tree was proposed by Wilhelms et al. [26]. Wittenbrink and Kim presented an octree-based approach to accelerate permutation warping-based volume rendering where subvolumes are decomposed using the octree [27]. Swift et al. combined quadtree slices of a data volume to an octree [24]. This process starts at leaf level of the quadtree and continues up to the root, where the combination of two slices can be performed in parallel. However, details on the actual parallel implementation are sparse. Kela and Wynn proposed a parallel construction scheme for quad- and octrees, but push-up of information during that construction process were not supported [14].

The major focus of this paper is the parallel, asynchronous, and dynamic construction of recursive tree structures. However, as an example application, we choose isosurface extraction from volumetric datasets on structured grids using the Marching Cubes algorithm [19]. Various approaches are known for the efficient isosurface or contour extraction of large volumetric scalar fields. Value partitioning methods [18, 22, 5] store the minimum and maximum values of the sample values of a cell as a pair in a 2D field. Livnat et al. used a k-D-tree structure [18], and Shen et al. [22] used lattice subdivision to subdivide this field. Cignoni et al. used an interval tree as search index for optimal efficiency [5]. Space and value partitioning methods are used for an efficient contour propagation, starting from a small set of seed cells [13, 1].

We are using a parallel implementation of the BONO approach by Wilhelms and Van Gelder [25]. By storing the minimum and maximum values of the voxels at each block of the octree, the blocks which do not contain the isovalue in their minimum/maximum interval, can be rapidly skipped. After selecting all these contributing voxels (“surface voxels/cells”) of these blocks, the isosurface is generated.

Please note that octrees only depend on a rectilinear grid topology (structured grids), not on a rectilinear grid geometry. Therefore, this approach is suited – and implemented – for regular grids, non-uniform regular grids (datasets A-D), and curvilinear grids as well (datasets E, F).

### 3 Parallel Construction of Recursive Tree Hierarchies

In general, recursive tree structures are constructed in two stages; a split-down of a parent into several children, and a push-up of the results of the children back to the parent, i.e. the standard deviation, or – in our case – the minimum and maximum voxel values.

The parallelization of a recursive split-down is a rather simple task. Depending on the workload and the available processors, a subtree could be assigned to a thread. Usually, the second stage causes difficulties for a balanced parallelization. Due to their recursive relationship, we need to maintain the parent/child information. On the other hand, a balanced parallelization requires a decoupling

of this structure. A simple distributed top-down subdivision, as suggested for the first stage, only provides the top-down information, where every parent knows its children. For a push-up, we also need the bottom-up information – i.e., which block is the parent of the current block and needs to be updated by the current block. In our approach, we solve this problem by combining a central workload splitting job queue and our asynchronous push-up [2].

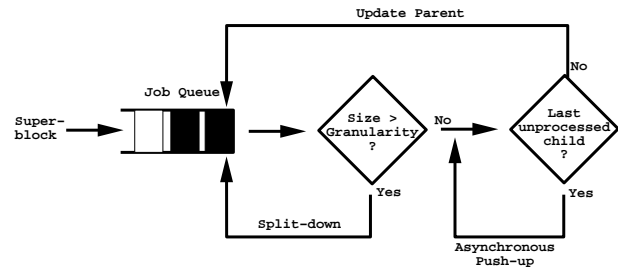


Figure 2: Flow of control of recursive tree construction.

In Figure 2, we outline the general design of our algorithm. After initially adding the superblock of the octree to the empty job queue, the algorithm starts to read the first job from the queue. If the size of this job, which is the block size of the octant, exceeds a certain granularity value, this block is split into its children which are added to the queue. Thereafter, a new job is read from the queue. If the block size is below the granularity value, the processing thread proceeds sequentially with the octant and the associated subtree. This differentiation is necessary in order to guarantee a balance between the parallelization overhead, and the parallelization benefits.

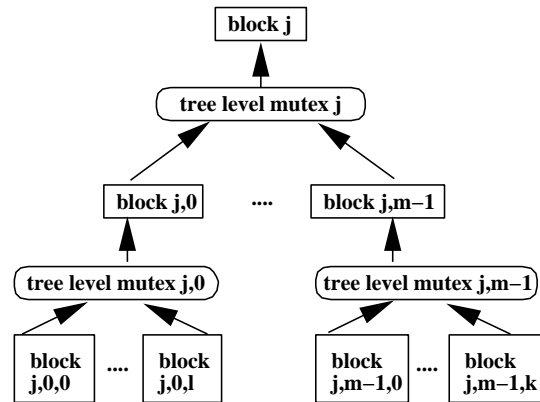


Figure 3: Tree level mutexes.

After processing the octant, we update its parent with the generated information. Subsequently, we check if this octant was the last child of its parent which did not complete its computation. If this condition is matched, the processing thread continues processing the parent block and the flow of control has returned to the parent. Otherwise, the thread simply gets a new job from the queue. We call this semantic an asynchronous push-up (apu). Generally, the update operation introduces a critical section to our algorithms which needs to be protected with a mutex. In our approach, we use a *tree level mutex* which protects only one parent block and its child blocks, resulting in a minimal obstruction for other threads (Fig. 3). This locking mechanism is somewhat similar to *predicate locking* or *tree locking* [7]. However, the flow of update information is dictated by the control flow, which guarantees that local updates do

not corrupt data in parent nodes of the octree. Hence, it is not required to lock other parts of the octree as well, which simplifies the locking mechanism significantly.

Figure 4 outlines the flow of control of the asynchronous push-up. Thread  $t_0$  is splitting the parent  $j$  into  $m$  child blocks, where thread  $t_1$  is processing child  $j, 0$ , thread  $t_2$  is processing child  $j, i$ , and thread  $t_0$  is processing child  $j, m - 1$ . After the completion of child  $j, 0$  and child  $j, m - 1$ , threads  $t_0$  and  $t_1$  get a new job from the job queue. Thread  $t_2$  processes the last uncompleted child of parent  $j$ . Therefore, after completion of child  $j, i$ , thread  $t_2$  performs the asynchronous push-up and continues with parent  $j$ .

Measurements of the time spent for mutex locking show that all potential bottlenecks added by our algorithm – the mutex protected job queue access and the mutex protected asynchronous push-up – turned out to be of no significance. Up to 5% of the octree construction time was used for locking and unlocking of the job queue mutex, while no measurable time was consumed by the tree level mutexes. Comparing these numbers with the saved time due to the parallel construction, we consider this amount as insignificant. However, synchronization costs of the job queue can become significant, if the number of threads gets much higher. At no point during the computation do the threads stall because of an empty job queue. Until the final phase of the octree construction, the queue is always sufficiently filled, even with a large number of threads. However, the construction process introduces heavy memory allocation which limits the scalability of the algorithm. We will discuss this problem in detail in Section 4.

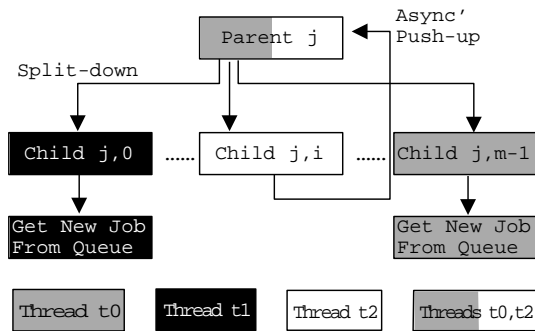


Figure 4: Flow of control of asynchronous push-up (apu).

## Parallel Isosurface Extraction

Following Wilhelms and Van Gelder [25], we store the minimum and maximum isovalues of the voxels of an octant at all levels-of-detail. Therefore, we can rapidly decide if a subtree contains contributing cells (“surface cell”), thus limiting the number of voxels examined by the Marching Cubes algorithm [19]. After assigning the contributing cells to the threads using a round-robin scheme, the threads are generating the polygons which represent the selected isosurface within the volumetric dataset.

Efficient isosurface extraction is an active field of research in visualization. However, our focus is only on parallel construction of recursive tree structures. Consequently, we refer to Section 2 for further information in that field. More details on our straightforward parallel implementation of Marching Cubes can be found in [2].

## 4 Reducing Synchronization Overhead of Parallel Memory Allocation

In this paper, we base our discussion on the pthread implementations on three different memory architectures of tightly coupled, (distributed) shared-memory MIMD systems; two NUMA (Non-Uniform-Memory-Access) systems (SGI Onyx2/Origin2000, SGI Origin200), and one UMA (Uniform-Memory-Access) system (SGI Challenge). All systems show different synchronization behavior, due to their architectural differences.

### SGI Origin200

The Origin200 is a four processor system which is split into two subsystems with each having 512 MB of main memory and two 180 MHz MIPS R10000 CPUs [15]. The memory and two CPUs of one subsystem are connected via a hub chip, implementing a four port crossbar. The two subsystems are connected via a “CrayLink” interconnect between the respective hub chips. The peak performance of the interconnect is 1.44 GB/s<sup>1</sup> Note that the performance deterioration of dataset C on the SGI Origin 200 (in contrast to the SGI Onyx2 and SGI Challenge) is due to swapping.

### SGI Onyx2

The Onyx2 used for our measurements has ten 195 MHz R10000 CPUs. These CPUs are organized on two processor modules, with a total of five node boards[12]. The first processor module contains four node boards, while the second only contains one node board. Each node board contains up to 512 MB main memory and two CPUs. These CPUs, the memory, and the connection to other parts of the Onyx2 are interlinked via a hub chip, implementing a four port crossbar. Two node boards are connected via a six port crossbar, thus interlinking both node boards via a router to the interconnection fabric of the processor modules. This interconnection fabric interlinks both processor modules using a hypercube topology. Similar to the Origin200, the peak performance of the interconnect is 1.6 GB/s.

### SGI Challenge

The SGI Challenge is a UMA architecture system. 3 GB main memory is connected with sixteen 195 MHz R10000 CPUs via a system bus running at 1.2 GB/s [11].

### 4.1 Memory Allocation Strategies

Three different methods for the parallel allocation of memory are examined. These methods produce very different results on the three different architectures. For the measurements<sup>2</sup>, we used six different volume datasets of different origin (Table 1); three large datasets (A, B, C) and three rather small datasets (D, E, F). Dataset A represents a velocity field generated by a computational fluid dynamics (CFD) simulation, where two sides of a fluid filled cavity are heated differently (Fig. 12b). Velocity magnitude is used as an isovalue, while the temperature is mapped as color onto the isosurface. Dataset B is a MRI scan of a human head with special focus on the cerebro-spinal-fluid (CSF) filled cavities (Fig. 13a). Dataset C is a rotational angiography dataset of a fusiform aneurysm of an arterial blood vessel in a human head (Fig. 13b). Both datasets from

<sup>1</sup>The SGI Origin200 is in a way a reduced, two-node board version of the SGI Onyx2/Origin2000 architecture. Therefore, both systems have several common features and characteristics.

<sup>2</sup>We measured the octree construction time with/on 1, 2, 4, 6, 8, 10, 12, 14, and 16 threads/CPUs, the detailed profiling only on 1, 2, 4, (6,) 8, and 16.

Dataset/Size	Octree Depth	#Nodes of Full Octree	#Nodes of BONO	#Contributing cells	#triangles
A: Cavity dataset 191×191×191	7	2,397K 100%	984K 41%	43K 2%	128K
B: MRI Head 258×258×212	8	19,174K 100%	2,044K 11%	103K 1%	859K
C: Angiography 514×514×260	9	153,392K 100%	9,917K 6%	185K 0%	1,554K
D: Vortex 45×45×55	5	37K 100%	3K 7%	1K 4%	10K
E: LOP 38×76×38	6	295K 100%	16K 5%	2K 1%	14K
F: Blunt Fin 40×32×32	5	37K 100%	6K 16%	1K 4%	10K

Table 1: Dataset overview

medical scanners (B and C) are only slightly larger (two elements) than a valid octree block size (256 and 512). Therefore, the next larger size is chosen, resulting in huge memory space savings of the BONO representation. Dataset D represents a vector field of a vortex breakdown, where a fluid is injected in another fluid, where we used the velocity magnitude as an isovalue (Fig. 11a). Dataset E is a simulation of a liquid oxygen flow across a flat plate with a cylindrical post (LOP). The velocity magnitude is used as an isovalue, and the internal energy is mapped as color onto the isosurface (Fig. 12a). Finally, dataset F is an air flow dataset over a flat plate with a blunt fin. The density is used as an isovalue, while the kinetic energy is mapped as color onto the isosurface (Fig. 11b). The first four datasets (A-D) are provided on a regular grid, while the final two datasets are computed on a curvilinear grid. On all datasets,

## Standard Memory Allocation

This initial technique uses the memory allocation functions (malloc, calloc, etc.) of the standard library (stdlib). The thread-safe versions of these functions use a global locking mechanism to guarantee mutual exclusion, usually denoted as a “big lock” [4]. Closer examination of the memory allocation using malloc/calloc shows that this mechanism introduced a significant synchronization overhead (Fig. 5c); approximately 95% of the time spend for memory allocation is used only for synchronization. While synchronization is scaling on the SGI Challenge down to a constant overhead, memory allocation on the SGI NUMA-architecture machines (Origin200 and Onyx2) deteriorates severely. This is due to the need of synchronization of the kernel threads on each CPU of the NUMA-architectures, which is significantly more expensive than the respective synchronization on UMA-architectures. Note that mem-

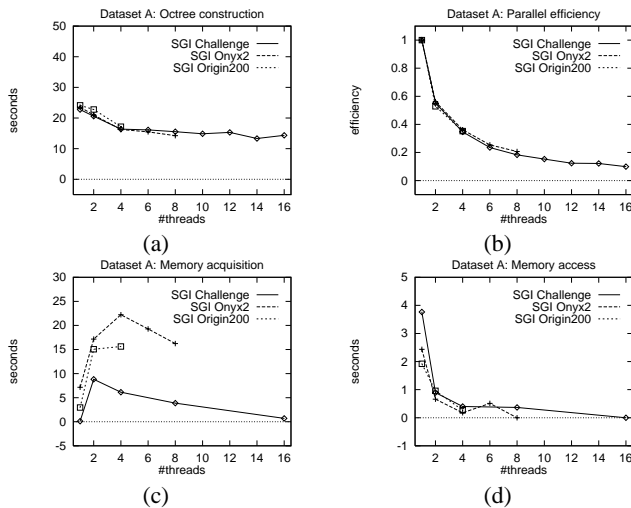


Figure 5: Octree construction of dataset A using standard memory allocation. (a) shows overall construction time, (b) shows the parallel efficiency of the octree construction, (c) shows time spent for memory allocation, and (d) shows a zoom into the time spent for memory access (the latter two are determined by profiling with 1, 2, 4, 8, and 16 CPUs only).

our algorithms showed the same general behavior. However, due to paper size limitations we only look in detail at dataset A, and show general behavior of datasets B and C.

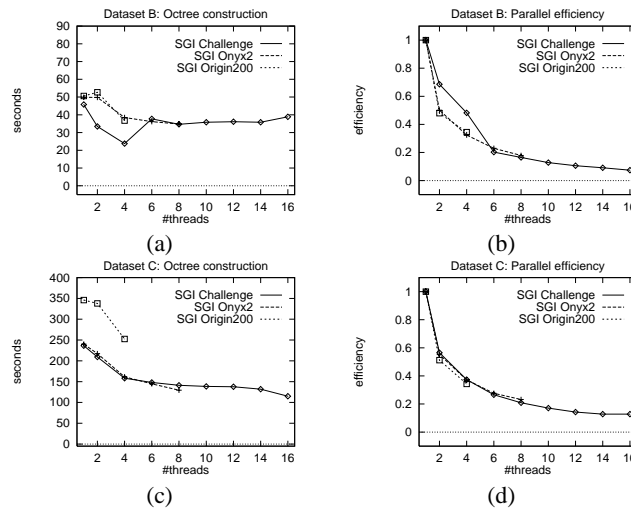


Figure 6: Octree construction of dataset B (a, b) and C (c, d) using standard memory allocation. (a, c) shows overall construction time, (b, d) shows the parallel efficiency of the octree construction.

ory access throughout the whole virtual memory of all three architectures scales nicely, thus exhibiting no varying memory latency between the memory levels.

## Process Global Pre-allocation

The previous experiment showed that the standard thread-safe memory allocation functions introduced an expensive memory-locking mechanism. However, the pthread mutexes used in the experiments suggested that the standard mutex locking mechanism is a faster, and therefore cheaper synchronization mechanism. Consequently, we introduced an alternative memory allocation method where a huge chunk of memory is allocated before entering the parallel region of our code. Later, we assign blocks of this memory to the octants using a customized data-structure, similar to an array of octants. The actual assigning action is protected by a data-structure local mutex, where only one structure is used for the whole construction process. Using this method, we obtained good scaling

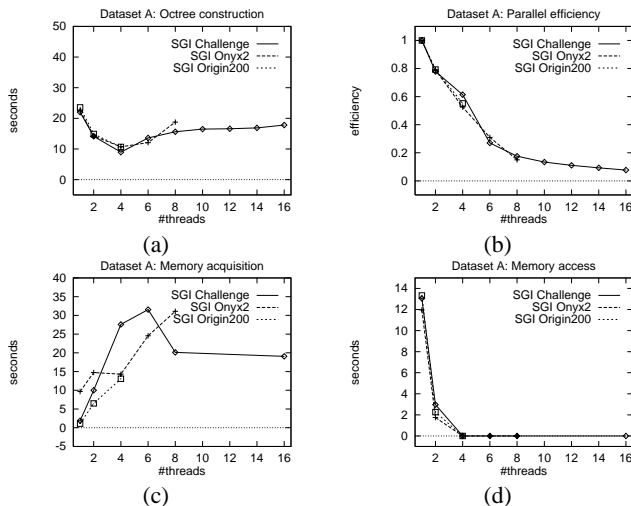


Figure 7: Octree construction of dataset A using process global memory pre-allocation. (a) shows overall construction time, (b) shows the parallel efficiency of the octree construction, (c) shows time spent for memory allocation, and (d) shows a zoom into the time spent for memory access (the latter two are determined by profiling with 1, 2, 4, 8, and 16 CPUs only).

on the SGI Origin200 architecture (Fig. 7 and 8). Memory synchronization in particular scaled down to a fraction of the original amount. The SGI Onyx2 architecture showed a different picture. While the four CPU Origin200 only needs one additional crossbar hop to the CPUs on the other subsystem, access to all other CPUs of the Onyx2 requires up to two hops via the interconnection fabric, thus increasing the synchronization overhead similar to the standard memory allocation scheme. On the SGI Challenge, increasing memory requests of the threads increased the costs for synchronization. In contrast to memory locking using the standard library functions, global mutex locking does not scale, resulting from increasing contention of the growing number of threads. Similar to the measurements of the previous standard allocation approach, memory access does scale without any measurable latency difference between the different memory levels.

## Thread Local Pre-allocation

From the previous experiment, we learned that mutex locking using only one global mutex can increase synchronization costs due to high contention. Consequently, we need to reduce this contention by using multiple locks. Due to the fact that all systems are shared-

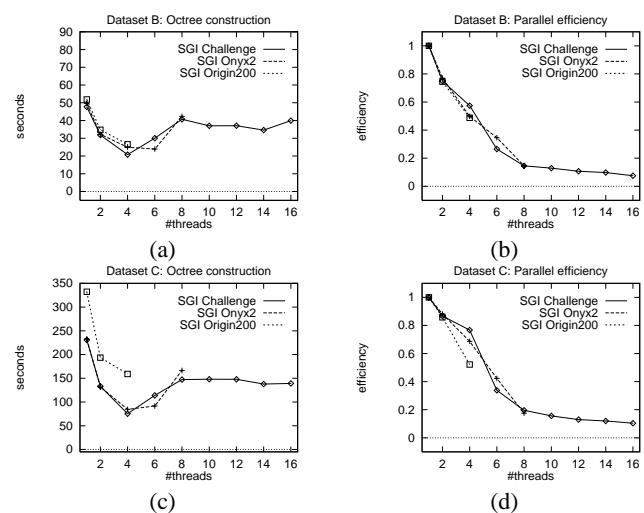


Figure 8: Octree construction of dataset B (a, b) and C (c, d) using process global memory pre-allocation. (a, c) shows overall construction time, (b, d) shows the parallel efficiency of the octree construction.

memory systems and that memory access through the interconnect always scaled nicely, despite the interconnection technology (bus or crossbar), this approach uses the previous pre-allocating data-structure for each thread. Therefore, specific memory locking is not necessary.

Figures 9 and 10 shows the results of this approach. Memory allocation time (including the synchronization overhead) could be reduced to a fraction of the previous amounts on all three systems. It scales throughout all CPUs, resulting in a balanced parallelization of the complete construction process. Furthermore, the measurements of the memory access show no evidence of a varying memory latency, which is consistent with the previous measurements.

## 5 Conclusion and Future Work

In this paper, we presented an algorithm for the parallel construction or reconstruction of recursive tree structures. Although this algorithm was discussed only for octrees, it is also suited for quadtrees, k-D-trees, BSP-trees, or other recursive tree structures.

In contrast to many statements in NUMA thread programming, cross-node memory access introduced no measurable latency or bottleneck in our application. At no time, did we find evidence of access penalties, once the memory access was leaving the lower hierarchy level of node board or processor module local memory. However, this cross-node memory access latency might become more significant using 32 or more CPUs on larger (but unfortunately to the author not available) systems.

Yet, synchronization turned out to be expensive on NUMA architectures. This potential bottleneck emerged during dynamic and parallel memory allocation. The necessary locking mechanism represented a significant slow-down in thread-parallel applications. We discussed three different approaches which address this problem. The final approach using a thread local pre-allocation scheme solved the problem on three different architectures and produced a scaling scheme for the construction of tree hierarchies.

Compared to the memory synchronization overhead, the mutex protected job queue access represented no significant overhead. Nevertheless, using even more threads, this access might introduce

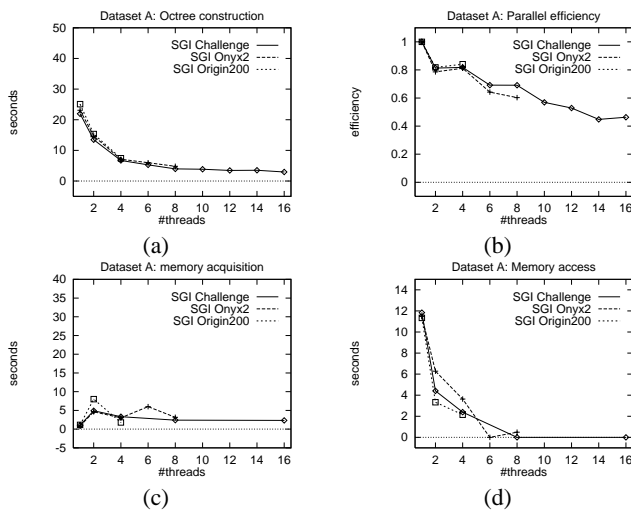


Figure 9: Octree construction of dataset A using thread local memory pre-allocation. (a) shows overall construction time, (b) shows the parallel efficiency of the octree construction, (c) shows time spent for memory allocation, and (d) shows a zoom into the time spent for memory access (the latter two are determined by profiling with 1, 2, 4, 8, and 16 CPUs only).

a more significant bottleneck. Therefore, future work will focus on a distributed job queue. Another focus for future work will be support for fast visibility queries in large dynamic polygonal datasets, which require a fast reconstruction of the hierarchy.

## Acknowledgments

This work was supported by the MedWis program of the German Federal Ministry of Education and Research. We would like to thank Arie Kaufman of the Center of Visual Computing at the SUNY Stony Brook for using the SGI Challenge at Stony Brook, Stephan Braun and Heinrich Bühlhoff for support and use of the SGI Onyx2 at the Max-Planck-Institute for Biological Cybernetics. Furthermore, we thank Michael Doggett for proof-reading this paper. The cavity dataset is courtesy of the Institute for Fluid Dynamics of the University of Erlangen-Nürnberg, the MRI and the angiography dataset is courtesy of the Department of Neuroradiology of the University Hospital at Tübingen, and the liquid oxygen post and blunt fin datasets are courtesy of NASA Ames.

## References

- [1] C. Bajaj, V. Pascucci, and D. Schikore. Fast Isocontouring for Improved Interactivity. In *Proc. of Symposium on Volume Visualization*, pages 39–46, 1996.
- [2] D. Bartz, R. Grosso, T. Ertl, and W. Straßer. Parallel Construction and Isosurface Extraction of Recursive Tree Structures. In *Proc. of WSCG'98*, volume III, pages 479–486, 1998.
- [3] J. Bentley. Multidimensional Binary Search Trees Used for Associative Search. *Communications of the ACM*, 18(9):509–516, 1975.

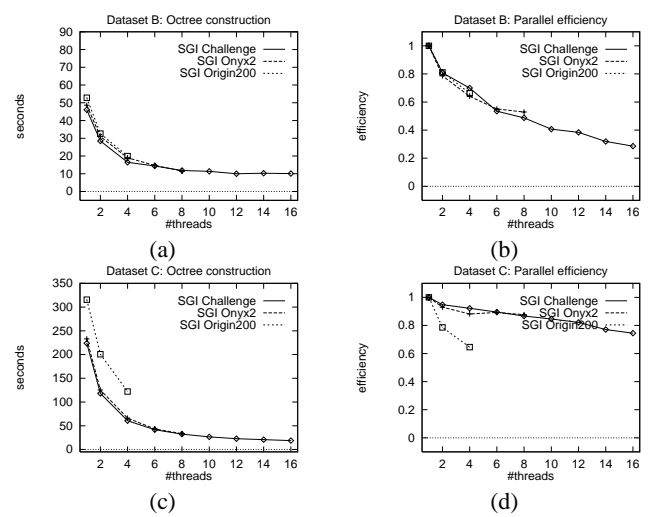


Figure 10: Octree construction of dataset B (a, b) and C (c, d) using thread local memory pre-allocation. (a, c) shows overall construction time, (b, d) shows the parallel efficiency of the octree construction.

- [4] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, 1997.
- [5] P. Cignoni, P. Parino, E. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 2(12), 1997.
- [6] H. Fuchs, Z. Kedem, and B. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Proc. of ACM SIGGRAPH*, pages 124–133, 1980.
- [7] H. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann Publishers, San Francisco, 1993.
- [8] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.
- [9] R. Grosso, T. Ertl, and R. Klier. A Load-Balancing Scheme for Parallelizing Hierarchical Splatting on a MPP System with Non-uniform Memory Access Architecture. In *Proc. of High Performance Computing for Computer Graphics and Visualization*, pages 125–134, 1995.
- [10] P. Hanrahan, D. Salzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. In *Proc. of ACM SIGGRAPH'93*, pages 197–206, 1993.
- [11] Silicon Graphics Inc. Power Challenge. Technical report, Silicon Graphics Inc., Mountain View, 1994.
- [12] Silicon Graphics Inc. Onyx2 Reality and Onyx2 InfiniteReality. Technical report, Silicon Graphics Inc., Mountain View, 1997.
- [13] T. Itoh and K. Koyamada. Automatic Isosurface Propagation Using An Extrema Graph and Sorted Boundary Cell Lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1995.

- [14] A. Kela and M. Wynn. Parallel Computation Of Exact Quadtree and Octree Approximations on Distributed Memory Multiprocessors. In *Proc. of 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 1193–1196, 1989.
- [15] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 241–251, 1997.
- [16] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proc. of ACM SIGGRAPH'91*, pages 285–288, 1991.
- [17] M. Levoy. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.
- [18] Y. Livnat, H. Shen, and C. Johnson. A Near Optimal Iso-surface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 1(2), 1996.
- [19] W. Lorensen and H. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proc. of ACM SIGGRAPH*, pages 163–169, 1987.
- [20] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, 1994.
- [21] R. Shekhar, W. Fayyad, R. Yagel, and J. Frederick. Octree-Based Decimation of Marching Cubes Surface. In *Proc. of IEEE Visualization*, pages 287–294, 1996.
- [22] H.-W. Shen, C. Hansen, Y. Livnat, and C. Johnson. Isosurfacing in Span Space with Utmost Efficiency (ISSUE). In *Proc. of IEEE Visualization*, pages 287–294, 1996.
- [23] O. Sudarsky and C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. In *Proc. of Eurographics'96*, pages 249–258, 1996.
- [24] L. Swift, T. Johnson, and E. Livadas. Parallel Creation of Linear Octrees from Quadtree Slices. *Parallel Processing Letters*, 4(4):447–453, 1994.
- [25] J. Wilhelms and A. Van Gelder. Octrees for Faster Isosurface Generation. *ACM Transaction on Graphics*, 11(3):201–227, 1992.
- [26] J. Wilhelms, A. Van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and Parallelizable Direct Volume Rendering for Irregular and Multiple Grids. In *Proc. of IEEE Visualization*, pages 57–64, 1996.
- [27] C. Wittenbrink and K. Kim. Data Dependent Optimizations for Permutation Volume Rendering. In *Proc. of SPIE Visual Data Exploration and Analysis V*, pages 284–294, 1998.

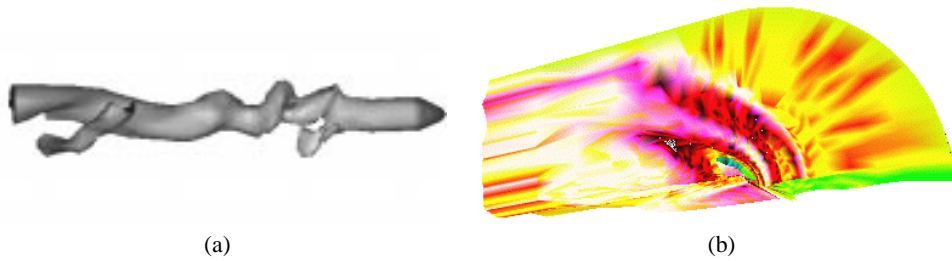


Figure 11: (a) Vortex breakdown of a fluid which is injected into another fluid; (b) Blunt Fin: Density is used as an isovalue, while the kinetic energy is mapped as color onto the isosurface.

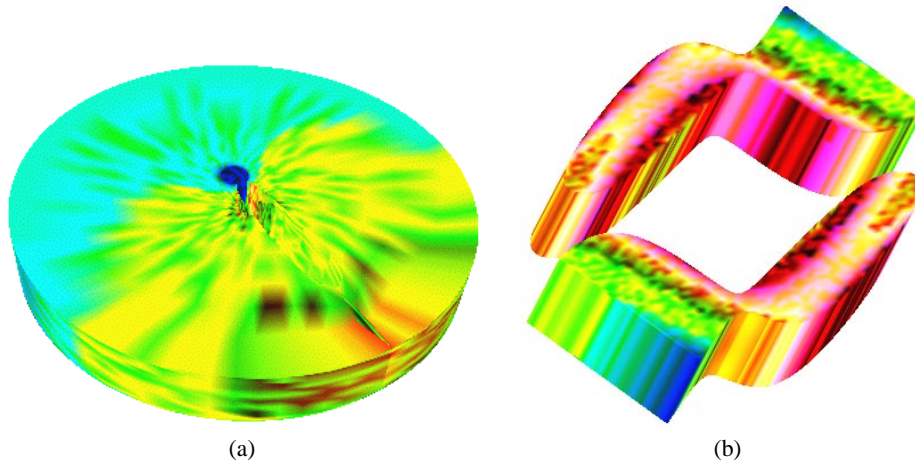


Figure 12: (a) Liquid Oxygen Post (LOP): Velocity magnitude is used as an isovalue, while the internal energy is mapped as color onto the isosurface; (b) Dataset A: Heated Cavity: Velocity magnitude is used as an isovalue, while the temperature is mapped as color onto the isosurface.

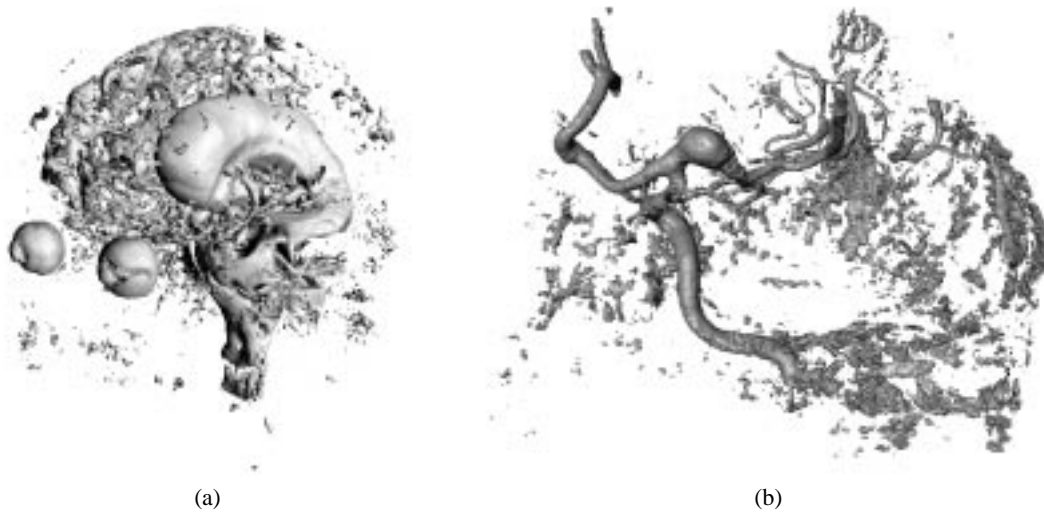


Figure 13: (a) Dataset B: CSF-filled cavities in MRI scan of a human head; (b) Dataset C: Reconstructed arterial vessel from rotational angiography (cropped snapshot with different isosurface compared to experiments).