

Asynchronous Parallel Construction of Recursive Tree Hierarchies*

Dirk Bartz and Wolfgang Straßer**

WSI/GRIS,
University of Tübingen,
Auf der Morgenstelle 10/C9,
D72076 Tübingen, Germany
<http://www.gris.uni-tuebingen.de>

Abstract. Multi-resolution methods are widely used in scientific visualization, image processing, and computer graphics. While many applications only require an one-time construction of these 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, arise which often need a dynamic reconstruction of a scene database.

In this paper, we address several problems of the construction or reconstruction of recursive tree hierarchies in parallel. In particular, we focus on parallel dynamic memory allocation and the associated synchronization overhead.

Keywords: Parallel hierarchies, recursive tree structures, octrees, memory synchronization, 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 [4], to reduce the light interaction between different parts of a scene [6], and so forth. Among the most popular spatial multi-resolution representations are recursive tree structures like quadtrees and octrees [13], and binary-space-partitioning trees (BSP-trees) [3]. Unfortunately, the construction or reconstruction of these representations is very costly. Therefore, we proposed to perform this process in parallel [1] by decoupling the recursive parent/children relationship of the tree elements. However, dynamic memory allocation is limiting the scalability of our algorithm by introducing additional synchronization overhead. In this paper, we discuss three approaches to remove this bottleneck on three different MIMD architectures.

Our paper is organized as follows: We first outline the background and the related work of our paper. Next, we briefly describe the basic algorithm for the parallel and

* To appear in *Parallel Computation, the ACPC 1999 proceedings*

** Email: {bartz, strasser}@gris.uni-tuebingen.de

balanced tree (re-)construction and isosurface extraction. 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, image processing, and computer graphics. Of special importance are schemes like quadtrees - which represent a regular two-dimensional subdivision - and BSP-trees - which represent an irregular binary subdivision of arbitrary dimensions. Although our results are generally valid for all recursive tree structures, in this paper we only focus on the three-dimensional counterpart of quadtrees, the octrees.

An octree is a hierarchical spatial data-structure to represent three-dimensional volumetric data at different levels of details [13]. Starting with the superblock - representing the whole dataset - each octant (an octree block) 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.

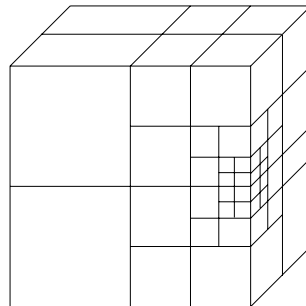


Fig. 1. Octree.

Due to the subdivision, the size of each octant is a power of two. Unfortunately, datasets usually do not have a size of this scheme. Therefore, some octants are “empty” - they do not intersect with the dataset, according to the alignment of the dataset within the octree. In order to save space, we use a minimal octree. The minimal octree approach enumerates only the octants - and their children - that are not empty.

Octrees are used in several applications to provide a multi-resolution representation. Laur and Hanrahan presented an octree-based scheme for hierarchical splatting [10]. Splats of different size and shape are 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 [5]. In their approach, a static parallelization is used, where

up to eight threads are processing up to eight children blocks of the superblock. Greene et al. use an octree and an image pyramid for visibility queries in large polygonal environments [4]. Shekhar et al. use an octree representation of a volumetric dataset to generate a block-oriented polygon reduction scheme of its isosurface [14]. Levoy presented an approach to accelerate ray casting by using octrees [11], where coherent (non-contributing) data can be skipped rapidly.

As an example application of our algorithm, an isosurface is extracted from a volumetric dataset which represents a surface with the same scalar value (isovalue) throughout this dataset. We are using a parallel implementation of the approach by Wilhelms and van Geldern [16]. 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 skipped rapidly. After selecting all these contributing voxels of these blocks, the isosurface is generated.

The construction of a multi-resolution representation 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. Furthermore, several applications require fast reconstructions of octrees (or other recursive tree structures). Changes of the color table or the transfer functions in volume rendering applications require a reconstruction of the used hierarchical representations [5, 11]. In occlusion culling applications, moving objects cause a partial reconstruction of the scene representation [15].

3 Parallel Construction of 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 the structure. A simple distributed top-down subdivision, as suggested for the first stage, only provides the top-down information; 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. In our approach, we solve this problem by combining a central workload splitting job queue and our new asynchronous push-up [1].

In Figure 2, we outline the general design of our algorithm. After initially adding the superblock of the octree¹ to the empty job queue, the algorithm starts to read the first job from the queue. If the size of this job, the block size of the octant, exceeds a certain granularity value, this block is splitted into its children which are inserted into

¹ As already mentioned earlier, we focus in this paper on octrees.

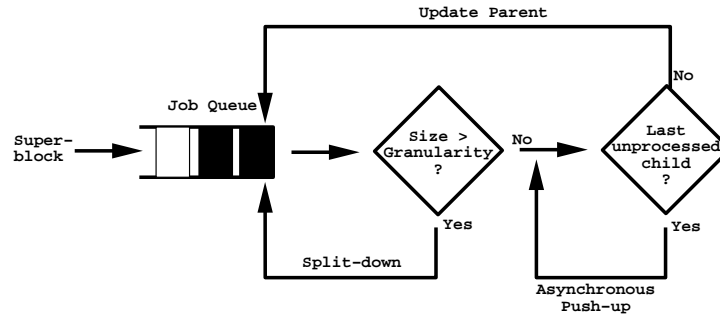


Fig. 2. General design of recursive tree construction.

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 queue and synchronization overhead, and the parallelization benefits.

After processing the octant, we update its parent. Subsequently, we check if this octant was the last child of its parent which had not completed its computation. If it is the last uncompleted child, 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).

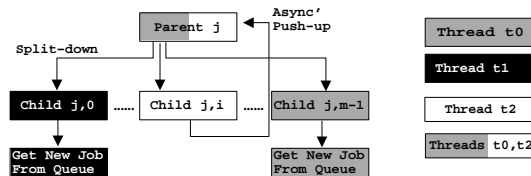


Fig. 3. Asynchronous Push-Up.

Figure 3 outlines the flow of control of the asynchronous push-up. Thread t_0 is splitting the parent j into m children 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 .

In [1], we 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. Furthermore, 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.

Parallel Isosurface Extraction

Following Wilhelms and van Geldern [16], we store the minimum and maximum iso-values of the voxels at all levels-of-detail. Therefore, we can rapidly decide if a subtree contains contributing cells², thus limiting the number of voxels examined by the Marching Cubes algorithm [12].

After assigning the contributing cells to the threads using a round-robin scheme, the threads are generating the polygons which represent the isosurface within the volumetric dataset. Due to page limitation, we refer to [12] for more details on the Marching Cubes algorithm, and to [1] for more details on the measurements of the parallel Marching Cubes implementation.

4 Reducing Synchronization Overhead of Parallel Memory Allocation

In [1], we discussed the presented algorithm on two different memory architectures; an NUMA-architecture³ (Convex SPP 1600) and on an UMA-architecture⁴ (SGI Challenge). In this paper, we base our discussion on the pthread implementation on three different memory architectures; two NUMA systems (SGI Onyx2, SGI Origin200), and one UMA 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 [9]. 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.

SGI Onyx2: The Onyx2 used for our measurements has 10 195 MHz R10000 CPUs. These CPUs are organized on two processor modules, with a total of five node boards

² A cell is called contributing if the isosurface passes through the cell.

³ Non-Uniform-Memory-Access

⁴ Uniform-Memory-Access

[8]. 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. The 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 1.44 GB/s.

SGI Challenge: The SGI Challenge is an UMA architecture system. 3 GB main memory is connected with 16 195 MHz R10000 CPUs via a global bus running at 1.2 GB/s [7].

4.1 Memory Allocation Strategies

Three different methods for the parallel allocation of memory are examined. Two of these methods produce very different results on the three different architectures. For the measurements, we used two different volume datasets of different origin (Table 1). 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. Dataset B is a MRI scan of a human head with special focus on the ventricle system of the human brain. On both datasets, our algorithms showed the same behavior. Therefore, we only look in detail at dataset B and present only the general results of dataset A.

Dataset/Size	Total number of cells in octree	Contributing cells	#triangles
A: Cavity dataset 191x191x191	6,968K 100%	2,204K 6.6%	4,960K
B: Ventricle system 258x258x212	14,112K 100%	1,781K 12.6%	1,523K

Table 1. Datasets

Standard Memory Allocation This technique uses the memory allocation functions of the standard library (stdlib). The thread-safe versions of these functions are using a global locking mechanism to guarantee mutual exclusion, usually denoted as a “big lock” [2].

Closer examination of the memory allocation using malloc/calloc shows that this mechanism introduced a significant synchronization overhead (Fig. 4); 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 the constant overhead, memory allocation on the SGI NUMA-architecture machines (Origin200 and Onyx2) deteriorates severely.

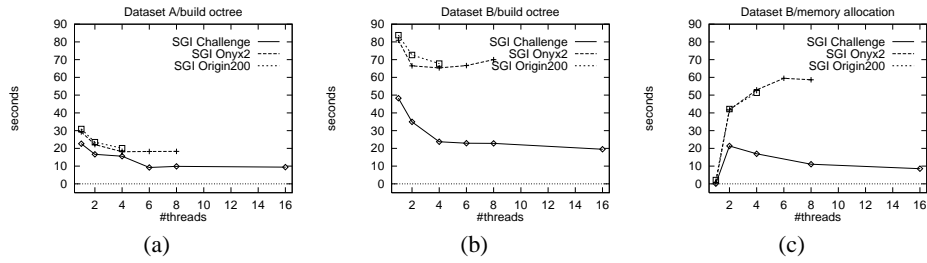


Fig. 4. Octree build operation of dataset A (a) and B (b and c) using standard memory allocation. (a) and (b) show overall construction time, (c) shows memory allocation (allocation times are determined by profiling).

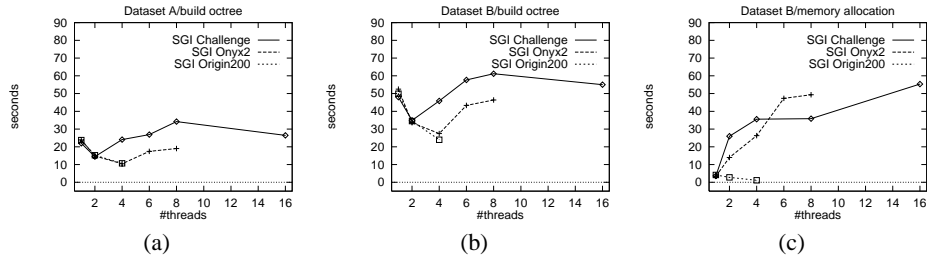


Fig. 5. Octree build operation of dataset A (a) and B (b and c) using global customized memory allocation. (a) and (b) show overall construction time, (c) shows memory allocation (allocation times are determined by profiling).

Global Customized Allocation The previous experiment showed that the standard thread-safe memory allocation functions introduced an expensive memory-locking mechanism. However, the 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. We pre-allocate a huge chunk of memory 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 octant arrays. The actual assigning action is protected by a data-structure local mutex. However, only one structure is used for the whole construction process.

Using this method, we obtained good scaling on the SGI Origin200 architecture (Fig. 5). 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. On the SGI Challenge, increasing memory requests of the threads increase the costs of synchronization. In contrast to memory locking using the standard library functions, global mutex locking does not scale. This probably results from increasing contention of the growing number of threads.

Local Customized 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 facts that all systems are shared-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, a specific memory locking is not necessary.

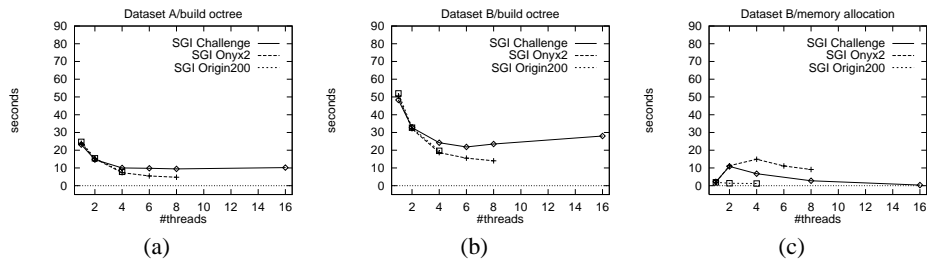


Fig. 6. Octree build operation of dataset A (a) and B (b and c) using local customized memory allocation. (a) and (b) show overall construction time, (c) shows memory allocation (allocation times are determined by profiling).

Figure 6 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. Furthermore, memory allocation scales throughout all CPUs, resulting in a balanced parallelization of the complete construction process.

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 to quadtrees, BSP-trees, or other recursive tree structures.

A main bottleneck of this algorithm is the global memory locking mechanism. Consequently, we presented three different approaches for solving this problem. The final approach using a local customized allocation scheme solved the problem on three different architectures and produced a scaling scheme for the construction of tree hierarchies.

However, while using an increasing number of threads, the job queue access might introduce a more significant overhead. 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.

While most CFD datasets are based on a curvilinear grid, our current applications are only based on rectilinear grids. However, for octrees, only the topology needs to be rectilinear, but not the geometry. Therefore, we can apply our scheme to curvilinear grids as well, which is another field of future work.

Acknowledgments

Numerous people helped us preparing this paper. Especially, we would like to thank Roberto Grosso and Thomas Ertl of the Computer Graphics Group at the University of Erlangen-Nürnberg for the collaboration on previous work. We 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 using of the SGI Onyx2 at the Max-Planck-Institute for Biological Cybernetics. The cavity dataset is courtesy of the Institute for Fluid Dynamics of the University of Erlangen-Nürnberg. Last but not least, we thank Michael Meißner and Michael Doggett for proof-reading.

This work was supported by the MedWis program of the German Federal Ministry of Education, Science, Research, and Technology.



Fig. 7. Vortex breakdown of a fluid which is injected in another fluid.

References

1. 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, 1998.

2. D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, 1997.
3. 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.
4. N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.
5. 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.
6. P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In *Proc. of ACM SIGGRAPH'93*, pages 197–206, 1993.
7. Silicon Graphics Inc. Power Challenge. Technical report, Silicon Graphics Inc., Mountain View, 1994.
8. Silicon Graphics Inc. Onyx2 Reality and Onyx2 InfiniteReality. Technical report, Silicon Graphics Inc., Mountain View, 1997
9. J. Laudon and D. Lenoski. System overview of the SGI Origin 200/2000 product line. Technical report, Silicon Graphics Inc., Mountain View, 1997
10. D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. In *Proc. of ACM SIGGRAPH'91*, pages 285–288, 1991.
11. M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.
12. W. Lorensen and H. Cline. Marching Cubes: A high resolution 3D surface construction algorithm. In *Proc. of ACM SIGGRAPH*, pages 163–169, 1987.
13. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, 1994.
14. 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.
15. 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.
16. J. Wilhelms and A. van Geldern. Octrees for faster isosurface generation. *ACM Transaction on Graphics*, 11(3):201–227, 1992.