To appear in Proc. WSCG'98, Vol. III, Plzen, 1998

PARALLEL CONSTRUCTION AND ISOSURFACE EXTRACTION OF RECURSIVE TREE STRUCTURES

Dirk Bartz, Wolfgang Straßer
WSI/GRIS, University of Tübingen
Auf der Morgenstelle 10/C9
D72076 Tübingen, Germany
Email: {bartz, strasser}@gris.uni-tuebingen.de

Roberto Grosso, Thomas Ertl
IMMD9, University of Erlangen-Nürnberg
Am Weichselgarten 9
D91058 Erlangen, Germany
Email: {grosso, ertl}@immd9.informatik.uni-erlangen.de

ABSTRACT

The visualization of volumetric datasets is usually limited by the amount of memory and processing power of computer systems. Several multiresolution methods have been developed in order to adapt the necessary work; recursive spatial tree structures, such as octrees, are among the most popular. The exploration of a dataset frequently requires a change of parameters, such as color table entries or isovalues. Therefore, the costly update of an octree becomes necessary. To overcome this drawback, we propose the parallel construction of octrees to improve their suitability for interactive volume visualization. Based on the thread model of the shared-memory paradigm, we developed a scheme for a balanced parallel construction. We apply this new scheme to generate isosurfaces in parallel, using the Marching Cubes algorithm.

Keywords: Volume visualization, octrees, hierarchical data structures, thread model, shared-memory paradigm

1 INTRODUCTION

Tree-recursive data structures like quadtrees, or their three-dimensional counterpart, the octrees, are widely used in image processing and computer graphics [Gross95], [Hanra93], [Green93], [Laur91]. Unfortunately, the construction or reconstruction of these data structures is very expensive. The use of parallel computers suggests a reduction of the tree construction time. However, due to the recursive nature of the tree structures, parallelization is difficult.

In [Gross95] a static parallelization of the first sublevel of the tree is proposed; each child of the super block of the tree is processed in parallel. The balance of this method depends on the regularity of the dataset, therefore, some of the processors may become idle very soon, while the others are still busy.

The development of a balanced scheme without any

knowledge of the data is a difficult task. Apart from the presented solution, we know of no other scheme at the moment.

The main problem is the recursive parent/children relation of tree structures that is necessary for the complete computation. A balanced parallelization requires a decoupling of this relationship in order to distribute the work to all processors. We achieve this goal by processing the tree blocks in a partial order; all children of a block are processed before the parent block is completed, where the last processed child triggers the completion of the parent.

After the construction of the octree, we apply a load-balancing scheme to generate a distributed job list, which is processed by a parallel version of the Marching Cubes isosurface algorithm [Loren87]. This application is only one of many. Moreover, a lot of work has been published on parallel Marching Cubes. Therefore, we will describe our approach only very briefly.

Although our new scheme is valid for general recursive tree structures, we focus on octrees.

Background and Related Work

An octree is a hierarchical, spatial data structure to represent 3D-data at different levels of details [Samet94]. Starting with the so called superblock - representing the whole dataset - each octant (an octree block) is subdivided into eight children blocks. Each of these children blocks has a size that is half as large as the size of the parent (Fig. 1).

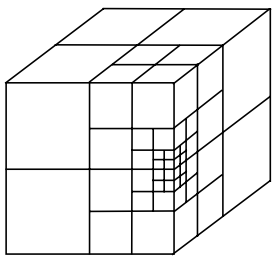

Fig. 1: Octree

This subdivision is performed until the lowest level is reached, where each block represents eight voxels. 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 multiresolution representation. Laur and Hanrahan presented an octree-based scheme for hierarchical splatting [Laur91]. Splats of different size and shape are used, according to the standard deviation of the color values of the different octree blocks. Greene et.al. use an octree and an image pyramid for visibility queries in large polygonal environments [Green93]. In [Shekk96], Shekkar et. al. use an octree representation of a volumetric dataset to generate a block-oriented polygon reduction scheme of its isosurface.

In our approach, we follow Wilhelms and van Geldern [Wilhe92]. By storing the minimum and maximum values of the voxels at each block of the octree, the blocks which do not intersect with the isosurface can be skipped rapidly. After selecting all contributing voxels of these blocks, the isosurface is generated.

Apart from hierarchical methods, a variety of thread-based algorithms for the visualization of volumetric datasets exist. Nieh and Levoy [Nieh92] propose an image space parallel ray casting to visualize the structures of the dataset. Additionally, other parallel volumetric methods have been examined [Singh94], such as the octree-using hierarchical radiosity approach of Hanrahan et. al. in [Hanra93].

Koning et. al. presented an approach similar to Nieh’s [Konin96]. The algorithms were implemented and measured on a Convex SPP 1000 and on a SGI Challenge. However, the replication of the dataset through all hypernodes of the Convex limits the feasible size of the datasets severely. Therefore, this technique is not applicable to our approach.

Our paper is organized as follows: in section two, we discuss our approach for a parallel and balanced construction of a recursive tree structure. Section three briefly presents an application of our scheme and is followed by our results in section four. Ultimately, we state our conclusion in section five.

2 PARALLEL OCTREE CONSTRUCTION

The main contribution of this paper is a scheme for an asynchronous, balanced and scalable parallel construc-

tion of a recursive tree structure, in our case an octree. We achieve this by combining the well-known concept of a workload-splitting job queue and our new asynchronous push-up.

In general, octrees 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 can 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/children 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.

We solve the problem by triggering the push-up of the parent with the completion of the processing of the last child.

First stage: split down

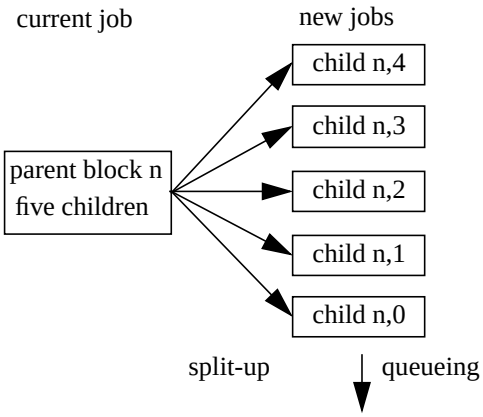

Fig. 2: Split-up and queueing

In an initial step, the superblock of the octree is added to the job queue. After being started, each thread reads a job from the queue and checks the size of the octree block of the job. If this size is above a specified granularity value, the thread splits this octant into children of smaller size, adds these children blocks to the job queue (Fig. 2), and gets a new job from this job queue.

If the octant's size is not above this specified value, the thread proceeds with this block and all its children sequentially. This differentiation is necessary to guarantee a balance between the queue and synchronization overhead, and the parallelization benefits.

Second stage: asynchronous push up

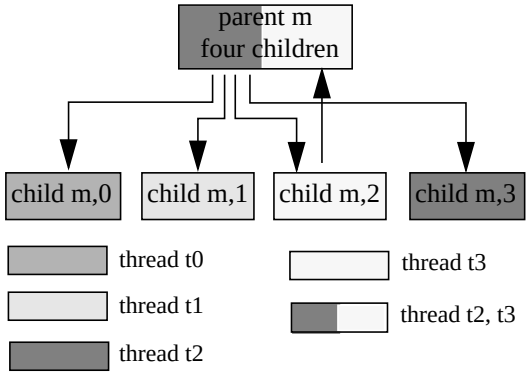

Fig. 3: Asynchronous push-up at child m,2 and thread t3

As mentioned before, the decoupling of the parent/children relation in recursive data structures is crucial for the successfully balanced parallelization of the construction process. However, this relationship must still be preserved. We obtain this goal with an asynchronous push-up.

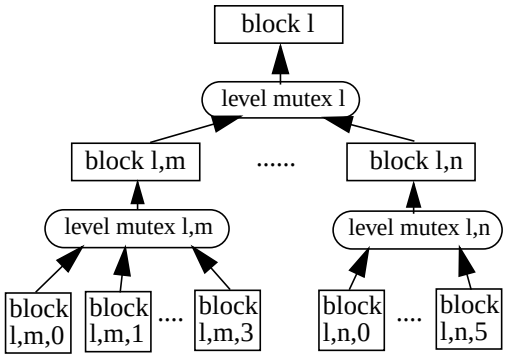

Fig. 4: Level mutexes

We provide each parent block with a counter of all valid children blocks and result fields for these children of the parent block. Each child that has finished its computations updates the appropriate result field of its parent, decrements the counter, and the thread - which processed this child - requests another job from the job queue. As soon as a thread realizes that it is

processing the last uncompleted child, the thread continues processing the parent of this child (thread t3 in Fig. 3). We call this semantic the asynchronous push-up.

The access to the counter and the children result fields is protected by a mutex. We call this mutex a level mutex. The level mutex only protects one parent. Therefore, we obtain an optimized exclusive access and a minimal obstruction for other threads (Fig. 4).

Discussion

Our algorithm has two potential bottlenecks; the job queue and the asynchronous push-up. Both are protected by mutual exclusion.

Optimally, each job generates eight additional jobs. Therefore, the queue contains a rapidly growing number of jobs until the algorithm reaches the granularity of this process. This number depends on the number of threads that were started and the size of the dataset. Apart from the access to the mutex protected job queue, the bottleneck of the queue arises from a possible undersaturation of jobs in the queue. Due to our experiments, it turned out that this happens only in the final phase, while the threads are waiting for the shutdown notification from the thread which is processing the superblock. The second bottleneck, the level mutex of the asynchronous push-up, is only within the direct siblings of one parent block. Although the split-down stage of the construction is realizing a kind of breadth-first order of processing, due to the parallel processing, the order is soon completely unsynchronized.

In the experiments, the time needed for locking and unlocking of all of the mutexes increased from 0.6% to 5% of the octree construction time[1], each time the number of threads/processors doubled. Compared to the saved time, we consider this amount insignificant.

Overall, our closer examination showed that the construction of the octree is a highly balanced process, even with unbalanced trees.

3 PARALLEL ISOSURFACE EXTRACTION

After the construction or the reconstruction of the octree, we need to calculate the contributing cells. In our case, the contributing cells are the cells which are intersected by our isosurface, that is our isovalue is between the minimal and the maximal voxel value of our cell.

In order to generate a load-balanced work distribution, we recursively and sequentially traverse the octree and select all contributing bottom-level blocks[2] into a job queue for each thread. Depending on the number of available threads, we assign the selected blocks in a round-robin manner. Using this scheme, the workload of the threads only differs by one block at most. Considering the usually large number of selected blocks, we have generated a balanced work distribution.

After the load-balancing, each thread starts its own Marching Cubes process to compute the isosurface in its assigned cells and stores the triangles in a FIFO-queue data structure. Due to the distributed job queues, no additional overhead is introduced.

4 RESULTS

Our measurements were performed on two different memory architectures; on a SGI Challenge and on a Convex SPP 1600 (Table 1).

Architecture	SGI Challenge	Convex SPP 1600
#Processors/ Hypernodes	16/-	16/2
Processor	64bit 194 MHz R10000	32bit 120 MHz HP PA7200
Memory Architecture	UMA physical shared-memory	NUMA virtual-shared-memory
High Level Interconnect	./.	Toroidal bus 4 x 600 MB/s, 2µs latency
Low Level Interconnect	Global Bus 1.2 GB/s 200 ns latency	5 port Crossbar 1.25 GB/s, 500ns latency

Table 1: Architectures

SGI Challenge

The SGI Challenge is implementing a physical shared-memory scheme on a UMA[3] architecture. All 16 processors are connected via a 1.2 GB/s system bus, using up to 3.0 GB of memory. For our implementation, we used the pthread library of SGI.

[1]The increase on the SGI Challenge was slightly lower.

[2]Each bottom-level block contains eight voxels.

[3]Uniform-Memory-Access

Convex SPP 1600

In contrast to the SGI Challenge, the Convex SPP is a dedicated MIMD parallel computer, implementing a virtual-shared-memory scheme on a NUMA[1] architecture. It consist of several virtual machines - so called subcomplexes - which can be considered as independent computers. We used a 16 processors/two hypernodes subcomplex with 1.3 GB of virtual-shared-memory. The processors within one hypernode are connected via a five port crossbar at 1.25 GB/s and a memory latency of 500 ns. The hypernodes are connected with a toroidal bus at 2.5 GB/s and with a memory latency of 2 µs. The CPS-library (Compiler Parallel Support) is used as implementation of the thread model.

Discussion

We measured the performance of our algorithms on four different cartesian grid datasets; two medical and two CFD datasets. A is an abdominal CAT-scan of a male patient, and B is a MRI-scan of a human head. Datasets C and D are vortices of two different fluids.The analysis shown in table 3 and table 4 are performed using dataset B on the Convex, and dataset A on the SGI Challenge. Two measures are provided; the wall clock time[2] of the experiments and the parallel efficiency

$$e = t_{seq} / (n_{threads} \cdot t_{parallel}) \qquad (1)$$

where t_{seq} is the sequential execution time, $n_{threads}$ the number of used threads, and $t_{parallel}$ the parallel execution time.

Within one hypernode on the Convex, memory allocation is limiting the construction of the octree and the isosurface extraction using the Marching Cubes[3] algorithms. While the parallel efficiency of both phases is in the mid-nineties, the memory allocation is a mutex protected sequential operation. Therefore, its contribution to both of the phases of our process is increasing from 5.1% to 9.4% of the construction phase (Table 3), and from 4.5% to 15.6% of the extraction phase (Table 4). The memory access to the data volume after its allocation scales nicely with an efficiency always above 90%.

Dataset/ size	Total number of cells in octree	Selected cells	#Triangles
A: Patient abdomen 514x514x183	48,348K 100%	3,204K 6.6%	4,960K
B: MRI Head 256x256x107	7,012K 100%	96K 1.4%	955K
C: Cavity vortex 191x191x191	6,968K 100%	115K 1.7%	432K
D: Vortex breakdown 45x45x55	111K 100%	290 0.3%	3K

Table 2: Selected cells

Using more than one hypernode deteriorates the performance severely, because the memory latency of the hypernode interconnect is approximately four times higher. Besides the memory allocation, the memory access introduces an additional slow-down. Due to a missing memory distribution strategy in our implementation, the systems default round robin strategy is used. A theoretically possible - yet not developed - explicit distribution would presumably improve the observed situation.

#threads	1/[s]	2/[s]	4/[s]	8/[s]	16/[s]
octree code	140.77 100%	71.68 98.2%	35.98 97.8%	18.42 95.5%	11.84 74.3%
memory allocation	17.96 100%	14.45 62.1%	10.30 43.6%	4.63 48.5%	56.13 2.0%
memory access	195.44 100%	100.65 97.1%	50.67 96.4%	26.46 92.3%	17.21 71%
total on Convex	354.42 100%	186.78 94.8%	96.95 91. 4%	49.51 89.5%	85.18 26%
octree code	0.32 -	4.08 100%	1.12 182.1%	2.09 48.8%	1.67 30.5%
memory allocation	39.00 -[a]	63.89 100%	51.58 61.9%	40.06 39.9%	31.47 25.4%
memory access	0.88 -	2.17 100%	0.23 471.7%	0.42 129.2%	- -
total on SGI	40.20 -	70.41 100%	52.93 66.5%	42.57 41.3%	33.14 26.6%

Table 3: Convex and SGI profiling of octree construction: wall clock and parallel efficiency

[a]No malloc-locking required. We use the two-thread measurements for normalization.

On the SGI Challenge, we can see a similar picture as in the one-hypernode measurements on the Convex. Memory allocation is limiting the scalability. Approxi-

[1]Non-Uniform-Memory-Access

[2]Note that the profiling process increases execution times. Therefore, only the measured efficiencies of the different experiments can be compared.

[3]Memory allocation is due to the storing of the generated triangles and vertices in FIFO-queues.

mately 80% of the allocation time is spend in the locking mechanism of the operating system. Additionally, the floating point operations of the isosurface extraction is introducing additional traffic on the system bus and therefore, establishs a potential overhead for this phase.

#threads	1/[s]	2/[s]	4/[s]	8/[s]	16/[s]
Marching Cubes	95.92 100%	51.43 93.3%	26.25 91.3%	13.86 85.5%	7.52 79.7%
memory allocation	13.82 100%	8.26 88.5%	6.52 56.1%	7.48 24.4%	17.08 5.3%
memory access	195.27 100%	100.31 97.3%	50.56 96.5%	26.23 93.0%	18.79 64.9%
total on Convex	305.01 100%	160.00 95.3%	83.33 91.5%	48.07 79.3%	43.39 43.9%
Marching Cubes	65.96 100%	46.63 70.7%	23.72 69.5%	22.06 37.4%	8.15 50.6%
memory allocation	13.78 100%	8.34 82.6%	4.14 83.2%	2.93 58.8%	3.15 27.3%
memory access	44.85 100%	26.29 85.3%	10.48 107%	9.07 61.8%	3.89 72.1%
total on SGI	146.40 100%	73.16 100%	56.76 64.5%	37.83 48.4%	19.07 48.0%

Table 4: Convex and SGI profiling of isosurface extraction: wall clock and parallel efficiency

There are several pros and cons of shared-memory architectures. We firmly believe that an important aspect is the easy straight-forward parallelization using threads without an architecture-dependent memory distribution strategy. However, a trade-off between the costly development of such strategies and the high costs for memory access of virtually-shared-memory must be considered.

Comparing the measurements on our two memory architectures, we made the observation that the one-hypernode Convex produces a better scaled scheme than the SGI Challenge. We suppose this is due to the better applicability of the crossbar of the Convex than of the system bus of the Challenge during data access.

Depending on the dataset size, the costs of the octree construction and computation of the workload distribution for the rendering varies between less than 58% (dataset C) and less than 41% (dataset D) using only one processor. Considering the obtained cell reduction rate of approximately 95% due to the octree, this seems to be a small price[1]. Nevertheless, the overhead of the octree construction has a considerable impact on the scalability with small datasets (i.e. dataset D).

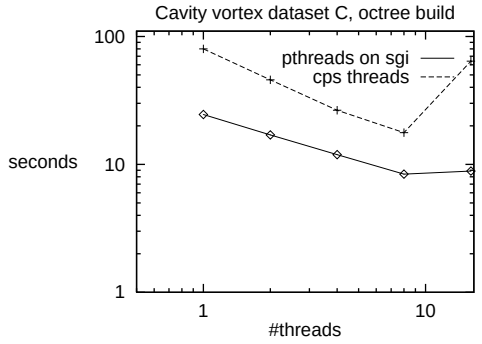

Fig. 5: Build measurements

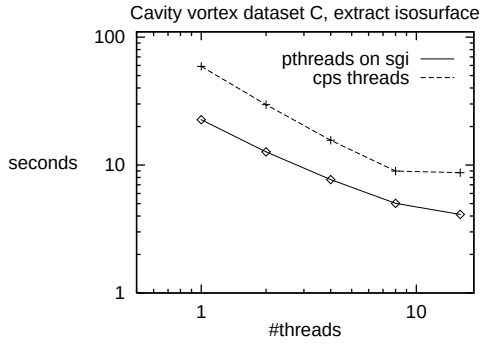

Fig. 6: Render measurements

5 CONCLUSION AND FUTURE WORK

We presented a method for an asynchronous, balanced parallel construction of recursive tree structures, in our case octrees. In addition, we implemented a parallel version of the Marching Cubes algorithms, using the cells selected by the octree. Both schemes are based on the shared-memory paradigm and are implemented using different thread models and memory architectures.

We were able to obtain a good speed-up within one hypernode on the Convex. Using more processors led to rapidly increasing communication costs on the Convex. An explicit memory distribution strategy would improve this situation, due to the reduced communication overhead. However, this strategy would also violate our concept of a virtual-shared-memory

[1]Running a Marching Cubes process without the preselection of contributing cells results in multiple execution times of this phase. However, only the classification of the cells is performed, which approximately is 50% of the total execution time.

architecture, as a foundation for easy and fast parallelization.

On the SGI Challenge, memory allocation and system bus were limiting the speed-up. The succeeding Onyx2-architecture uses a crossbar as interconnect. Comparing our results with results on that architecture update would produce meaningful comparison of the pros and cons of these interconnect technologies.

Overall, the octree construction time is small compared with the total extraction time of the datasets. In comparison to a non-hierarchical, straightforward parallelization of the volumetric datasets this seems to be a small price for an approximately 95% reduction rate of cells (Table 2).

In the presented work, the use of the octree is limited to the search for contributing cells. However, this is not the most beneficial application for octrees in 3D-rendering. Therefore, a future focus will be on direct volume rendering techniques, such as ray casting, and on visibility queries in large dynamic environments.

A drawback of the current implementation is the limitation to cartesian grids, while most datasets in CFD are based on curvilinear grids. The octree data structure only depends on a rectilinear topology. Consequently, an extension for the processing of curvilinear grids is another future focus.

Acknowledgements

The abdominal patient dataset is courtesy of the Visualization Laboratory of the State University of New York at Stony Brook. The cavity dataset is courtesy of the Institute of Fluid Mechanics of the University of Erlangen-Nürnberg.

We like to thank Martin Steckermeier, Matthias Gente, and Michael Schröder for their support using the Convex at the Computing Center at Erlangen. Additionally, we like to thank Rüdiger Westermann and Philipp Slusallek of the Computer Graphics Group at Erlangen for useful discussions and support using the local computing environment. Last but not least, we thank Arie Kaufman and Pat Tonra for support using the SGI Challenge at Stony Brook, and our reviewers for their helpful comments.

This work has been partially supported by DFG project SFB 182 and the MedWis program of the German Federal Ministry for Education, Science, Research and Technology.

References

[Conve95] Convex Computer Corporation: *Exemplar Programming Guide*^{2}, 1995.

[Green93] Greene, N., Kass, M., Miller, G.: *Hierarchical Z-Buffer Visibility*, in *Proc. SIGGRAPH'93*, pp.231-238, 1993.

[Gross95] Grosso, R., Ertl, Th., Klier, R.: *A Load-Balancing Scheme for Parallelizating Hierarchical Splatting on a MPP-System with Non-Uniform Memory Access Architecture*, in *Proc. HPCGV'95*, pp.125-134, 1995.

[Hanra93] Hanrahan, P., Salzman, D., Aupperle, L.: *A Rapid Hierarchical Radiosity Algorithm*, in *Proc. SIGGRAPH'93*, pp.197-206, 1993.

[Konin96] Koning, A., Zuiderveld, K., Viergever, M.: *Volume Visualization on Shared-Memory Architectures*, in *Proc. First Eurographics Workshop on Parallel Graphics and Visualization*, pp.129-143, 1996.

[Laur91] Laur, D., Hanrahan, P.: *Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering*, in *Proc. SIGGRAPH'91*, pp.285-288, 1991.

[Loren87] Lorenson, W. E., Cline, H. E.: *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, in *Proc. SIGGRAPH'87*, pp.163-169, 1987.

[Nieh92] Nieh, J., Levoy, M.: *Volume Rendering on Scalable Shared-Memory MIMD Architectures*, in *Proc. Workshop on Volume Visualization'92*, pp.17-24, 1992.

[Samet94] Samet, H.: *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, 1994.

[Shekk96] Shekkar, R., Fayyad, W., Yagel, R., Fredrick, J.: *Octree-Based Decimation of Marching Cubes Surface*, in *Proc. IEEE Visualization'96*, pp.287-294, 1996.

[Silic94] Silicon Graphics Inc.: *Power Challenge Technical Report*, Silicon Graphics Inc., Mountain View, 1994.

[Singh94] Singh, J.P., Gupta, A., Levoy, M.: *Parallel Visualization Algorithms: Performance and Architectural Implications*, in *IEEE Computer* Vol 27, No. 7, pp.45-55, July 1994.

[Wilhe92] Wilhelms, J., van Geldern, A.: Octrees for Faster Isosurface Generation, in ACM Transactions on Graphics, pp.201-227, July 1992.

Processors/Dataset	1/[s]	2/[s]	4/[s]	8/[s]	16/[s]
A:octree	170.50	100.42	56.40	35.20	39.87
	100%	84.9%	75.6%	58.9%	26.8%
isosurface[a]	156.78	87.144	60.72	33.52	24.85
	100%	90%	64.6%	58.5%	39.4%
B:octree	25.58	17.97	12.88	9.51	8.51
	100%	71.2%	49.6%	33.6%	18.8%
isosurface	29.07	18.34	11.71	7.57	5.37
	100%	79.2%	62.1%	48%	33.8%
C:octree	24.53	17.01	11.91	8.38	8.87
	100%	72.1%	51.5%	36.6%	17.3%
isosurface:	22.59	12.67	7.70	5.02	4.11
	100%	89.1%	73.3%	56.2%	34.3%
D:octree	0.40	0.26	0.19	0.32	0.44
	100%	57.7%	50.9%	15.3%	5.7%
isosurface	0.49	0.47	0.61	1.05	2.11
	100%	51.9%	20.2%	5.8%	1.5%

Table 5: SGI Challenge/Pthreads: wall clock and parallel efficiency

[a]Measurements are inclusive memory allocation.

Processors/Dataset	1/[s]	2/[s]	4/[s]	8/[s]	16/[s]
A:octree	491.93	339.16	201.47	135.67	642.71
	100%	87.2%	73.4%	54.5%	5.8%
isosurface[a]	261.14	131.82	66.11	33.89	17.63
	100%	99.1%	98.8%	96.3%	92.6%
B:octree	81.08	45.92	26.92	18.00	79.86
	100%	88.3%	75.3%	56.3%	7.8%
isosurface	76.17	41.29	24.59	19.25	16.44
	100%	92.2%	77.5%	49.5%	29%
C:octree	79.88	45.77	26.47	17.69	64.12
	100%	87.3%	75.4%	56.5%	7.8%
isosurface:	59.99	29.69	15.64	8.98	8.72
	100%	101%	95.9%	83.5%	43%
D:octree	1.39	0.82	0.57	0.65	2.29
	100%	84.8%	61.4%	26.7%	3.8%
isosurface	2.34	1.64	2.01	4.96	8.45
	100%	71.4%	29.1%	5.9%	1.7%

Table 6: Convex SPP/CPS threads: wall clock and parallel efficiency

[a]Due to insufficient memory, no FIFO-queue is used. Therefore, no memory allocation is performed.

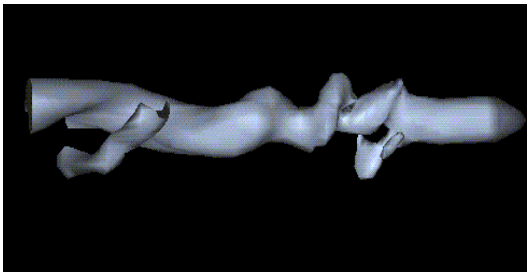

Fig. 7: Dataset D - Vortex breakdown of an injected fluid at time frame 300

Fig. 8: Dataset D - Vortex breakdown of an injected fluid at time frame 360.

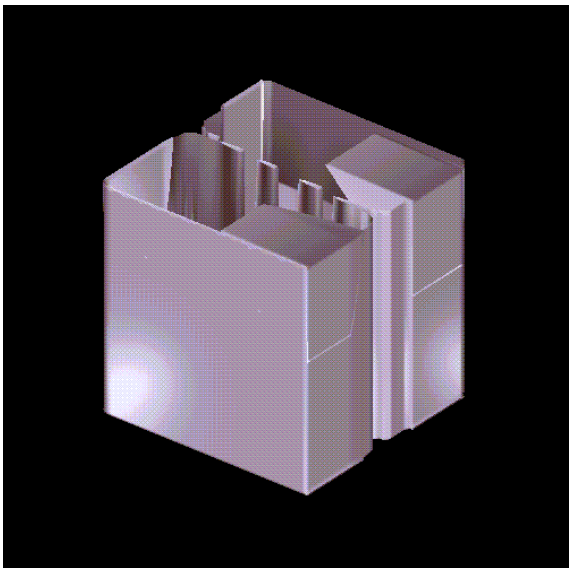

Fig. 9: Dataset C - Cavity vortex:
Particular z-components of velocity vector field of a fluid within a cavity. Two sides of the cavity are heated differently.