Eurographics Symposium on Parallel Graphics and Visualization (2007)
Jean M. Favre, Luis Paulo dos Santos, and Dirk Reiners (Editors)

Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling

Luciano Soares, Clément Ménier, Bruno Raffin, and Jean-Louis Roch.

INRIA, Laboratoire d'Informatique de Grenoble - LIG, Grenoble, France

Abstract

This paper introduces a dynamic work balancing algorithm, based on work stealing, for time-constrained parallel octree carving. The performance of the algorithm is proved and confirmed by experimental results where the algorithm is applied to a real-time 3D modeling from multiple video streams. Compared to classical work stealing, the proposed algorithm enforces a relaxed width first octree carving that enables to stop computations at anytime while ensuring a balanced carving.

Categories and Subject Descriptors (according to ACM CCS): C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors) I.4.5 [Image Processing and Computer Vision]: Reconstruction

1. Introduction

Mastering parallelism is a major challenge when developing computationally intensive interactive applications, but it can enable to reach the targeted low latencies and high refresh rates.

This article presents a dynamic load balancing algorithm, based on work stealing, that enables to stop computations at any time. This algorithm is applied to 3D modeling. 3D Modeling consists in building a 3D model of people or objects being filmed by a set of calibrated cameras. This 3D model must be computed in real time from the different video streams before it is injected into the virtual world to enable interactions [MP04, GWN*03]. Many different algorithms exist for 3D modeling. A classical approach, the one our experimental results rely on, is to "carve" an octree [Sze93]. 3D modeling has been selected as it is an interesting case study where interactivity is critical and the amount of computations to perform can be significant.

Parallelizing such octree carving algorithm raises two main isssues:

- load-balancing: the shape of the octree is irregular and depends on the input data. Thus a static load balancing scheme fails to guarantee an efficient use of the available computing resources. An efficient dynamic load balancing is required.
- time-constraint: when the timeout occurs we expect the octree to be balanced, i.e. that computing resources coordinate their efforts to avoid having a branch deeply explored while an other is seldom tested. It requires extra synchronizations between processors to enforce a relaxed width first octree carving.

To achieve this goal we propose a modified work-stealing technique. The efficiency of this approach is validated formally and experimentally.

Our parallel algorithm dynamically balances the processor work load and ensures it can be stopped at anytime while guaranteeing a balanced space carving. The implementation on a 16 cores architecture (8 dual-core processors) speeds up the computations up to 14.4 times in comparison to the sequential execution. We also present early experiences using one GPU as a co-processor. The performance increases by 30% compared to using only one CPU, and fades as the number CPUs involved increases.

The paper is organized as follow. Section 2 presents the octree based algorithm for 3D Modeling. Work stealing and the associated theoretical results are detailed in Section 3. The parallel octree algorithm is presented in Section 4, its proof in Section 5, its implementation in Section 6 and the experimental results in Section 7. Section 8 discusses the GPU based tests before conclusions are draw in Section 9.

2. Octree Based Voxel Carving

We present in the following the sequential octree based voxel carving algorithm. The algorithm takes as input data video streams from a network of cameras (Fig. 1). To ensure a high quality modeling, cameras must be properly calibrated (brightness, color and position) and synchronized. From each image a 2D silhouette is extracted by background substraction. Various methods exist [HHD99]. we rely on [KGYS05]. Pixels outside the silhouettes are set to white, while the others are set to black (Fig. 2). The octree algorithm is executed on each set of silhouette images taken at the same time. Starting from one initial voxel corresponding to the acquisition space, the algorithm probes each voxel to compute if it lies outside or inside the visual hull, i.e. if the voxel projects at least outside one silhouette or inside all silhouettes. Uncertain voxels (intersecting a silhouette contour) are split in 8 smaller voxels. A stopping condition can be set to bound the execution time, based on a timer or a maximum depth level for instance.

Figure 1: *6 Cameras filming a user.*

Figure 2: *Silhouettes from 3 cameras.*

There are several approaches to test if a voxel is full (inside the visual hull) or not. We are using an exhaustive method where a voxel is subdivided into a regular 3D grid. The algorithm iterates on each grid vertex, projecting it into the silhouette. Computations for this voxel are stopped as soon as the voxel is known to be full, empty or uncertain. The number of grid vertices contained in a voxel is proportional to its size.

This algorithm is interesting on different aspects. First, as all octree based approaches, it enables to reduce the amount of computations, compared to a space partitioning with a fixed voxel size. Second, it provides a volumetric model in the form of a list of cubes, making it easy for computing collisions with other objects. Finally, it easily enables to control the amount of time allotted to carving by having the stop condition waiting for a timeout or an external event. In this case it is important to underline that the tree carving should be performed width first rather than depth first. If performed depth first, the octree will be very detailed in some areas, while seldom tested in others (Fig. 3). For a sequential execution, the only consequence of a width first carving is an extra memory consumption.

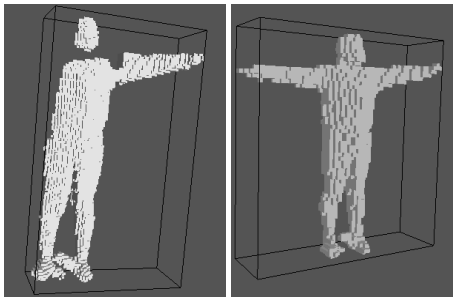

Figure 3: *A depth first(left) and width first(right) octree carving with the same elapsed time. The former leads to an unbalanced space carving, the latter to a balanced one.*

3. Work Stealing

We recall the principle of work stealing, the main associated results and analyze why work stealing is well adapted for octree carving.

Work stealing is a classical approach for dynamic load balancing. It has been used for various computations, including parallel graphics [HA98, CD99]. It extends the Graham list scheduling principle [Gra66] for programs that create tasks recursively. The principle is simple. When starting the execution, a first processor is assigned all source tasks (the initial ready tasks). At runtime, each processor maintains a local list where it stores the ready tasks it has locally created. A task becomes ready when all its predecessor tasks, i.e. the tasks it depends on, have been executed. The tasks are organized in the list according to a total sequential depth first order. When a processor P completes a task, it pops the first one t (according to depth first order) in its local ready list if non empty. Else, P is idle and becomes a thief: it randomly selects another processor until finding one victim processor V that owns ready tasks. Then it picks-up the oldest ready task t in the ready list of V. In both cases, P starts the execution of t.

Work stealing achieves a provable performance with respect to the *work* and *depth* of the parallel algorithm. The work W_1 is the total number of elementary operations performed during the execution of the algorithm. An instruction

may be a standard operation or a task creation. The depth W_∞ is the critical path in number of operations for an execution on an unbounded number of processors, i.e. the number of instructions along the longest dependency path. Let T_p be the execution time on p identical processors with execution speed Π (in number of instructions per time unit). An execution takes a time $T_1 = \frac{W_1}{p\Pi}$ on a single processor and a time $T_\infty = \frac{W_\infty}{\Pi}$ on an unbounded number of processors. On p processors, work stealing ensures that with a high probability [ABP01]:

$$T_p \leq \frac{W_1}{p\Pi} + O\left(\frac{W_\infty}{\Pi}\right) \quad (1)$$

and the number of steals is small, $O(W_\infty)$ per processor.

Thus, if the depth W_∞ is small compared to the total amount of work W_1 the parallel execution time is close to the lower bound $\frac{W_1}{p\Pi}$. This motivates the use of work stealing to schedule parallel programs having a small depth W_∞.

The octree shows properties making it very well adapted to work stealing. We consider a task the computation associated with one voxel. The dependency graph corresponds to the octree graph, as a given voxel can be computed as soon as its parent has been treated. In the worst case where no pruning occurs, a tree of depth n leads to $W_1 = \frac{8^{n+1}-1}{7}$ tasks, while the critical path is $W_\infty = O(n)$. Thus, having $W_1 \gg W_\infty$, work stealing should lead to optimal parallel executions. Even when pruning occurs, the ratio is usually very favorable to work stealing. For instance, our test data set, consisting of a full human body (Fig. 4), has $W_1 = 162799$ voxels when going up to depth level $W_\infty = 8$. The following properties also contribute to keep the overhead of work stealing small:

- All tasks perform the same computation. The only difference between two tasks is the coordinates and size of the considered voxel. Thus stealing work only consists in stealing a list of a voxel coordinates and size. It leads to light memory transfers.
- Task dependencies are simple: a task only depends on its parent. When a processor splits a voxel, it can directly add its 8 children voxels to its ready list. There is no other dependency to be solved that could require to wait for another processor.

Close to the octree root, the amount of parallelism is reduced (1 voxel at the root, 8 at depth 1, 64 at depth 2), and the cost of projecting a voxel into the silhouettes is higher (proportional to the voxel size). This can impair the performance of work stealing. To soften this effect, usually octree carving starts at a higher depth level (level 2 for our tests).

4. Adaptive Octree Algorithm

The goal of this work stealing algorithm is to dynamically schedule the work load to better use the processors available. We assume the target parallel machine supports a global address space with a shared (or virtually shared) memory access and manages data locality.

Each voxel is represented as a quadruple of its coordinates and level (i, j, k, d). To manage the workload the voxels are organized in ready lists. Each ready list consists of a vector of voxels and pointers to the first, last and current voxel. We call a task, the computations required to test a voxel status.

4.1. Initialization

The algorithm starts at depth level n with 8^n initial voxels. The number n is usually the smallest number to have more initial voxels than processors. These voxels are split into p ready lists, where p is the number of processors. The goal is to avoid the performance bottleneck of the first depth levels that do not provide enough parallelism.

The ready lists are organized in a singly-circularly-linked list. There is one list of ready lists per depth level, called a level list. All level lists, except the ones from the starting depth level, are initially empty. Each processor is assigned a first ready list from the starting level.

One processor has a manager role. It takes care of initializing the first ready lists and signal all other processors when computations must be stopped (because the timeout occurred).

4.2. Octree Level Computation Based on Work Stealing

Each processor tests the voxels of its ready list. If a voxel needs to be split, its 8 child are inserted into the ready list of the next level.

Each processor cycles twice through the level list. A processor scans the level list from a randomly chosen position, looking for a free ready list in the level list until it completes one cycle. If it finds one, it takes it. Else, since there is no more ready list to grab, it performs a second loop, but this time the processor becomes a thief. It traverses the ready lists trying to get part of the remaining voxels. For a target ready list, the thief processor locks the current working pointer of the victim processor (the owner of the list). It grabs half of the remaining voxels, leaving a minimum number of voxels (fixed by a threshold) to avoid voxels to be stolen back and forth. The thief creates a new ready list containing these voxels. This operation just involves pointer settings and does not lead to voxel copies. The working pointer of the victim is unlocked as soon as it can safely restart processing its ready list.

Finally, when a processor ends its second cycle through the current level list, it starts working on the next level, processing the voxels of its ready list if not empty.

4.3. Overlapping Level Synchronization

Let $L_i(t)$ be the level of the voxel being computed by processor i at time t, and $\sigma = \max_t \max_{i \neq j} |L_i(t) - L_j(t)|$ be the maximal level synchronization, i.e. the largest distance in term of levels between two processors. To enforce a (relaxed) width first like octree carving it is required that σ is always kept small. This is achieved by introducing extra synchronizations. Setting a global barrier after each level guarantees a synchronization $\sigma = 0$ at any time, but it prevents level overlapping and then restricts parallelism. A synchronization $\sigma \leq 1$ enables to overlap synchronization overhead while limiting the octree unbalance. To ensure $\sigma \leq 1$, we implemented it as follows. To each level d corresponds a shared counter $C[d]$ initialized to the number of processors p. When a processor completes all its ready tasks at level d and if $C[d-1] = 0$, it starts stealing. If it does not succeed to get voxels from other processors, then it decreases $C[d]$ by 1 and starts the computation of its local voxels at level $d+1$. Once completed, it waits until $C[d-1] = 0$ before starting new steal requests.

4.4. Time Control

A time control routine was integrated in the algorithm to bound the time spent to carve the octree. The main objective is to enforce a real-time behaviour for the final application, i.e. a minimum latency and maximum refresh rate.

The manager processor is in charge of checking the time elapsed and signal other processors that the time is over.

One difficulty is to define the time check frequency, to limit the overhead while being frequent enough to enable a good time control. A good choice is to have the manager control the time elapsed each time it completes a ready list. Because synchronizations are present into the code to enforce a width first tree traversal, it enables to stop the algorithm at any time keeping a well balanced space carving.

5. Provable Performance of the Adaptive Octree Algorithm

This section deals with the proof of the performance of the adaptive octree algorithm. It is analyzed with respect to the reference sequential algorithm.

For the sake of simplicity, we consider that the adaptive algorithm implements a maximal level synchronization $\sigma = 0$, with a synchronization barrier after each octree level. This prevents level overlapping: level d is computed only when level $d-1$ is completed, like in the sequential computation. Then parallelism occurs only at each level. Furthermore, we will consider that the sequential computation starts with the same voxels as the parallel algorithm. So if we consider p processors, we consider the sequential computation to be initialized at depth $\lceil \log_8 p \rceil$ with $8^{\lceil \log_8 p \rceil}$ voxels.

Under those assumptions, when there is no time limit, the following theorem states that the adaptive algorithm is almost p times faster than the sequential one if the depth of the octree is small w.r.t. its number of voxels.

Theorem Let T_s be the time of the reference sequential algorithm to compute an octree with n nodes and depth d on a processor with speed Π. The adaptive algorithm running on p identical processors with speed Π computes the same octree in time:

$$T_p =_{n \to \infty} \frac{T_s}{p} + O\left(\frac{d \log n}{\Pi}\right) \tag{2}$$

Proof. The state of each voxel being deterministically computed, both algorithms compute the same octree. Let $T_s(i)$ (resp. $T_p(i)$) be the time of the reference sequential algorithm (resp. adaptive algorithm) to compute level i, $1 \leq d \leq i$, and n_i be its number of nodes. At each successful steal, a processor steals half the ready voxels of a non idle processor. Then, on an infinite number of processors, the parallel algorithm has critical depth $W_\infty = \log n_i$. The operations performed by the parallel algorithm are either voxels computations, i.e. $T_s(i).\Pi$ unit operations, or overhead instructions to manage parallelism (locks, steals, list management). Due to work stealing, the number of steal requests is $O(W_\infty)$ per processor, i.e. $O(\log n_i)$. Except steals, the only other overhead operations are when a processor access its own local ready list to extract voxels. This requires a lock to avoid contention with possible thieves. However, if there are v voxels in the ready list, then $\log v$ voxels are extracted at the price of only one lock. Then, following [RTB06], the number of lock operations is $O\left(\frac{n_i}{\log n_i}\right)$. Then, from the work stealing fundamental theorem (Section 1), we have $T_p(i) \leq \frac{T_s(i)}{p} + \frac{1}{\Pi} O\left(\frac{n_i}{p.\log n_i} + \log n_i\right) =_{n_i \to \infty} \frac{T_s(i)}{p}$. Summing for all the levels concludes the proof. △

However, due to real time interactive constraints, the depth of the octree is truncated at a given unknown time limit. The computation time of the algorithm is fixed and the objective is to maximize the level of details, i.e. the number of voxels computed. Indeed, taking benefit of parallelism, the adaptive octree algorithm not only computes faster but also more details. The next theorem states that in a fixed time t, the adaptive algorithm on p processors computes almost the same precision as the reference sequential algorithm in a time $p.t$.

Theorem

Let n_p be the number of voxels computed by the adaptive algorithm in a time limit t on p identical processors. Let n_s be the number of voxels computed by the sequential reference algorithm in a time limit $p.t$ on 1 processor. Let d_p (resp. d_s) be the last fully completed level of the adaptive (resp. sequential) algorithm.

Then:

$$n_p = n_s - O(\log n_s) \text{ and } |d_p - d_s| \leq 1 \quad (3)$$

Proof. The proof is also based on the work stealing theorem, applied to each level. Let d be the maximal level of a voxel computed by the adaptive algorithm. Since this algorithm performs a barrier after each level, clearly $d \leq d_p + 1$. From previous theorem, in a time t, the adaptive algorithm performs $W_p = p.t.\Pi$ operations among which at most $O\left(d \log n_p + \frac{n_p}{\log n_p}\right)$ are overhead instructions with respect to voxels computation. Then, if $T_s(n_p)$ denotes the sequential time to compute the corresponding voxels, $p.t.\Pi = T_s(n_p) + O\left(d \log n_p + \frac{n_p}{\log n_p}\right)$. Then, asymptotically for n_p large enough w.r.t. d, $p.t.\pi \simeq T_s(n_p)$. Moreover, all those n_p nodes are at most on two levels, d_p and $d_p + 1$. Then, since $p.t.\Pi = T_s(n_s)$, $n_p \simeq n_s$ and, due to barrier, $d_s = d_p$ or $d_s = d_p + 1$. △

Asymptotically for a large number of nodes w.r.t. the depth of the octree, both theorems generalize to the practical case where $\sigma = 1$ (then at most two levels may differ between the sequential and the adaptive algorithm).

6. Implementation

The algorithm was implemented in C++ using Posix Threads. As stated earlier, we target parallel computers supporting a global address space with shared (or virtually shared) memory. In this context Posix threads provide a well adapted programming environment.

For a better performance the use of mutex like semaphores was eliminated. Instead assembly atomic operations like compare_and_swap "cmpxchg" and atomic_add_return "xadd" combined with the LOCK prefix were used. These atomic operations are supported by most modern CPUs. We noticed a performance increase of about 20% compared to a mutex based implementation.

The "yield" instruction("sched_yield" systems call) was used to better manage waiting times. It improves the performance in the waiting loops informing the kernel to schedule other processes.

The application is launched with one thread per processor. The first thread is the manager. We consider we are the only users of the computer and no other application is running. To prevent the migration of threads during execution, which would impact performance, each thread was locked on a given processor. For that purpose, we used the "pthread_setaffinity_np" instruction. This technique also improves the frame memory control, avoiding cache misses and sparse memory allocations.

To better balance the work load into the initial ready lists, the voxels are distributed in a round-robin fashion. The goal is to give each working list voxels from different space regions.

To reduce contention, a thread does not wait to steal from a locked ready list. If one thief fails to lock a working list, it does not try a second time. It just steps to the next ready list in the chain. It avoids waiting for a lock release.

We relied on the GCC compiler to make an efficient use of SIMD parallel instructions available on the processor. A careful manual code optimization could probably further improve performance.

7. Results

The computer used for the tests has 8 dual Core AMD™2.2GHz Opteron processors, 32 GB of memory, and is running Linux kernel 2.6.17. This is a CC-NUMA (Cache coherency Non Uniform Memory Access) architecture with a virtually shared memory using the HyperTransport™inter-processors communication layer.

7.1. Off-line Cameras

Tests were first performed with two off line series of images. The first one is a sequence of a full body person filmed with 8 cameras, called the Ben benchmark, freely available at `https://charibdis.inrialpes.fr/`. Each camera image has a resolution of 780x582 pixels. The second benchmark, called Al Capone, is a synthetic 3D model, from which we computed 64 images (resolution of 300x300 pixels) from different view points. This model enables to test our algorithm with a very large number of silhouettes. Though today marker-less motion capture environments have usually less than 64 cameras, the trend is to increase this number as it improves the quality of the obtained 3D model. Notice that both image sets fit in the 1MB L2 cache available per core. Ben requires 456KB and Al Capone 713KB.

We first compared a pure sequential implementation of the octree carving algorithm with our parallel code launched with only one thread. The overhead due to the extra code introduced into the algorithm for work stealing is small. It is about 4% for the Ben model and below 1.3% for Al Capone.

We ran the algorithm for both benchmarks with varying numbers of CPUs, without time limit but with various max depth levels. All results are averages over 100 runs. The execution times include the time to load the images from the local disk. We plot (Fig. 6 and Fig. 7) the execution times (logarithmic scale on the y-axis) and the speed up ($s = \frac{T_1}{T_n}$). The gain of using 16 CPUs is very significant with an efficient use of the resources (high speed-ups). For instance Ben at max depth level 8 is computed on 1 processor in about 234.2 ms. The same model takes only 16.82 ms on 16 processors.

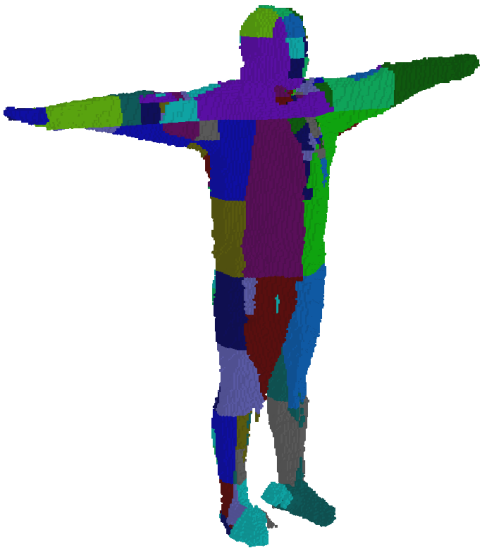

Figure 4: *Ben. Max depth level set to 8.*

Figure 5: *Al Capone. Depth level set to 5 (left) and 7 (right).*

At max depth level 7, the Al Capone goes from about 441.1 ms with 1 CPU to about 31.15 ms with 16 CPUs. Notice that the reconstruction at level 5 does not scale well, since at this low level the execution time is dominated by the image loading – sequential – step. As the amount of parallelism increases while going deeper in the tree, the speed-up increases with the max depth level. Al Capone was also tested with work stealing turned off (Fig. 7). The performance is significantly affected. A static load balancing is inefficient as the shape of the octree, and thus the work load, cannot be predicted.

The number of steals is low in comparison to the amount of cells as predicted by the theory. Table 1 presents the average of all voxels computed against the number of full voxels, and the relative number of steals. About 60% of steal attempts are successful. A side effect from this low number of steals is the good space locality of voxel distribution. By associating a color per CPU, we notice that large contiguous area are processed by the same thread (Fig. 4 and Fig. 5).

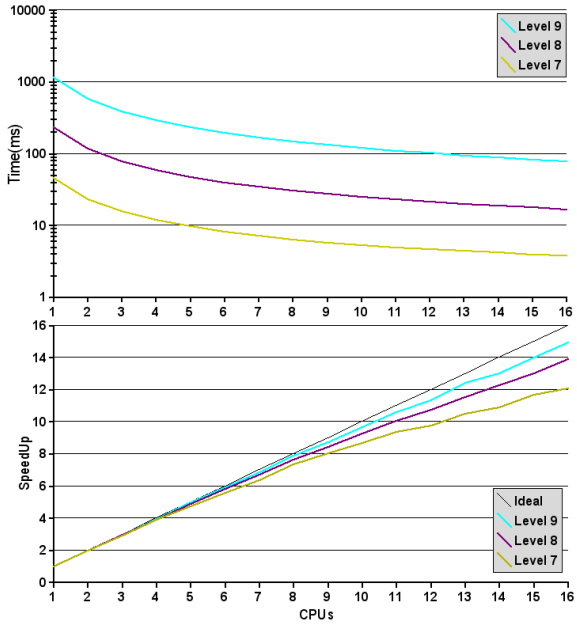

Figure 6: *Execution time and speed up for Ben.*

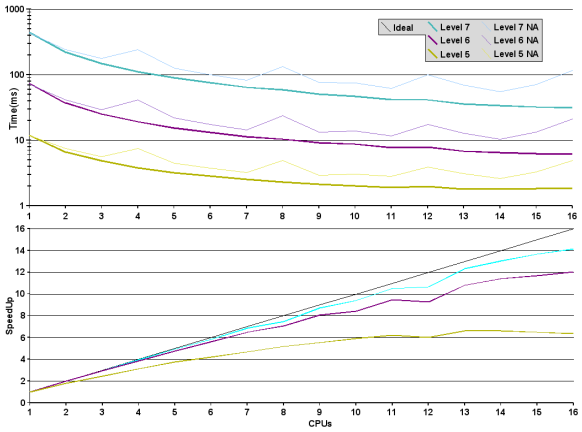

Figure 7: *Execution time and speed up for Al Capone, with work stealing enabled or disabled (NA for Non-Adaptive)*

We tested the time control routine with Ben (Fig. 8) and a 30 ms deadline. The simulation is set to go up to depth level 8. With just one processor it is not even possible to complete level 5, making the model unrecognizable. The execution time is significantly larger than 30 ms, because the processor does not check the elapsed time before it completes the first ready list. Up to 8 processors, the time control is effective: the execution stops before all voxels of level 8 are computed. Notice that the measured execution time is usually slightly higher than 30 ms because after the timeout occurs all processors apply a fast test algorithm to guess

		Voxels		Steals/threads	
Dataset	Level	Computed	Full	Tries	Success
Ben	8	162799	67398	42.59	25.26
Al Capone	7	44840	34601	26.21	16.18

Table 1: *For each data set computed up to a given max level, the number of voxels computed is given with the number of voxels identified as full, the number of steal attempts and successful steals per thread.*

if each pending voxel is full or empty. With 8 processors, the 30 ms limit enables to reach the max depth level. Next, as the number of processors further increase, the extra computing resources available enable to decrease the execution time, ending below 20 ms (the number of voxels computed at level 8 stops to increase).

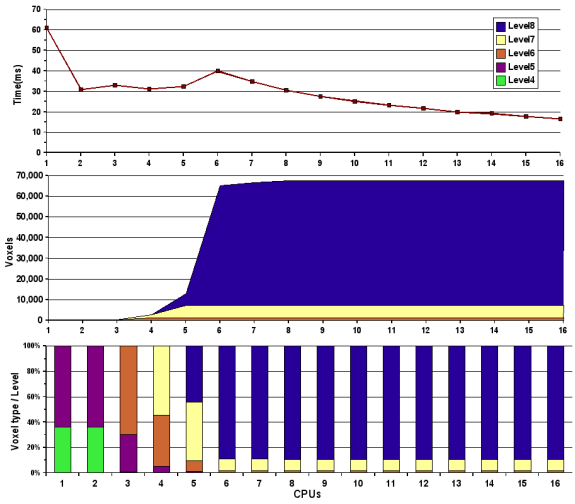

Figure 8: *Ben modeling with a 30 ms time limit. The graph plots the total execution time, the middle graph plots the amount of voxels produced per level, and the lower graph the percentage of voxels types.*

7.2. On-line Cameras

We tested the algorithm in a live environment with 5 FireWire cameras (image resolution 780x580) filming a person in real time. Cameras are genlocked through a specific network. Each one is connected to one computer (dual xeon), processing the incoming video stream to remove the background and compute the silhouette images. Then, the silhouettes are forwarded to the 16 cores computer. It computes the octree and sends the list of full voxels to 16 dual Opteron computers powering a 16 projectors high resolution display wall. These computers render the voxels. All computers are connected through a gigabit Ethernet network.

This application was developed on top of FlowVR [AR06] for coupling and distributing the different software components. FlowVR Render [AR05] was used for the distributed rendering on the display wall.

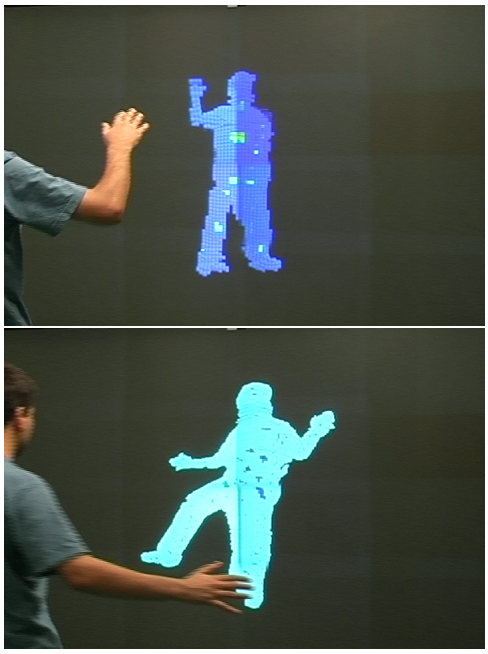

Figure 9: *Live tests with 5 on-line cameras with max depth level 6 (top) and 8 (bottom).*

Refer to the video associated with the article for the results. Notice the resolution of the video is lower than the display wall resolution, making it difficult to distinguish the smaller voxels while they are clearly visible on the display wall. When rendering, the voxels are colored according to their depth level. Tests were performed with and without time control, with various numbers of processors and different levels of max depths. The quality significantly increases with the max depth (Fig. 9). Fingers become visible at level 8. Some artifacts (ghost leg) are visible in some situations. This is due to the accumulation of small errors from camera calibration, background subtraction and voxel projection tests. The time control enables to keep the latency low and the frame rate stable. Some momentaneous performance drops are visible in the video. Though the cause of these drops are not yet clearly identified, it is probably related to network issues (we suspect the linux network driver).

8. Involving the GPU

The implementation was modified to use a GPU as a co-processor for one thread. The work stealing algorithm is not modified. The only difference comes from the way a GPU

processes a voxel. As stated earlier, to test a voxel, different points contained in the voxel are projected back onto the silhouettes. On a CPU, the result of each point projection is probed to detect if the status of the voxel can be defined. If so the CPU skips to the next voxel. Due to the SIMD nature of a GPU, making so many probing tests is highly inefficient. To bypass this limitation, the CPU provides to the GPU a list of points to project back onto the silhouettes. The GPU performs all these projections and the CPU gets back the results to define the status of the voxels. The GPU becomes faster than the CPU (compared to the case where the CPU performs all the projections) only if the number of points to test is large enough to hide the overhead of transferring the data back and forth between the CPU and the GPU. So the use of the GPU is triggered only if the number of tests to perform reaches a certain threshold. To reach that threshold several voxels can be tested at once if available in the ready list.

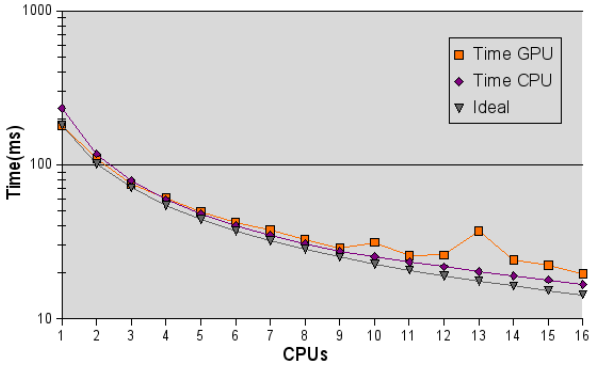

Figure 10: *GPGPU x Pure CPU.*

Experiments were performed on the 16 core machine equipped with one Nvidia Geforce 7900 graphics card (Fig. 10). Involving the GPU instead of relying only on the CPU increases the performance by 30% from 234.20ms to 180.17ms. Using these numbers as the reference sequential execution times ($T_1^{GPU} = 180.17$ms and $T_1^{CPU} = 234.20$ms), we can compute a lower bound for the execution time when $p-1$ CPUs and one CPU/GPU couple are involved in the computation:

$$T_{ideal}(p) = \frac{1}{\frac{(p-1)}{T_1^{CPU}} + \frac{1}{T_1^{GPU}}} \quad (4)$$

Experimental results shows that our implementation usually fails to stick to this ideal case, often a pure CPU based execution being more efficient. The CPU is in fact often faster than the GPU because it can bypass many projection tests while the GPU will always perform all tests even if the voxel status can be defined after just a few tests. The main interest of this early experiment is to show that a GPU can be involved in the computation without having to deeply revisit the work stealing algorithm. Future experiments will focus on involving more GPUs and improving the GPU implementation, targeting a Nvidia G80 GPU programmed with the CUDA library.

Notice that the theoretical results (Section 5) does not apply to computing units running at different speeds. However we should be able to extend our result to this case by relying on Bender and Rabint's proof of work-stealing for heterogeneous processors of different and possibly changing speeds [BR02].

9. Conclusion

This paper introduced a work stealing algorithm for a time-constrained octree carving. The algorithm enables to dynamically balance the work load while ensuring a relaxed width first octree carving, required to get a balanced octree carving when the timeout occurs.

The algorithm was validated theoretically as well as experimentally by applying it to 3D modeling. The algorithm is general enough to be applied to other problems, for instance from computer graphics where octrees are common. It can also be applied to different tree structures. Just notice that the smaller the tree arity, the smaller the ratio $\frac{W_1}{W_\infty}$.

Future work will focus on improving the GPU implementation to efficiently involve multiple CPUs as well as multiple GPUs into the computation.

10. Acknowledgements

The authors wish to thank Thomas Arcila, Everton Hermann and Florian Geffray for their help with the experiments.

This work is partly funded by ANR grant BGPR/SafeScale.

References

[ABP01] ARORA N. S., BLUMOFE R. D., PLAXTON C. G.: Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst. 34*, 2 (2001), 115–144.

[AR05] ALLARD J., RAFFIN B.: A shader-based parallel rendering framework. In *IEEE Visualization Conference* (Minneapolis, USA, October 2005).

[AR06] ALLARD J., RAFFIN B.: Distributed Physical Based Simulations for Large VR Applications. In *IEEE Virtual Reality Conference* (Alexandria, USA, March 2006).

[BR02] BENDER M. A., RABIN M. O.: Online scheduling of parallel programs on heterogeneous systems with applications to cilk. *Theory of Computing Systems Special Issue on SPAA '00 35*, 3 (2002), 289–304.

[CD99] CLYNE J., DENNIS J.: Interactive direct volume rendering of time-varying data. In *Eurographics Data Visualization '99 Conference* (1999), pp. 109–120.

[Gra66] GRAHAM R. L.: Bound for certain multiprocessing anomalies. *Bell System Tech. J.* (1966), 1563–1581.

[GWN*03] GROSS M., WUERMLIN S., NAEF M., LAMBORAY E., SPAGNO C., KUNZ A., KOLLER-MEIER E., SVOBODA T., GOOL L. V., S. LANG K. S., MOERE A. V., STAADT O.: Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In *Proceedings of ACM SIGGRAPH 03* (San Diego, 2003).

[HA98] HEIRICH A., ARVO J.: A competitive analysis of load balancing strategies for parallel ray tracing. *The Journal of Supercomputing 12*, 1–2 (1998), 57–68.

[HHD99] HORPRASERT T., HARWOOD D., DAVIS L. S.: A Statistical Approach for Real-time Robust Background Subtraction and Shadow Detection . In *IEEE ICCV'99 frame-rate workshop* (1999).

[KGYS05] KARAMAN M., GOLDMANN L., YU D., SIKORA T.: Comparison of static background segmentation methods. In *Visual Communications and Image Processing (VCIP '05)* (Beijing, China, July 2005).

[MP04] MATUSIK W., PFISTER H.: 3D TV: A Scalable System for Real-Time Acquisition, Transmission, and Autostereoscopic Display of Dynamic Scenes. In *Proceedings of ACM SIGGRAPH 04* (2004).

[RTB06] ROCH J.-L., TRAORE D., BERNARD J.: On-line adaptive parallel prefix computation. In *EUROPAR'2006* (Dresden, Germany, August 2006), Springer-Verlag L. ., (Ed.), pp. 843–850.

[Sze93] SZELISKI R.: Rapid Octree Construction from Image Sequences. *Computer Vision, Graphics and Image Processing 58*, 1 (1993), 23–32.