Master of Science in Informatics at Grenoble (MoSIG)
Master Mathématiques Informatique – spécialité Informatique
option Graphics, Visions and Robotics

Real-Time Polyhedron Intersection for Multi-Camera 3D Modeling

Nguyen Ngoc Lien Hoa

Research project performed at team Moais, INRIA Grenoble Rhône-Alpes

Under the supervision of
Prof. Bruno Raffin
Prof. Matthijs Douze
Prof. Jean-Sebastien Franco

Defended before a jury composed of
Prof. James Crowley
Prof. Olivier Aycard
Prof. Jean-Claude Fernandez
Prof. François Faure
Prof. Thierry Fraichard

June 2014

Abstract

This project studies the algorithm of general boolean operation on polyhedrons. One application of the algorithm is for 3D model reconstruction from multi cameras. We implement a parallel version of the algorithm to run on multi-core machine using task stealing for a dynamic load balancing scheme. Other aspects of performance optimization such as tree structure, granularity and contention is also considered.

Acknowledgement

I would like to thank my supervisors Bruno Raffin, Matthijs Douze, and Jean-Sebastien Franco for giving me a lot of guidance and mentoring throughout the project. Also my thanks to many others at team Moais and Inria that have supported me during my work. I am greatful for the Persyval-lab for funding my studies and especially professor Ioannis Parissis for his recommendation to follow the master program of MoSIG at Grenoble University. Finally I would like to thank my friends and family for their love and support during my time here.

This work has been partially supported by the LabEx PERSYVAL-Lab
(ANR-11-LABX-0025-01)

Table of Contents

Index of Figures

Index of Tables

1. Introduction

3D Modeling is a topic that has received a lot of attention of research, for its interesting geometry characteristic and vast applications, from manufacturing to virtual reality and telepresence[1]–[3]. The most common method of finding the 3D model of an object is to find its bounding volume by intersecting several viewing cones. As the number of view cones increases, the bounding volume gets closer to its realistic object thus the complexity of the computation also increases. To overcome this, some methods use a tree structure to partition the space thus quickly eliminateing the irrelevant part, called “carving”[4]. Another approach is to take advantage of epipolar geometry to identify the edges at a cheaper computation cost, as discussed by Matusik et al.[5] and Franco et al.[6]

Previously at Inria an efficient polyhedron modeling algorithm (EPVH) has been developed [6] and motivate several experiments on 3D modeling application[1], [2], [7], [3]. These experiments have shown that it is possible to reconstruct 3D subject in real-time. However these experiment were running with a few (less than 8) cameras. The new target for the team is to have an algorithm that can be able to run in realtime with as many as 64 cameras that capture high resolution images. The new challenge is to develop a highly parallel algorithm that can be able to scale for a much more complex datasets.

At the same time at Inria a new algorithm is being developed by Jean-Sebastien Franco, Matthijs Douze and Bruno Raffin for solid modeling. Based on the algorithm of EPVH [6] but rather restricted to view cones intersection, it generalize to any boolean operation on polyhedrons. The accuracy of the algorithm has been verified, however the performance of the current implementation are not sufficient to reach a realtime execution on complex datasets(10 second for a complex synthetic model with 42 cameras). The purpose of this project is to implement a parallel version of the current algorithm that can run at a real-time processing rate. The two main aspect of the project is first to identified the possibility where the algorithm can be parallelize and second, to implement it with dynamic load balancing and study its optimization for performance.

The report is organized as follow. First we discuss the known parallel algorithms for visual hull processing and related appraoches. Next we present the mkCSG algorithm. The third section deals with our main contribution, the parallel version of mkCSG. We then present the results and the conclusions.

2. Related Works

Since the first concept of geometrical modeling from 2D image silhouette introduced by Baumgart [8], the other following studies on a more efficient algorithm has been heavily research. A notable volume based approach was presented by Szeliski[4] . Using a tree of recursively subdivided cubes, Szeliski can represent the bounding volume of the object efficiently. This data structure facilitates efficient volume carving and was applied to model simple objects (cone, blocks, sphere) on a turning table. Matusik et al. take a different approach to the find the bounding volume, i.e surface based, using intersection of viewing ray on the image planes of the silhouettes to find the intersection segments of all viewing cones [5] This algorithm reduce the complexity of finding intersection in 3D space to finding intersection in 2D image space. Current state of the arts EPVH algorithm by J-S. Franco and E. Boyer[6] develop on Matusik's method to finding viewing edges but then add explorations of the connectivities between edges for an exact polyhedral visual hull.

With the developing power of multi-core machines, other researchs have been focusing on how to exploiting the parallel processing of the multiple processor available to get better performance. Previous study on task-stealing prove the efficiency of dynamic load balancing for the parallel construcion of octree in 3D modeling[9] Other studies at Grenoble example several implementations of EPVH for a distributed system use the method of partitioning the input space[10] or distribute the computation to a PC cluster using FlowVR . A recent study at Salford implement a parallel version of J.S Franco's EPVH for realtime reconstruction from video stream, taking advantage of GPU processors[11] . All the above mention has succeed in implement a realtime modeling system for an input of not more than 10 cameras.

Tree structure for accelerate the computation is a well studied topic. An adaptive octree carving was proved efficient for the volume based approach of 3D modeling.[12] Other researches in ray tracing application have also proposed technique for fast, parallel construction of kd-tree [13] The kd-tree in this application is built with cost estimation at every split, using a surface area heuristic (SAH) [14] for a better quality of the tree.

3. Algorithm

3.1 Overview

The algorithm of mkCSG consists of three main parts:

1. Tree expanding: reading all input information and register it to the root node of the tree. At the end of this step, the root node's bounding volume is a rectangular hexahedron containing all vertices, facets and their relevant information.
2. Tree exploration: recursively process every node of the tree. Compute the relative position of the node's bounding volume according to each input polyhedron. If the node is completely inside or outside all meshes then stop processing it further. If it is completely outside/inside all but one mesh, or if its bounding volume or number of vertice is less than a predefine value, do not split, go inside node and find possible vertice/intersections contributing to the final hull. Else split it into two nodes.
3. make facets: building ouput facets from the vertice that are found in the previous step, with its correct orientation.

3.2 Boolean operation on polyhedrons

3.2.1 Overview

Let x be a point in space, polyhedron A is defined by function A(x)=1 if x is inside A and 0 otherwise. A general boolean operation on polyhedrons is defined by f such that:

$$A(x) = f\,(A_1(x), A_2(x), \dots , A_n(x)\,)$$

We say A is the result polyhedron of operation f on the input polyhedrons A_1 , A_2, ...A_n

For example: $A(x) = A_1(x) \setminus A_2(x)$

We can make the following observation:

1. The vertices of the result consist of either primitive vertices from the input polyhedrons(meshes) and/or of secondary vertices created at the intersection between edges and facets of different input polyhedrons.
2. The vertice on the surface of the result polyhedron are also at the surface of the input polyhedrons, in other words, a vertex is on the surface of the output if and only if there exist a change in value of A(x) where there is a change in value of the inherent mesh(es).
3. Consider the cases of two non-coplanar facets intersecting at 1 point and the cases of more than 3 facets intersecting at 1 point as degenerate cases, there are only 2 cases where a new vertex is generated: when an edge intersect with a facet and when 3 facets intersect each other. We further discuss how to identify these points on the surface of output in 3.2.2.
4. As a consequence of the previous observation, there can be new vertices and edges but no new facets to be generated.

3.2.2 Algorithm

The test to see if a point is on surface of the result is interesting because it leads us to the final result geometry while also giveing us the ability to ignore a substantial portion of the computation that do not lead to meaningful information. Indeed anything happens completely inside or outside of the final hull does not alternate the final result, thus it is of benefits to identify them as early as possible. There are three possible types of vertice on the surface of solid: primitive vertex, double vertex and triple vertex.

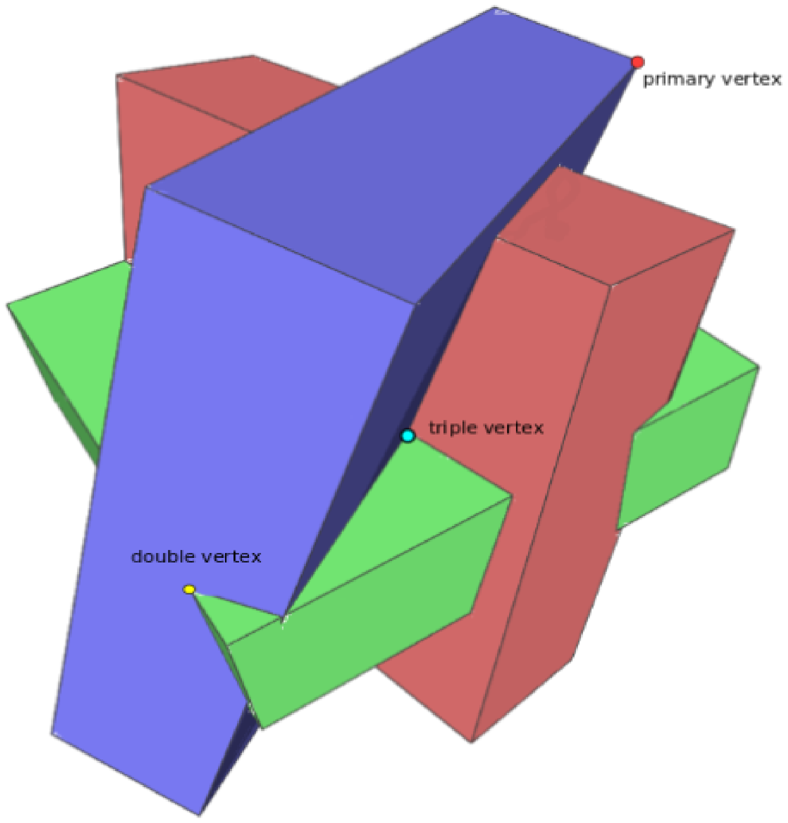


Figure 1: Types of vertices

Primitive vertex is vertex that is originally from one of the input polyhedrons. A primitive vertex is on the surface of the final result if and only if A(x) changes value at x. In other words, if x is a primitive vertex from mesh A_i, then:

$f(A_1(x), \dots, A_i(x)=0, \dots, A_n(x)) \neq f(A_1(x), \dots, A_i(x)=1, \dots, A_n(x))$

Double vertex is the intersection of an edge from A_i and a facet from A_j. At such intersection the space can be divided into 4 section whether it is inside or outside the two incident meshes.

Let $f_{i,j}(x, b, b') = f(A_1(x), \dots, A_i(x)=b, \dots, A_j(x)=b', \dots, A_n(x))$

Then we need to examine the value of $f_{i,j}(x, 0, 0)$, $f_{i,j}(x, 0, 1)$, $f_{i,j}(x, 1, 0)$ and $f_{i,j}(x, 1, 1)$.

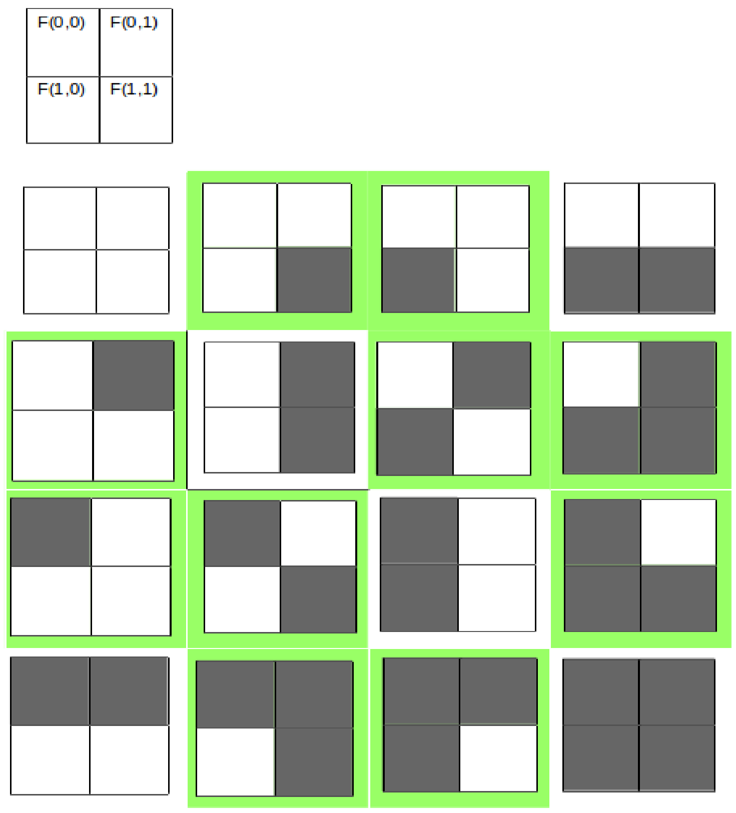


Figure 2: Evaluation of double vertex if it is on the surface of the result polyhedron. The patterns in green return true

Similarly, a **triple vertex** is at the intersection of 3 facets from meshes i, j, and k.

Let $f_{ijk}(x, b, b', b'') = f(\ldots A_i(x)=b, \ldots, A_j(x)=b', \ldots, A_k(x)=b'', \ldots)$

Then we need to examine the value of $f_{ijk}(x, 0,0.0)$ … $f_{ijk}(x,1,1,1)$ simultaneously. A truth table similar to the previous figure can be created with the same logic.

3.3 Data Structure

Bit vector
A bit vector stores the position of one point relative to all input polyhedrons. Bit i of the bit vector can take the value of Inside, Outside or Unknown, accoding to the position of point in space relative to the input polyhedron i. The bit is set everytime after a new vertex is created and help to determine if the point is inside outside or unknown of the final output polyhedron according to the considered boolean operation.

Geometry
Each input polyhedron is present as a mesh of vertices. With the vertex numbering depends on the orientation of the surface of polyhedron. The mesh also keeps a table of facets and polygons that belong to it. The global table of polygons is sorted according to its incoming

mesh so that polygons belong to the same mesh are adjacent to each other.

3.4 KD-tree

The KD-tree is use to reduce the computation complexity of all the vertex iterations. It divides the node into 2 child nodes by intersecting with a plane. A node can be thought of as a bounding box containing all the geometry splited. Each node keep a bit vector indicates it position relative to each input polygons. The bit vector indicates if it is fully inside or outside of each input polyhedron(or unknown if the polyhedron intersects with the node)

Each node contains a table of all its vertices and polygons. It has to keep information of its current state, its bounding volume, and its children nodes. The kdtree includes all the nodes and its children and also on the tree operation policy such as spliting policy.

Node spliting policy is define by the KD-tree. It is plit at the median of the coordinate at dimension c, where c is the dimension with greatest difference. After 'cutting' the node's bounding box into 2 children node bounding box, all polygons on each relative side after the intersection is pushed to the children node and evaluate its bit vector. We also define a condition at which the node is consider a leaf node and no more spliting need to take place.

```
Task split(node){
      Vec3 diff = BoundingBox.max - BoundingBox.min
      dim = dimension with maximum difference
      for all vertice // find median coordinate in dimension[dim]
            tmp += vertice[i]
      done
      threshold = tmp / vertice.size()
      intersection plane : x[dim] = b
      for all polygons
            new polygon_left = polygons[i]
            new polygon_right = polygons[i]
            polygon_left = polygon->intersection(plane, left)
            polygon_right = polygon->intersection(plane, right)
            child1->polygons.push_back(polygon_left)
            child2->polygons.push_back(polygon_right)
      done
}
```

a

b

4. Parallel Algorithm

4.1Task-based versus thread-based programming

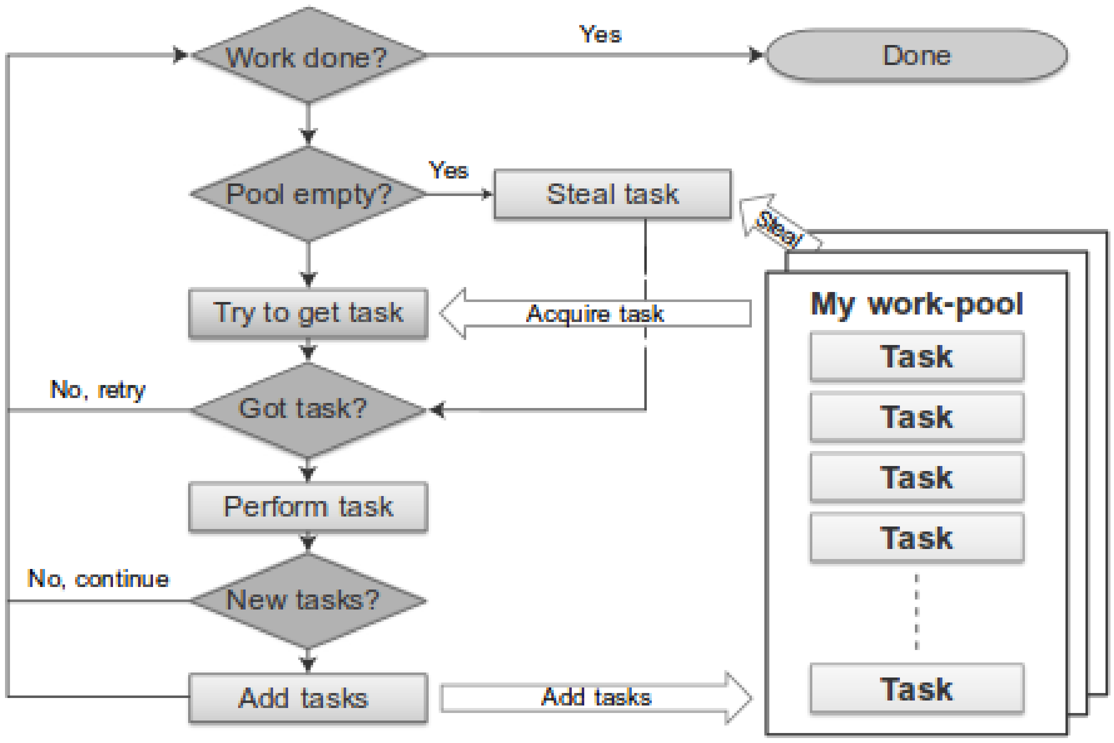
Previous method of parallel programming propose to directly handle threads, where the threads created by a threading package are logical threads, which then map to the physical threads of the hardware. Task-based programming, on the other hand, uses a higher level of abstraction, where the user delimit potential parallelism through tasks. A task-based approach makes it not only more convenient for the programer to code since (s)he is allowed to focus on the parallel logic of the tasks instead of dealing with low level thread interactions and physical synchronization, it is also more efficient than threads in terms of scheduling and work load balancing.

Threads are schedule in a Round Robin scheme, of which time-slices are given to each threads in circular order. This scheme ensure for fairness and starvation-free but does not take it to account the higher level logic of the program, thus does not give an optimal performance. Time-slicing also destroy performance by incurring a lot of context switch between logical threads. On the other hand, the scheduling for task relies on work stealing: ready tasks are first executed by the thread that generates it. If a thread runs out of ready tasks, instead of become idle, it will try to steal ready tasks from other threads selected randomly. This scheduling policy is known to guarantee good performance and is today supported by various programming environtment like Cilk and TBB.

In a thread-based program, highest efficiency is attained when there is exactly one running logical thread per physical thread. If there are less logical threads than physical ones, called undersubscription, it will waste computation power. Oversubscription, when there are more logical threads than physical, on the other hand, incurs overheads. But managing to get an exact number of threads is tricky and not always possible. In addition, it is also important to distribute work evenly across the threads. For our case, the shape of the tree is not known in advance, and usually unbalanced. Therefore one thread can take significant longer processing time than all the others, worsening the execution time. To do load balancing, the Intel Threading Building Block (TBB) relies on work stealing, As long as the defined task is small enough, TBB's scheduler will assigned ready task that is waiting on one thread to another available thread, in effect balancing the work load among all threads. The main principle of work stealing is discussed futher in the next section.

4.2 Work-stealing for dynamic load balancing

A popular method of dynamic load balancing is work stealing. The basic idea of work stealing is to assign to each thread a local task-pool which will be processed mostly by that thread. Any new spawned subtask will be push in to the same work pool as its parent task, which lead to better cache utilization, as subtask usually access the same data as their parent. The tasks in the queue are executed in a depth-first manner, select the newest ready task to run. After a processor is done with its local task pool, it will become a thief: it will steal a ready task from another thread's work pool to execute. This will ensure all ready tasks are executed as early as possible and no processor spending idle time which lead to an evenly amount of work distributed across threads in the end of the program's execution regardless of how uneven they are assigned in the beginning. The general algorithm of work stealing is describle in figure 4.



Work-stealing s is well-suited for our KDTree structure because of several reasons. First of all, as the 3D subject most likely takes an irregular distribution in space (unless it is a uniform cube or sphere locate in the centre of all image, which rarely will be the case), the KD-tree is generally an unbalanced structure. Thus we have to rely on a dynamic scheduler to distribute the work evenly on all processors. Besides, the nature of KD-tree exploration using recursive call and stop at a certain condition can be easily describe with a task. Finally, as children node access the same geometry as their parent node, locality is in favor.

4.3 Recursive tree

TBB's task parttern suits well with our kd-tree algorithm. The kd-tree is a specific example of the divide-and-conquer strategy, in which the original task is recursively subdivided into smaller tasks. In our case, the task of tree traversal is divided into multiple subtasks, at the same time the geometry is split by a cutting plane and the data is also assigned to the subtask. This implies two aspects: first there is no shared memory between tasks so that a bottleneck can be avoided. Next, since the subtasks' data is generated from their predecessor task, a depth-first execution will exploit cache usage.

```
Task exploreLeafNode (node){
      if node is not splited : split(node)
      else if node is splited and is a leaf or is simple enough:
            findNodeVertice(node)
      else { // node is splited and not a leaf, explore its children :

            exploreLeafNode(node->child1)
            exploreLeafNode(node->child2)
      }
}
```

4.4 Parallel_for

Another possibility for exploiting multiple processors is to parallelize loop iterations when the iterations are independent. Parallel_for is a template provided by TBB that executes for-loop in parallel relying internally on work stealing to balance the work load. In parallel_for, the whole range of iteration is recursively split to smaller range. Once an iteration range is split, it is available for other threads to steal. This mechanism balance the load between threads, use cache efficiently and generally have good scalability. To use parallel_for, we must identified the loop where iterations are independent such as the step makeFacet, where we iterate on all input facets and on each one, follow its incident vertice and edges to get the facet that is relevant to the final output.

```
Struct: makeFacet{
 for all vertices:
      if it is mark as not keep: continue;
      else:
            follow its incident half-edge until found a loop
            if (found a loop) record the facet with incident vertice.
}

parallel_for(block_range(start, end, grain_size), makeFacet)
```

5. Analysis

5.1 Granularity

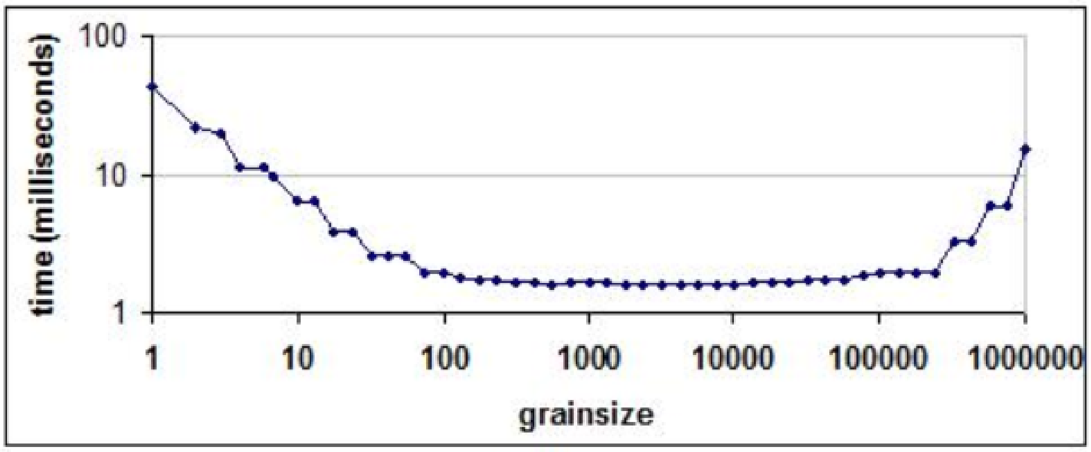


Figure 5: Typical curve of performance depend on granularity. The execution time of a parallel sum varies with grainsize

Granularity is the issue that when the tasks are too small, the work of communication between tasks and assigning task to threads cause too much overhead, on the other hand, if the tasks are too coarse, then performance can suffer from load imbalance due to a lack of expressed parallelism. The performance depending on grain size usually take a "bathtub curve" and a practical approach is to try a few test run to find the a good grain size. In the parallel_for template, TBB is implemented with a built in mechanism to select grain size automatically. However, in the recursive tree exploration, parallelism is not explicit to the compiler, thus the grain size must be set manually. Without explicitly limiting the grain size to a certain level, TBB creates at each node a new task, giving the scheduler thousands of task to manage, incurring terrible overhead. Therefore we need to define a threshold point where after that the node splitted simply created 2 subnodes but no new task.

The figure below shows the geometry inside a node at level 6 and at level 20 of dataset knots from 42 cameras. At level 6 the node contains the whole final result and the geometry is much more complicated than at level 20 it is quite simple with only around a hundred polygons.

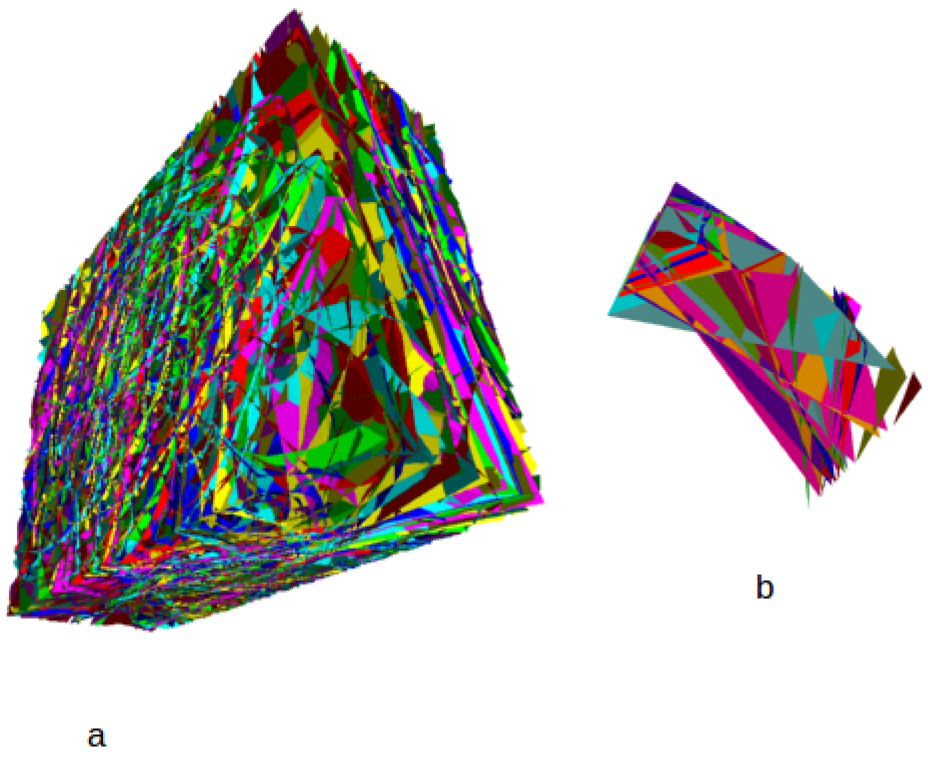


Figure 6: Complexity of geometry in different node level: a) at level 6 and b) at level 20

We can use either the depth of node or number of polygons to control the granularity. We set that on the top level the program will push task to the work pool until we have enough tasks at a certain point, we stop creating tasks and the thread will continue on to process descendant nodes sequentially. Starting at the root node, only 1 task is created. As we set the threshold to a greater level, more small tasks is created, granting a more load balance thus increasing performance, but at a certain level when the task created is too fine the overhead is too high and it decreases the performance.

5.2 Parallel of top tree task

The kd-tree is constructed by recursively spliting current node into two children node. Therefore, after one split, there are 2 task created. As the number of nodes grow exponentially, after some time there will be enough tasks for all processors to run. But in the beginning this exponential growth is quite slow, and also the node in the beginning is bigger, so that the time to split a node is also longer. This lack of parallelism in the exploration of the top of the tree significantly impair the performance, as shown in the figure 7 , in the beginning there is a lot of threads in idle state.

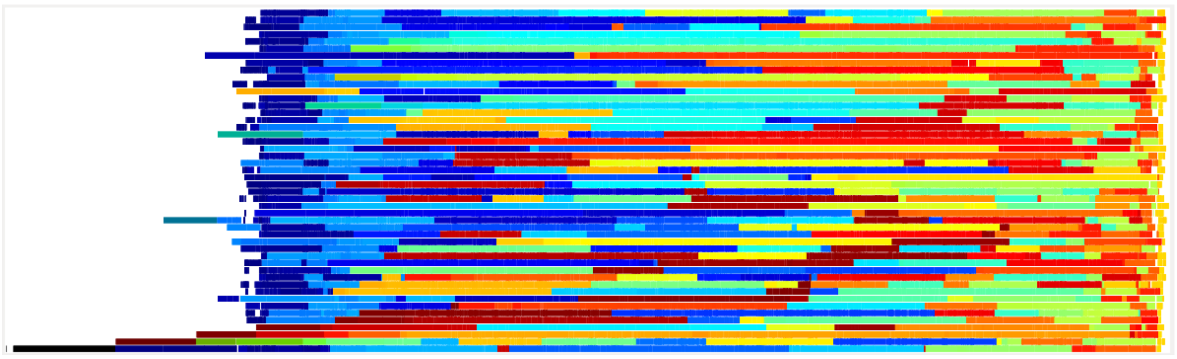


Figure 7: Thread occupancy: a few number of threads are running in the beginning

To overcome this problem, we decided to parallelize the execution at the top of the tree by concurrently spliting it mesh by mesh. Instead of sequentially iterating through all vertices, we define a task split that iterate through all vertices of only one mesh, each mesh then can be handle by independent processor. A parallel_for loop will execute the task split on every meshes, completely split the node. An extra step to combine each thread's results must be done when all meshes are finish spliting.

5.3 Memory allocation and synchronization

Memory allocation is a bottleneck in concurrent environments, as the default memory allocator is not designed for concurrent programming, threads have to compete to have mutual exclusive access to the shared memory. This affects the ability of the code to scale. In our experiment, by replace the default memory allocator to tbb_malloc the exploration time is significantly reduce for program running with five or more threads.(Figure 9)

Another challenge of the algorithm to parallel programming is the need to simultaneously update data. At the end of every of parallel split we have to combined the bit vector to get the correct position of node relative to each mesh. We also have to push back the result from each thread's local output. This is currently an inevitable bottleneck for the performance of the program.

6. Results

The multicore platform used in the experiment is a 48 cores AMD ManyCours platform with 256GB of main memory. The machine has 8 NUMA nodes each node consisting of 6 cores running at 2.2GHz and 5MB shared L3 cache.

The dataset in the experiment is the synthetic knots image from 42 view, each view yeilding a contour of average 200 points. Compare to other dataset this datasets is interesting because it has quite complex shapes and contours with several holes inside the silhouette. The number of points and cameras are also much larger than other datasets tested. A good performance on this datasets is promising for scalability on a 64 camera system.

Each of the test point is run 10 times on 48 cores machine and taken the average performance.

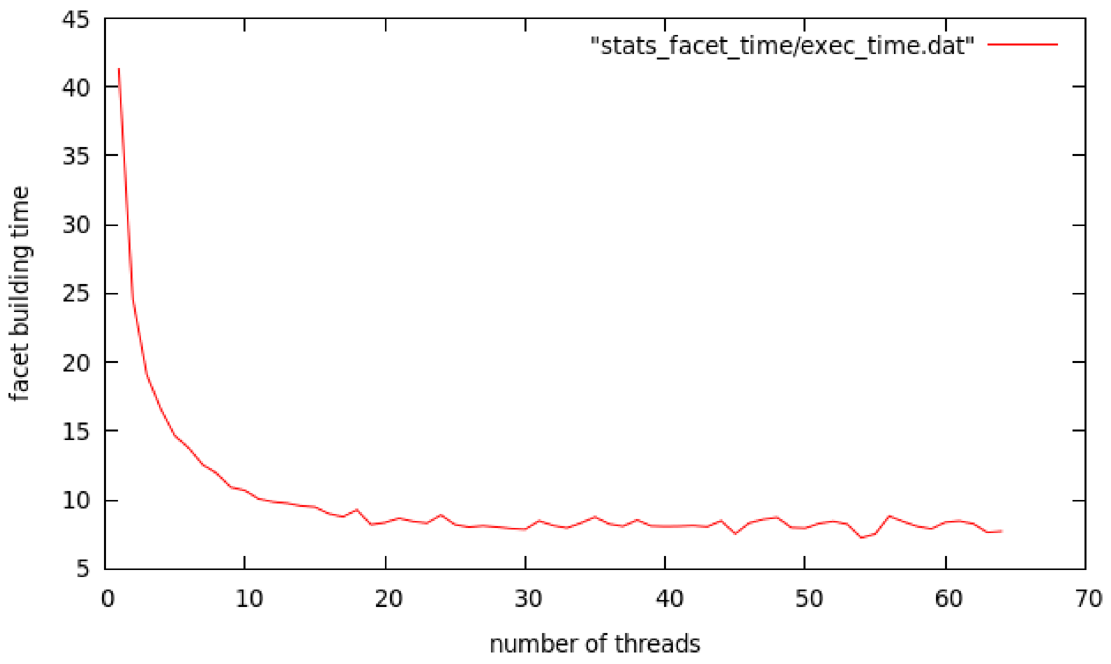


Figure 8: facet time depending on number on running threads

The parallel_for loop has improves the facet building time by 8 from 41 ms to 5ms. It scales linearly for up to 5 threads. Beyond, for more threads running, the performance gain slows down. Since the work load at this step is small, potential parallelism is limited, not providing enough work to keep busy all threads.

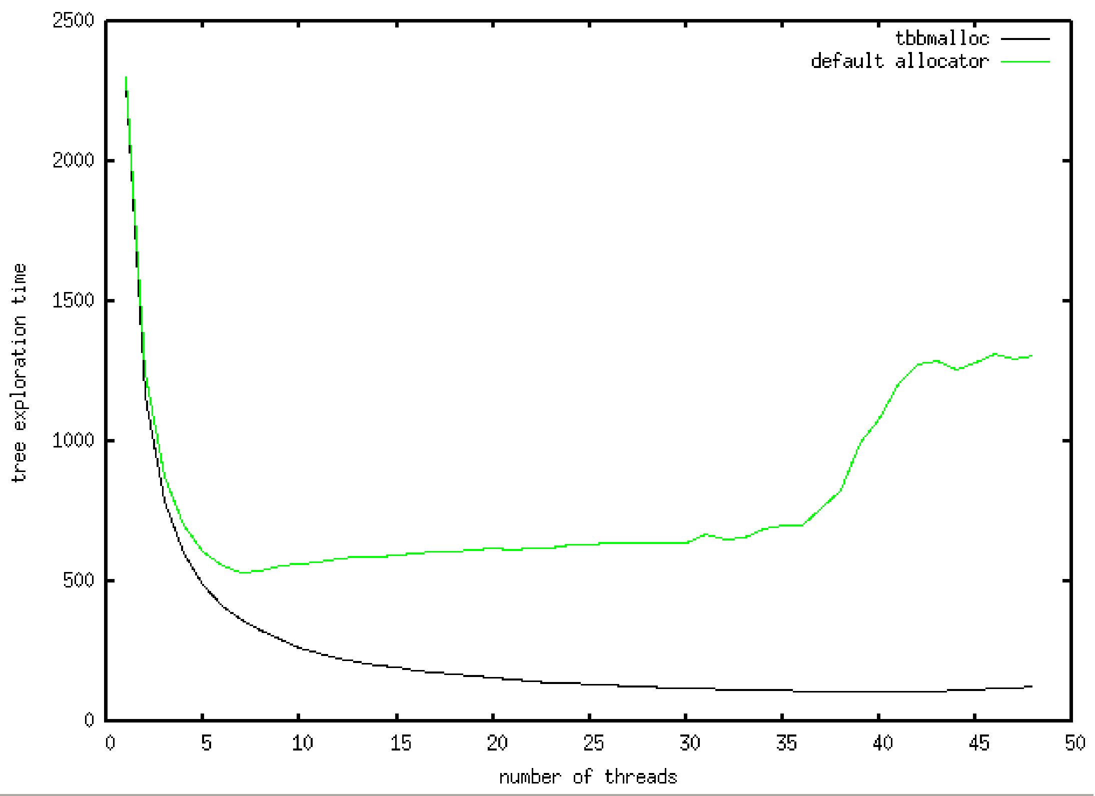


Figure 9: Tree exploration time depending on number of threads

With tbbmalloc, the performance continue to scale for more than 5 threads, achieving a better exploration time of 100 ms, 22 times faster than the sequential execution. The performance does not scale linearly for more than 5 threads but improves slowly. This step of kd-tree exploration incurs a lot of memory allocation, and even though tbbmalloc have help, memory still create a contention.

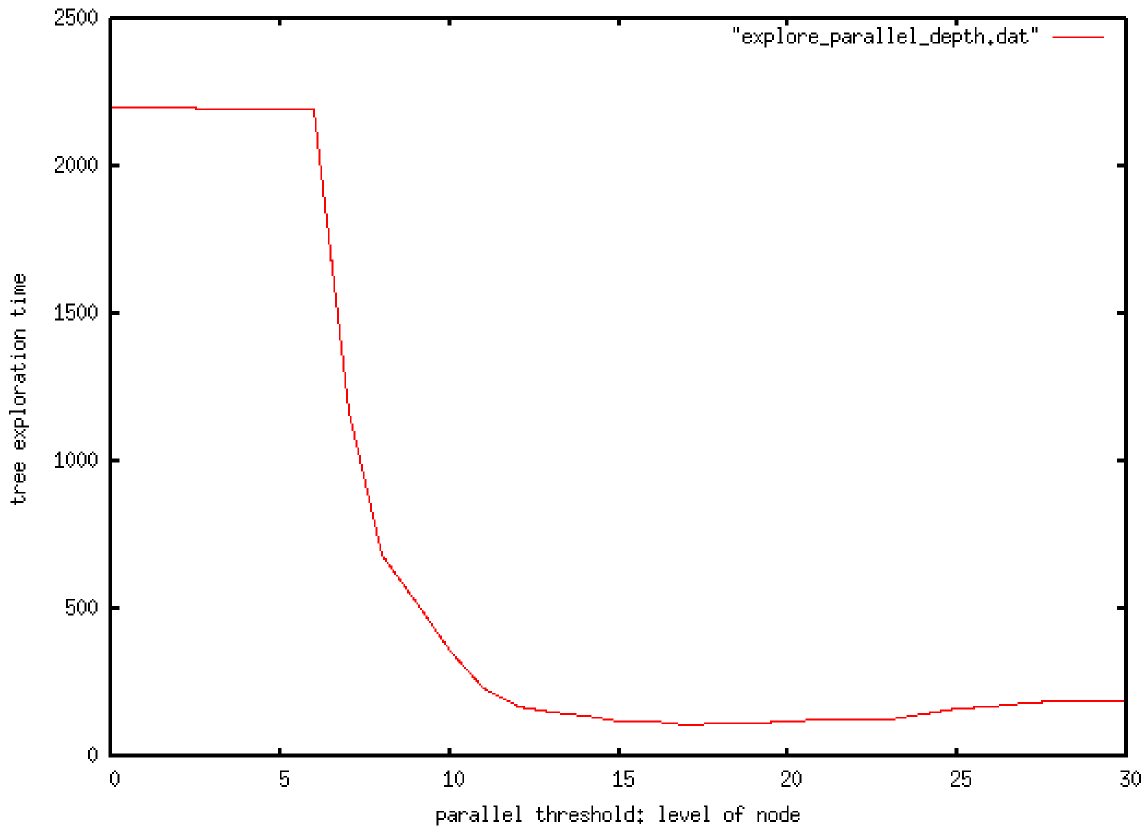


Figure 10: Effect of parallel threshold by the level of node

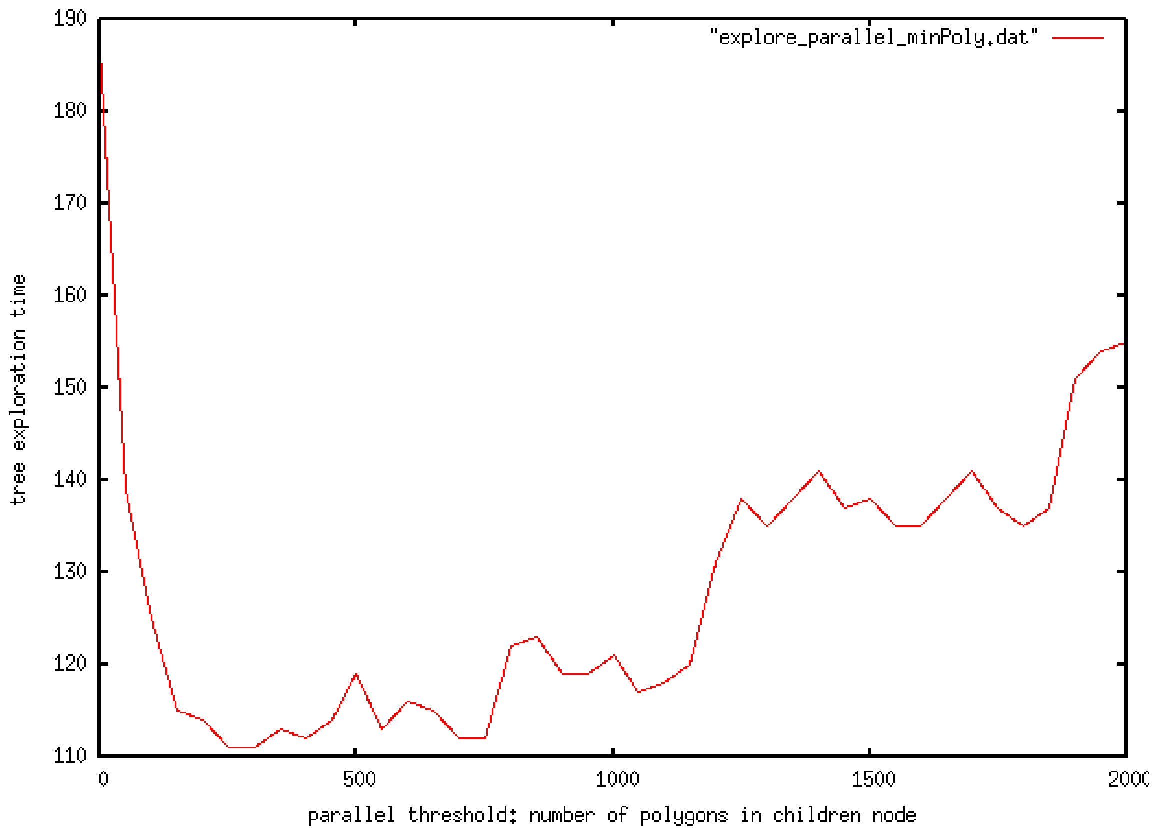


Figure 11: Effect of parallel threshold by number of polygons

We study the effect of granularity by example run on a sweeping threshold. The threshold of node level gets the best performance at the level of 15 at 120ms. For the threshold by minimum number of polygons in node, the performance although slightly fluctuate, it still resembles the classic bathtub curve of performance over granularity. The best performance for this threshold policy is 115ms which is considerably the same performance at the other policy. Recognize that using the depth policy is not guarantee the task created will be of the same size, because the tree is unbalanced, and we will miss some node deep in the tree but have lots of geometry. We set the threshold to have at least 100 polygons to be executed in parallel.

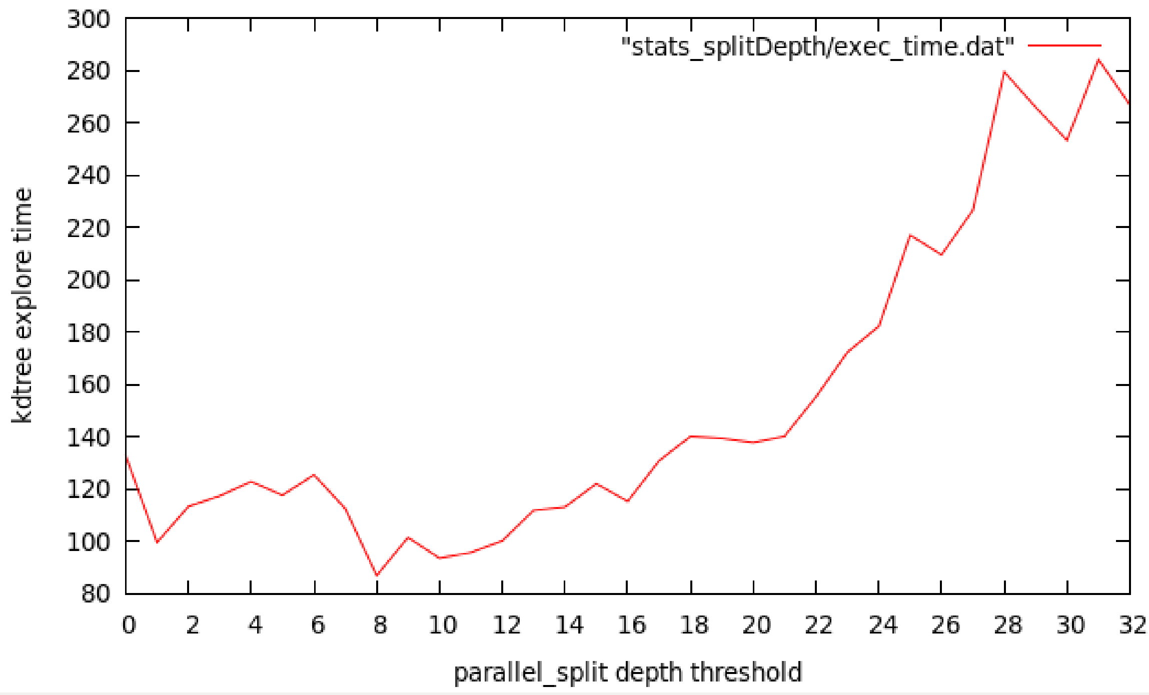


Figure 12: performance of parallel_split depending on threshold level

The effect of parallel split mesh by mesh is represented here. It has earn us about 25ms of performance for a parallel_split up to level 8 over no parallel split at all. It is quite clear we should only limit parallel split for concurrency in the top node of of the tree, since letting it go in deeper nodes will slow down the split by having to combine result at the end.

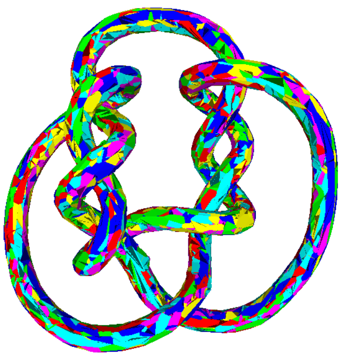


Figure 13: Knots model output

Table 1: Breakdown of execution time

	Tree expanding	Tree exploration	Facet building	Real execution time
Parallel run	80ms	100ms	5ms	0.4s
Sequential run	70ms	2200ms	40ms	2.5s

Table 1 shows the execution time on each step of the algorithm and the real execution time of the program. The real execution time includes also reading and writing input/output to file thus taken a proportionally longer elapsed time.

Besides the knots, we tested our algorithm on multi-camera sequences accquired on the Grimage platform. The model below is get from 8 cameras with a total of less than 1000 input points for segmented contours. Test run on 4 core pc Intel i7 clock rate 2.20 GHz memory 7941 MB, cache size 6144KB

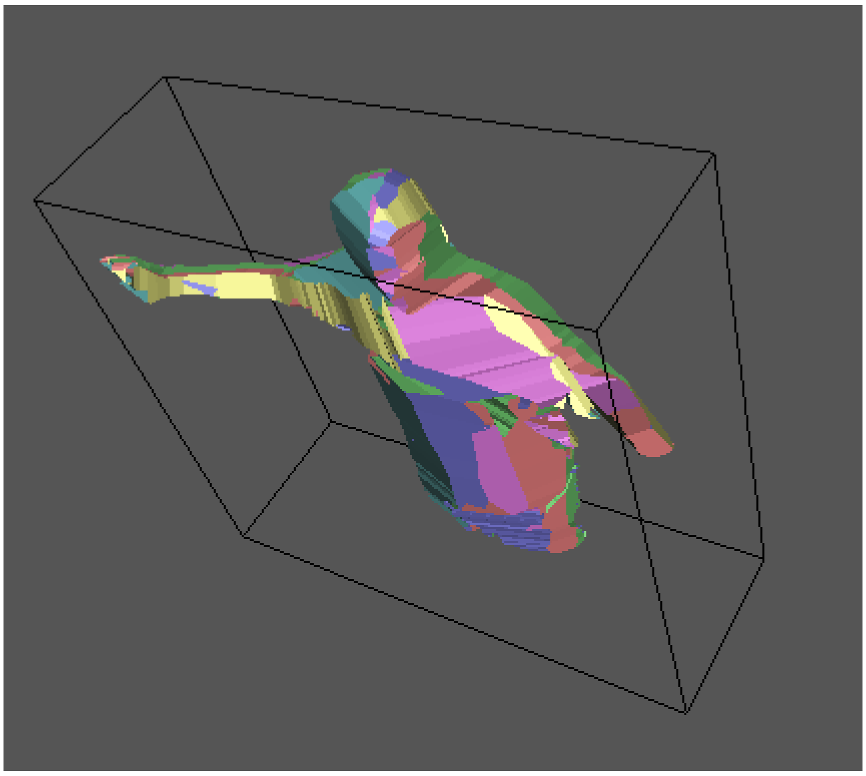


Figure 14: Result model of realistic dataset 8 cameras

Table 2: Execution time realistic dataset 8 cameras

	Tree expanding	Tree exploration	Facet building	Real execution time
Parallel run	2.2ms	12ms	1ms	0.04s
Sequential run	3.4ms	48ms	2.9ms	0.08s

The performance of parallel run on laptop on a 8 cameras datasets is qualified for a realtime application with a processing rate equivalent to 25 fps, the result in reality without file input/output should be even faster.

7. Conclusion

This work explores the possibility of running 3D polyhedrons intersection modeling in realtime and it has been shown to be very likely. We achieved a speed up of 22 on the task of tree exploration on a 48 cores machine. We have also identified other parts of the algorithm where parallel processing is possible and implement them in a parallel for loop where we can get a 8 to 10 time speedup of a specific task.

We also did a study on the effect of granularity on the overhead and found a optimum threshold policy to control granularity for the kd-tree parallel task, using the maximum depth of node and minimum number of polygons inside a node. With the defined granularity and task stealing, we have achieved a balanced workload. We also optimize the result by combining different parallel programming patterns for top part and lower part of the tree to maximize cpu usages.

As the current stage of platform development, the test can only be performed on existing datasets and the execution timed of the whole application also includes reading from and writing to input/output file. This does not reflect the real situation of online processing on the platform since file I/O can be quite slow compared to input/output to the network. Therefore we expect to have a better performance on the real platform of Kinovis when it is ready. Currently we are also in the process of creating synthetic datasets that are more resemble the scene in Kinovis acquisition platform with 64 cameras. Once done the datasets will give better hint of the actual realistic run.

The future work could be studying what kind of other tree structure and heuristic that could improve performance. A direction to a more efficient tree could be how to quickly propagate the node through the whole acquisition space and focus the computation power on the relevant nodes with the object inside. Another direction is for a more balanced tree so that task stealing would happen less which leads to less overhead and better cache usage. The trade-off between creating the “best” tree and the time consumed to construct such tree must also be taken into account.

8. Bibliography

[1] B. Petit, J.-D. Lesage, C. Menier, J. Allard, J.-S. Franco, B. Raffin, E. Boyer, and F. Faure, “Multicamera Real-Time 3D Modeling for Telepresence and Remote Collaboration,” *International Journal of Digital Multimedia Broadcasting*, vol. 2010, pp. 1–12, 2010.

[2] B. Petit, J.-D. Lesage, E. Boyer, J.-S. Franco, and B. Raffin, “Remote and Collaborative 3D Interactions,” in *Proceedings of the 3DTV Conference (3DTV-CON 2009)*, Postdam, Germany, 2009.

[3] J. Allard, C. Ménier, B. Raffin, E. Boyer, and F. Faure, “Grimage: Markerless 3D Interactions,” in *Proceedings of ACM SIGGRAPH 07*, San Diego, USA, 2007.

[2] R. Szeliski, “Rapid Octree Construction from Image Sequences,” *CVGIP: Image Understanding*, vol. 58, no. 1, pp. 23–32, Jul. 1993.

[5] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan, “Image-based Visual Hulls,” in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2000, pp. 369–374.

[6] J.-S. Franco and E. Boyer, “Efficient Polyhedral Modeling from Silhouettes,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 3, pp. 414–427, Mar. 2009.

[7] B. Petit, T. Dupeux, B. Bossavit, J. Legaux, B. Raffin, E. Melin, J.-S. Franco, I. Assenmacher, and E. Boyer, “A 3D Data Intensive Tele-immersive Grid,” in *ACM Multimedia (ACMM’10)*, Firenze, Italia, 2010.

[8] B. G. Baumgart, “Geometric Modeling for Computer Vision.,” Stanford University, Stanford, CA, USA, 1974.

[9] L. Soares, C. Ménier, B. Raffin, and J.-L. Roch, “Work Stealing for Time-constrained Octree Exploration: Application to Real-time 3D Modeling,” in *Proceedings of the 7th Eurographics Conference on Parallel Graphics and Visualization*, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 61–68.

[10] J.-S. Franco, C. Ménier, E. Boyer, and B. Raffin, “A Distributed Approach for Real Time 3D Modeling,” in *Proceedings of the IEEE Workshop on Real Time 3D Sensors and Their Use*, Washington, USA, 2004.

[11] T. Duckworth and D. J. Roberts, “Parallel processing for real-time 3D reconstruction from video streams,” *J Real-Time Image Proc*, pp. 1–19, Dec. 2012.

[12] L. Soares, C. Ménier, B. Raffin, and J.-L. Roch, “Parallel Adaptive Octree Carving for Real-time 3D Modeling,” in *IEEE Virtual Reality Conference*, Charlotte, USA, 2007.

[13] M. Shevtsov, A. Soupikov, and A. Kapustin, “Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes,” *Computer Graphics Forum*, vol. 26, no. 3, pp. 395–404, Sep. 2007.

[14] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart, “Parallel SAH k-D Tree Construction,” in *Proceedings of the Conference on High Performance Graphics*, Aire-la-Ville, Switzerland, Switzerland, 2010, pp. 77–86.