A Distributed Approach for Real Time 3D Modeling

Jean-Sébastien Franco[1] Clément Ménier[1,2] Edmond Boyer[1] Bruno Raffin[2]
[1]GRAVIR - INRIA Rhône-Alpes, France [2]ID - INRIA Rhône-Alpes, France

Abstract

This paper addresses the problem of real time 3D modeling from images with multiple cameras. Environments where multiple cameras and PCs are present are becoming usual, mainly due to new camera technologies and high computing power of modern PCs. However most applications in computer vision are based on a single, or few PCs for computations and do not scale. Our motivation in this paper is therefore to propose a distributed framework which allows to compute precise 3D models in real time with a variable number of cameras, this through an optimal use of the several PCs which are generally present. We focus in this paper on silhouette based modeling approaches and investigate how to efficiently partition the associated tasks over a set of PCs. Our contribution is a distribution scheme that applies to the different types of approaches in this field and allows for real time applications. Such a scheme relies on different accessible levels of parallelization, from individual task partitions to concurrent executions, yielding in turn controls on both latency and frame rate of the modeling system. We report on the application of the presented framework to visual hull modeling applications. In particular, we show that precise surface models can be computed in real time with standard components. Results with synthetic data and preliminary results in real contexts are presented.

1. Introduction

Recent advances in camera technologies have generalized real time acquisition of digital images, and today, any modern PC can acquire such images at standard video rates. This allows complete acquisition systems to be built by simply connecting sets of digital cameras and PCs, without help from specific components. The interest arises in various application domains where digital cameras are involved and where image information extracted in real time is required for interaction or control purposes. These domains include, for instance, scene virtualizations, video surveillances or human-machine interfaces. However, while much research has been devoted to algorithms that address the situation where a few cameras are connected to a single or few PCs, less efforts have been made toward larger sets of cameras and PCs which are, on the other hand, becoming standard. Moreover, most computer vision applications that make use of several cameras connected to several PCs do not take advantage of the available computing power and generally rely on a single PC for computations. In this paper, we address these scalability and optimality issues by considering parallel strategies for 3D modeling computations. The objective is to propose practical and scalable solutions to produce highly precise 3D models in real time using multiple cameras.

Only a few distributed 3D modeling systems have been proposed and demonstrated in the past. The CMU robotics institute introduced a 3D dome with around 50 cameras for virtualization [Narayanan98]; the 3D scene model being built using a stereo-vision approach. Other systems have also been proposed at CMU with fewer cameras and a voxel based approach [Cheung00], or its combination with a stereo based approach [Cheung03]. However, these systems do either work off-line, or in real time but with a few cameras, since no parallelisation is considered for modeling computations. Another class of real time but non-parallel approaches make use of graphic cards to directly render new viewpoint images [Li03]. Using graphic card hardware highly accelerates the rendering process but such systems still rely on a single PC for computations and do not provide explicit 3D models as required by many applications. In [Borovikov03], a parallel framework that stores, retrieves and processes video streams on PC clusters is presented, but the emphasis is put on data management and real time applications are not considered. Real time and parallel modeling systems have also been proposed to handle voxel based modeling methods [Kameda00, Arita01]. Voxel based methods produce discrete 3D models by using regular space discretizations where parallelepipedic cells -the voxels- are carved according to image information [Slabaugh01, Dyer01]. Such methods can easily be parallelized since a significant part of the computations is achieved independently per voxel. The mentioned approaches make use of this fact and report good performance. However, voxel based methods are imprecise and time con-

0-7695-2158-4/04 $20.00 (C) 2004 IEEE

suming. Furthermore the principles developed in the proposed approaches do not easily extend to possibly better modeling methods. In this paper, we attempt to develop concepts at a higher abstraction level in order to make their application possible to various modeling methods. Another interesting work [François01] deals with video streams and proposes a multi-threading strategy to improve latency and frame rate when processing video streams for segmentation or tracking tasks. We mention also here the Skipper project [Sérot02] which develops a parallel programming environment for image processing. The two latter works also tackle distribution issues related to computer vision, but they provide middleware solutions mainly and do not consider algorithmic aspects as necessary with the targeted modeling applications.

In this paper, we present a real time and parallel architecture for multi-view algorithms and report on its application to 3D modeling applications and, in particular, to silhouette based approaches. Our goal is to provide scalable solutions by using a parallelization methodology. Such a methodology is intended to be general enough to apply to different contexts while maintaining the required parallelization effort reasonable. Our contribution with respect to the mentioned works is twofold: first, we extend parallelization concepts already proposed to multi-view algorithms; second, we apply this concept to silhouette based approaches and propose new practical implementations that are scalable while surpassing standard voxel based approaches.

The paper is organized as follows. Section 2 introduces our distribution scheme for parallelizing multi-view tasks. In section 3, their application to visual hull computations using image silhouettes is presented. Section 4 demonstrates the proposed principles in a real context and gives numerical evidence on synthetic data.

2. Distribution Scheme

In this section we present our methodology to parallelize multi-view algorithms. It relies on classical parallelization techniques. The simplicity of these techniques limit the effort required to parallelize existing algorithms. Experimental results show that this methodology leads to good performance.

Let n be the number of cameras available and m the number of hosts (PCs). We assume each camera is connected to one host dedicated to acquisition and local image processing. We consider that all cameras issue images at the same *frame rate*. We will consider that a *frame* corresponds to the set of n images taken at a time t. We also assume that m is greater than n. The extra $p = m-n$ hosts are dedicated to computation. Hosts are interconnected through a standard network. Accessing data on a distant host takes much more time than accessing local data. We do not use any tool masking this disparity, like a virtually shared memory. They generally lead to lower performance than tools that enable the user to take into account this disparity to optimize data transfers, like message passing libraries [Gropp94]. As we will see, the effort of explicitly handling data transfers is relatively low and worthwhile.

As a performance criterion, we measure the *speed-up*, obtained by dividing the sequential execution time by the parallel execution time. The efficiency of the parallelization increases as the speed-up factor nears the number of processors. For real-time constraints, we measure the frame processing rate and the *latency*, i.e. the time to process a single frame.

The methodology we propose is based on two different levels of parallelization, a *stream level* parallelization and a *frame level* parallelization.

2.1. Stream Level Parallelization

Multi-view applications process a sequence of frames, called a *stream*. A very classical method to speed up stream based applications is to use a *pipeline*. The application is split in a sequence of stages (see fig. 1(b)), each stage being executed on different hosts. It enables a host to work on frame t while the next host in the pipe-line stage works on frame $t-1$. The different stages are naturally extracted from the structure of the program. Trying to redesign the application into a longer pipe-line is time consuming and increases the latency due to extra communications.

A stage is *time-independent* if it does not use temporal consistency, i.e. the process of frame t does not depend on the results from preceding frames. It enables to duplicate identical frame computation schemes on different hosts, called *processing units* (see fig. 1(c)). A new frame t can be processed as soon as one of the processing units is available. The number of processing units should be large enough to avoid any frame to wait for its processing. Adapting this technique to a time-dependent stage may still be possible but requires advanced scheduling strategies and extra communications.

This scheme can be applied to the classical voxel-based approach. Frames go through 2 processing steps, background extraction and voxel carving that can be assigned to 2 pipe-line stages. The first stage being usually much faster than the second one, several processing units can be dedicated to this time-independent second stage.

2.2. Frame Level Parallelization

The preceding distribution techniques can significantly improve the processing frame rate. However, the latency is negatively affected. The pipe-line introduces extra communication time that increases the latency, thus reducing the reactivity of the system. To improve latency, one can re-

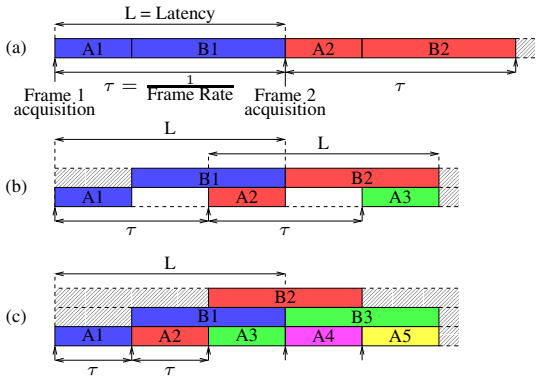

Figure 1: Different levels of parallelization proposed by our framework. A and B are the two computation stages of the program. At and Bt relate to the processing of frame t. Each row corresponds to a processor. Colored blocks correspond to a task execution and white blocks to inactivity periods. The graph (a) represents a sequential execution, the graph (b) a 2 stage pipeline execution. Graph (c) adds a second processing unit for the second pipeline stage B.

duce the time taken by a stage to process a single frame. This can be done by parallelizing the work done on a single frame among several hosts of a single processing unit (see fig. 2). We base our scheme on the classical *Bulk Synchronous Programming* (BSP) model [Valiant90] that proposes a trade-off between performance and programming complexity. The execution is done in a sequence of phases, each one decomposed in a data exchange involving all hosts followed by local computations performed on each host. This model eases parallel algorithm description as it splits communication from computation. Based on this BSP approach, we propose a 3 phase scheme for parallelizing processing unit computations:

- **Data preparation**: the first phase (input data distribution) consists in sending the input data to the hosts requiring them. Next, each host locally performs the initialization computations needed for the next parallel phase.
- **Parallel computation**: in parallel, each host executes locally (no communication) a different task assigned to it.
- **Sequential computation**: all partial results from the parallel computation phase are gathered on one host. This host sequentially performs the remaining computation that could not have been parallelized in the previous phase. Depending on the requirements of the next pipe-line stage, sequential computation can be duplicated on several hosts. It can enable to reduce communication load to transfer data to this next stage.

Though very simple this model is very generic. In worst cases, all the computation is done in the last sequential phase. However this is trivially inefficient. We later show that it is possible to obtain a parallel computation phase significantly larger than the two other phases. In such a situation we will show that this scheme leads to real-time performance.

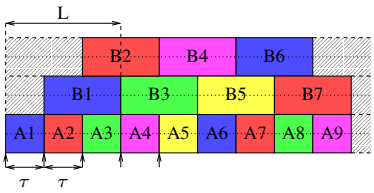

Figure 2: Frame level parallelization (2 processors for stage A and 4 processors for stage B). It shows latency improvement in comparison with stream level parallelization (see fig. 1.)

We can illustrate application of our scheme to the parallelization of voxel carving as described by Arita *et al.* [Arita01]:

- Data preparation: initialize locally the voxel space.
- Parallel computation: for each image, compute the visual cone in the local voxel space.
- Sequential computation: gather all visual cones on one host and compute their intersection.

Results given by Arita show that this scheme yields high performance. Generally, our 3 phase scheme does not lead to an optimal parallelization. Many optimizations can still be done. On this voxel example, Borovikov *et al.* has given an algorithmic optimization [Borovikov03] for computing the voxel cone intersection in a more complex way that cannot be represented with this 3 phase scheme. However we show in this paper that the proposed methodology offers a simple yet efficient scheme to address stream and frame level parallelization for real-time constraints.

3. Silhouette-Based Modeling Approaches

We now deal with silhouette-based modeling approaches and how to make them suitable for a real-time context using our methodology. We focus on visual hull reconstruction approaches, as they are quite popularly used for 3D modeling from multiple views given their speed and simplicity. Recall that the visual hull is a simple approximation of the scene objects defined as the maximal shape consistent with the silhouettes of the objects in the input views [Laurentini94]. A number of algorithms have been proposed for computing the visual hull. Some use a discrete partitioning of space in voxels. Such volume-based

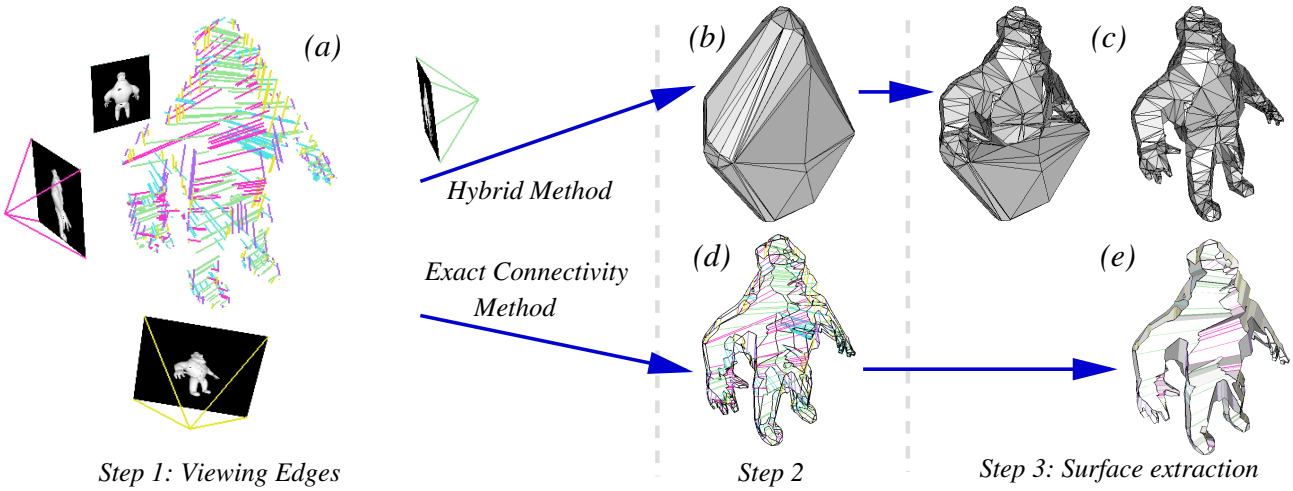

Figure 3: Outline of the visual hull modeling techniques chosen for parallelization. (a) Images of an object are taken, silhouettes are identified, their contours vectorized, and viewing edges are computed for each point of the discretization. (b) The hybrid method computes the Delaunay tetrahedron decomposition of space based on viewing edge vertices. (c) Each tetrahedron is carved according to silhouette consistency, and the final visual hull model is obtained. (d) The exact connectivity method computes the cone intersection components belonging to the visual hull, to complete the entire visual hull polyhedron mesh. (e) Faces are extracted from the mesh representation and the final polyhedron model of the visual hull is obtained.

schemes prove to be simple to implement and can be very fast. We have given a case study for the parallelization of these approaches in the previous section. In this section, we will therefore focus on the parallelization of methods that have never been studied before in this context.

Namely, a recently popular category of *surface-based* modeling approaches have focused on recovering the surface of the visual hull, and provide a polyhedral representation of the visual hull surface as output [Baumgart75, Matusik01], some giving additional topological guarantees with a simpler framework [Franco03]. An interesting hybrid approach also exists that combines advantages of both volume and surface-based families [Boyer03]. While these methods provide a precise model of the visual hull and are generally fast, they are still too slow for a hard real-time setup with as many as 10 cameras. On the other hand this makes them outstanding potential beneficiaries of parallelization. In this context we can show that parallelism is a tool to bridge the gap between generally fast vision algorithms, and vision algorithms that *guarantee* very high frame processing rates of 30fps or above.

3.1. Outline of the Modeling Methods

In order to offer a broad view of the parallelization of silhouette-based approaches, we will focus on two of the most recent methods, the *hybrid* method [Boyer03], which offers a robust trade-off between volume and surface-based approaches, and one of the available surface-based methods, the *exact connectivity* method [Franco03]. See figure 3 for an overview. Recall the context of such methods: n calibrated cameras are used to generate n views of an object; a standard background subtraction process is used to extract the silhouette bitmaps from the images. The contours of the obtained silhouette masks are vectorized so as to obtain oriented 2D polygons bounding the silhouette regions. This discrete representation of silhouettes induces a discrete visual hull polyhedron. The hybrid method provides a close approximation of this polyhedron, while the exact connectivity method computes it exactly.

Three steps are used to achieve the reconstruction goal in both cases, as depicted in figure 3. The first step, common to both methods, computes an initial subset of the visual hull geometry, the *viewing edges*, in the form of points and edges located on the viewing lines of each discrete silhouette contour point (details follow in 3.2). The second step's common goal is to compute an intermediate representation which implicitely contains the visual hull surface. To this goal, the hybrid method partitions space into convex cells, which can easily be carved according to silhouette consistency of their projection in images. In contrast, the exact connectivity method computes the exact visual hull polyhedron as a generalized cone intersection. Finally, the third step's common goal is to identify the underlying surface information, by extracting the visual hull interface polygons from the previous representation. The following sections give more details about these steps.

Note that, as a first possibility for applying our methodology, we can easily identify each conceptual step of the methods with a stage in a multi processing unit pipe-line, to increase the output frame rate. This is valid since the algorithms are intrisincally time-independent: each set of

silhouettes in frame t is used for reconstructing a shape, and the information will never be used again in subsequent frames. We will now deal with more specific issues in the following sections.

3.2. Computing the Viewing Edges

We now describe the computation of viewing edges at discrete image contour vertices, as it is a common processing step in the presented methods (see fig. 3).

Viewing edges are intervals along viewing lines. They correspond to viewing lines contributions to the visual hull surface and are thus associated to image points on silhouette contours. As such, viewing edges are simply obtained by computing the set of intervals along a viewing line that project inside all silhouettes (see fig. 4).

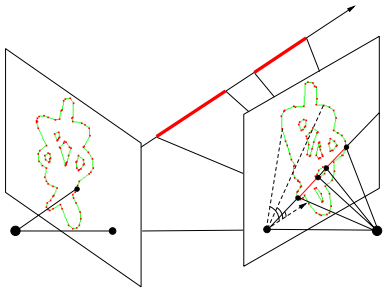

Figure 4: Viewing edges (in bold) along the viewing line. Epipolar line angles can be used to accelerate the search for the image segments intersecting the epipolar line.

Interestingly, this algorithm provides great freedom for frame-level parallelism, as it consists in the computation of a set of numerous partial but independent results. That is, each viewing line's contributions can be computed regardless of all others, assuming the silhouette information from all images is available. An efficient frame-level parallelization of viewing edge computation can hence be obtained by partitioning all viewing lines of all images in p sets during the data preparation phase, and distributing each batch to one of p hosts for processing (parallel computation phase). One must be careful in balancing the workload between hosts, in order to reduce the time spent waiting for the slowest host. Building sets of identical cardinality during data preparation proved to be efficient as we will show. Observe that this parallel scheme heavily constrains how we will perform data preparation: as each host requires all silhouette information, silhouette contours must also be broadcasted during that phase. Finalization of the task simply consists in gathering the union of all partial results on the hosts that require it, during the sequential computation phase.

We are able to achieve speed-ups of the order of 8 with 10 hosts, which is very good, especially given the low effort required to parallelize the algorithm. Higher speed-ups can be achieved, but with a substantially higher complexity, much at the expense of the gain/effort tradeoff.

3.3. A Distributed Hybrid Method

We will now describe the parallelization of the hybrid method [Boyer03]. After computing the viewing edges, the hybrid method uses a Delaunay triangulation of the viewing edge vertices to obtain a decomposition of space into tetrahedrons, as a second step (see fig. 3). The union of these tetrahedrons form the convex hull of the input points: some of them must be carved in order to isolate the visual hull. The discretization consists of convex cells of a more generalized and flexible shape than regular voxels, but can still be carved with voxel-like silhouette consistency checks such as those in [Cheung00]. This is used in the third step to determine which tetrahedrons lie inside or outside the visual hull, and the surface polygons are extracted from this model by simply isolating the triangles which lie at the interface between the two regions.

Although the algorithm is conceptually simple, building its parallel counterpart brings challenges we have to account for as we seek to apply our proposed methodology. The main issue here is the Delaunay triangulation, which generates many partial, but globally constrained results: the Delaunay tetrahedrons. Some possibilities for distributing the Delaunay triangulation have been explored [Cignoni93], with the main idea of subdividing the problem among space regions where concurrent tasks take place. This idea can be applied to many vision algorithms. One obstacle, also widely generic, is the complexity of detecting and dealing with region interrelationships in the algorithm. In the case of the Delaunay triangulation, a programmer would spend most of his time re-implementing the tedious algorithmics intrinsic to such a method. Under such conditions, it is wise to sacrifice system reactivity to implementation simplicity. Stream level parallelization can still be used to improve the frame rate of an already available sequential code. Yet we will see in the next section a case where tackling the multiple region interdependency problem is worthwhile.

However we do have another opportunity for parallelization, as the cell carving task is much friendlier. Much like in a usual volume-based technique, the per-cell carving results are independent, which ensures a well-behaved parallel execution phase. The only requirement is that all silhouette information is available at all hosts, which can be provided for during data preparation. The sequential execution phase will then simply gather the carve state of all tetrahedrons, and finally extract the surface triangles from this information, as this takes very little time and does not require distribution. Under such favorable conditions we are able in this carving task context to reach speed-ups of 9.5 for 10 hosts.

3.4. A Distributed Surface-Based Method

We now briefly describe the application of our proposed parallelization methodology on the exact connectiv-

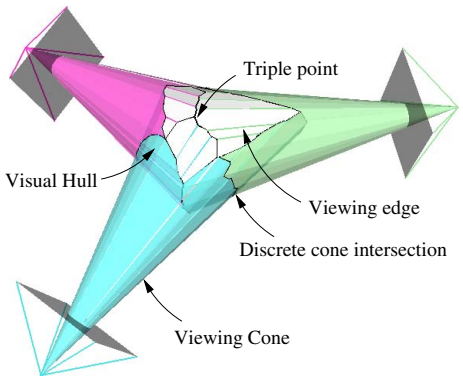

Figure 5: Visual hull of a sphere with 3 views.

ity method [Franco03] (overview available in figure 3). Figure 5 provides a representation of all geometric entities involved.

As seen previously, the viewing edges give us an initial subset of the visual hull geometry. However, this does not account for all edges of the visual hull polyhedron. The visual hull is the intersection of the *viewing cones*, which back-project from silhouettes in images. For a complete visual hull, one must also compute the missing cone intersection curves that participate to the visual hull polyhedron. It can be observed that such curves, which are piecewise-linear, snake around the visual hull surface, connecting together viewing edge vertices and extra vertices called *triple points*. Triple points are the locus of three cone intersections; as such they are always the meeting point of three cone intersection curves on the visual hull (see fig. 5).

With this in mind, the exact method simply seeks to follow these curves while creating them, starting from viewing edge vertices and recursing at cone intersection junctions, i.e. the triple points. The algorithm does so by iteratively computing new edges of the curve, using incidence information previously inferred. Checking for the silhouette consistency of each newly created edge ensures that it is part of the visual hull mesh; it also enables the algorithm to detect and create the missing triple points, when such consistency is violated. When all cone intersection edges are recovered, faces of the polyhedron surface are extracted by walking through the complete oriented mesh while always taking left turns at each vertex, so as to identify each face's 2D contours and complete the third step in the algorithm. Refer to [Franco03] for details.

Any parallelization effort for this algorithm will likely be confronted to the strong spatial dependencies inherent to a mesh representation. In order to allow for concurrent task execution, we classically choose to partition space into p different regions using $p-1$ parallel planes, thus subdividing space in p "slices". Slice width is adjusted by attributing a constant number of viewing edge vertices per slice for workload balancing. Since we mainly manipulate edges and vertices, partitioning of primitives among regions during the data preparation phase is a very low cost operation. Thus, a host of the distributed exact connectivity method can be instructed to follow intersection curves within its dedicated region $\mathcal{R}_i$, until this curve crosses the border toward another region $\mathcal{R}_j$. The host then stops processing this curve, thereby delegating the computation of the rest of the curve to the host in charge of $\mathcal{R}_j$ during the parallel computation phase.

Observe that region dependencies are very easy to identify as they only materialize at edges that cross a partition plane. It is yet again straightforward to identify the three simple phases of our frame-level parallel model in this case. Data preparation partitions the job among regions; parallel computation tasks compute mesh fragments associated to their dedicated region; the sequential computation phase gathers and carefully merges the partial meshes across region borders. This proves to be very efficient as we reach speed-ups of 6 with 10 hosts with our implementation, an excellent result given the reasonable implementation time and the dependency issues. This will be confirmed by the global measurements provided in the next section.

We are also able to distribute the surface extraction step: the complete mesh is broadcasted to p hosts during data preparation, then the p hosts independently compute a subset of the face information, and the sequential finalization simply gathers all sets of faces. This leads to very good speed-ups of the order of 7 for 10 hosts.

4. Implementation and Experimental Results

In this section, we detail experimental results obtained from the implementation of the two preceding algorithms paralellized with our methodology. We obtain real time performance for high quality 3D modeling; recall that for the second method the computed polyhedron is exact with respect to the input silhouettes. Tests with synthetic data show that sustained performance is obtained with a large number of view points.

Our 16 processor cluster is composed of 8 dual Xeon PCs (2.66 GHz) connected through a Gigabit Ethernet network. Latency is measured from the beginning of the viewing-edge step. Our implementation uses the standard MPI message passing library [Gropp94] for comunications. The Delaunay triangulation is computed with the high performance sequential library Qhull [Qhu]. All presented results are based on sets of 10 experiments.

4.1. Real Time Conditions

Our real experimental setup is composed of 4 IEEE 1394 cameras each connected to a PC handling acquisition, back-

ground substraction and silhouette vectorization. Images (640x480) are acquired at 30 frames per second. The scene is composed of a person (see fig. 6), an object of average complexity (150 contour vertices per image).

Figure 6: Real time reconstruction of a real person with 4 cameras and the exact connectivity approach.

Using such a system to run the hybrid method, we achieve real time 3D modeling at 30 frames per second using 16 processors. One processing unit parallelized on 4 hosts is dedicated to the first step. Ten processing units are dedicated to the sequential Delaunay triangulation. Carving is achieved in a single processing unit parallelized on 2 processors. The measured latency comes in the average of 400 ms, but is highly limited by the sequential execution time of the triangulation time, which can reach 300 ms.

The exact connectivity method proved to be more efficient as real time execution (30 frames per second) is achieved with only 12 processors. Each stage has 2 processing units, each one being parallelized on 2 processors. The measured latency is about 100 ms. This low latency and real time frame processing rate enable to use this algorithm for interactive applications. Videos are available at `http://www.inrialpes.fr/movi/people/Franco/CVPR04`.

4.2. Validation with Large Numbers of View Points

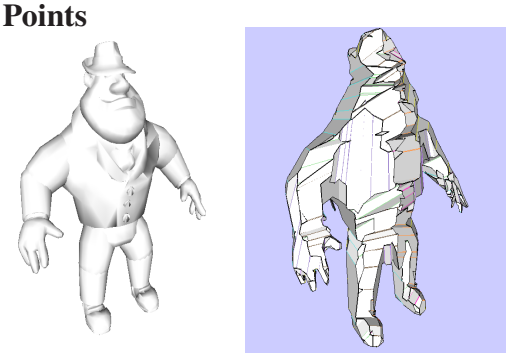

Figure 7: (left) Original model. (right) Reconstruction of the Model with 12 view points.

Not having more than 4 cameras available, the scalability of our distribued algorithms was tested with images from multiple view points generated from a synthetic model. We focus on the latency issue. We only consider the exact connectivity 3D modeling algorithm as the hybrid one is latency limited by the Delaunay triangulation. The real time frame rate issue is not discussed as it can be solved by multiplying the number of PCs assigned to the stream level parallelization.

The model we consider is a synthetic person with a complexity close to a real person (about 130 contour vertices per image). Figure 8 presents the obtained latency with regard to the number of processors involved for 16, 25 and 64 view points. The parallelization of the algorithm enables to significantly reduce the latency (almost by an order of magnitude). With 25 view points and 16 processors, latency is below 200 ms, a latency level suitable for interactivity.

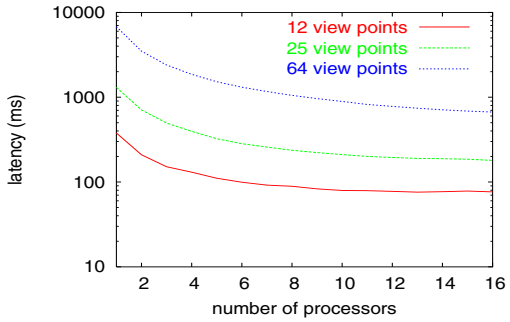

Figure 8: Log plot latencies for the synthetic person.

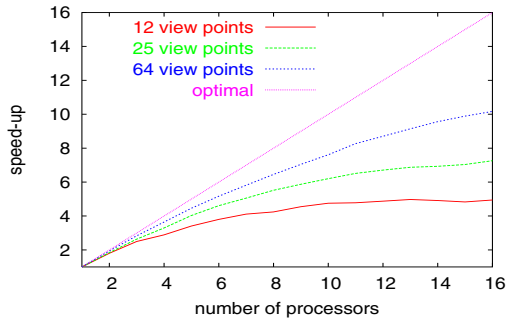

Figure 9: Speed-ups for the synthetic person.

Figure 9 presents the associated speed-ups. Up to 9 processors for 12 view points, 14 processors for 25 view points, and more than 16 processors for 64 view points, the speed-up is above half of the processors used. Next, the speed-ups tend to stabilize as the workload in the parallel computation phases decreases compared to the data preparation and sequential computation phases.

5. Summary and Conclusions

We have presented a 3D modeling system which uses parallelism to reach real time executions with a flexible number

of cameras and PCs. Such system is based on a distribution framework we propose, which is intended to multi-view applications in computer vision. We have demonstrated its effectiveness in 3D modeling applications using silhouettes. The high quality visual hulls generated by these parallel algorithms can be used for various applications, including virtual reality (see fig. 10). Our main contribution with respect to existing works in the field is to provide new parallel 3D modeling implementations as well as a methodology for the parallelization of multi-view applications. Results on real and synthetic data show that our approach allows for scalability in modeling systems and extends therefore the potential of such systems. We are currently studying generalization of the given principles to other computer vision applications. We are also extending our experimental setup so that it includes more than 20 cameras and provides a complete pipe-line from the image acquisition to the model visualization in multi-projector environments, with all the associated tasks distributed on a PC cluster.

Figure 10: Two new views of the visual hull model of figure 6 with view-dependent texturing.

References

[Arita01] D. Arita and R.-I. Taniguchi. RPV-II: A Stream-Based Real-Time Parallel Vision System and Its Application to Real-Time Volume Reconstruction. In *Computer Vision Systems, Second International Workshop, ICVS, Vancouver (Canada)*, 2001.

[Baumgart75] B.G. Baumgart. A polyhedron representation for computer vision. In *AFIPS National Computer Conference*, 1975.

[Borovikov03] E. Borovikov, A. Sussman, and L. Davis. A High Performance Multi-Perspective Vision Studio. In *17th Annual ACM International Conference on Supercomputing, San Francisco (USA)*, 2003.

[Boyer03] E. Boyer and J.-S. Franco. A Hybrid Approach for Computing Visual Hulls of Complex Objects. In *CVPR'2003, Madison (USA)*, volume I, pages 695–701, 2003.

[Cheung00] G. Cheung, T. Kanade, J.-Y. Bouguet, and M. Holler. A real time system for robust 3d voxel reconstruction of human motions. In *CVPR'2000, Hilton Head Island, (USA)*, volume 2, pages 714 – 720, June 2000.

[Cheung03] G. Cheung, S. Baker, and T. Kanade. Visual Hull Alignment and Refinement Across Time: A 3D Reconstruction Algorithm Combining Shape-From-Silhouette with Stereo. In *CVPR'2003, Madison (USA)*, 2003.

[Cignoni93] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D Delaunay Triangulation. *Computer Graphics Forum*, 12(3): 129–142, 1993.

[Dyer01] C.R. Dyer. Volumetric Scene Reconstruction from Multiple Views. In L.S. Davis, editor, *Foundations of Image Understanding*, pages 469–489. Kluwer, Boston, 2001.

[François01] A. François and G. Médioni. A Modular Software Architecture for Real Time Video Processing. In *Computer Vision Systems, Second International Workshop, ICVS, Vancouver (Canada)*, pages 35–49, 2001.

[Franco03] J.S. Franco and E. Boyer. Exact Polyhedral Visual Hulls. In *BMVC'2003, Norwich (UK)*, 2003.

[Gropp94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engeneering Computation Series. The MIT Press, 1994.

[Kameda00] Y. Kameda, T. Taoda, and M. Minoh. High Speed 3D Reconstruction by Spatio Temporal Division of Video Image Processing. *IEICE Transactions on Informations and Systems*, (7): 1422–1428, 2000.

[Laurentini94] A. Laurentini. The Visual Hull Concept for Silhouette-Based Image Understanding. *IEEE Transactions on PAMI*, 16(2): 150–162, February 1994.

[Li03] M. Li, M. Magnor, and H.-P. Seidel. Improved hardware-accelerated visual hull rendering. In *Vision, Modeling and Visualization Workshop, Munich, (Germany)*, 2003.

[Matusik01] W. Matusik, C. Buehler, and L. McMillan. Polyhedral Visual Hulls for Real-Time Rendering. In *Eurographics Workshop on Rendering*, 2001.

[Narayanan98] P.J. Narayanan, P.W. Rander, and T. Kanade. Constructing Virtual Wolrds Using Dense Stereo. In *ICCV'1998, Bombay, (India)*, pages 3–10, 1998.

[Qhu] Qhull, convex hull and delaunay triangulation library. http://www.geom.uiuc.edu/software/qhull/.

[Sérot02] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12): 1685–1708, 2002.

[Slabaugh01] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafe. A Survey of Methods for Volumetric Scene Reconstruction from Photographs. In *International Workshop on Volume Graphics*, 2001.

[Valiant90] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8): 103–111, August 1990.