UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Rendu interactif de larges terrains sur grappe de PC

Valérie Gouranton
Souley Madougou
Emmanuel Melin
Cyril Nortet
Université d'Orléans, LIFO

Rapport N^o **RR-2004-07**

Interactive rendering of massive terrains on PC cluster

V. Gouranton[1], S. Madougou[1,2], E. Melin[1] and C. Nortet[1]
[1]Laboratoire d'Informatique Fondamentale d'Orléans, LIFO Orléans, France
[2]BRGM, France

January 10, 2005

Abstract

We describe a parallel framework for interactive smooth rendering of massive terrains. We define a parallelization scheme for level of detail algorithms in cluster-based environments. The scheme relies on modern PC clusters capabilities to address the scalability issue of level of detail algorithms. To achieve this, we propose an efficient tile-based data partitioning method that allows both reducing load imbalance and solving the well-known border problem. At runtime level of detail computations are performed in parallel on cluster nodes. A hierarchical view frustum culling combined to a compression mechanism harnessing the frame-to-frame coherence are used to drastically reduce the inter-tasks communication overhead. We take into account level of detail algorithms visual quality issue by providing geomorphing and texturing supports. We are able to interactively and smoothly render terrains composed of hundreds of millions to billions of polygons on a cluster of 8 PCs.

Résumé

Nous décrivons un environnement parallèle pour le rendu interactif et continu de terrains très étendus. Nous définissons un schéma de parallélisation des algorithmes de gestion de niveau de détail pour les environnements à base de grappe de PC. Ce schéma se base sur la performance des grappes de PC récentes pour proposer une solution au problème de passage à l'échelle des algorithmes de gestion de niveau de détail. Pour accomplir ceci, nous proposons une méthode efficace de partitionnement de données à base de morceaux de terrain qui permet de réduire le déséquilbrage des charges et de résoudre le problème bien connu de simplification des frontières. Pendant l'exécution, les calculs de niveau de détail sont effectués en parallèle sur les noeuds de la grappe. Un culling hiérarchique combiné à un mécanisme de compression exploitant la cohérence temporelle est utilisé pour réduire, de manière drastique, le surcoût des communications inter-processus. Nous tenons en compte la qualité visuelle des algorithmes de gestion de niveau de détail en proposant un support pour le geomorphing et les textures. Nous sommes capables de rendre de manière interactive et sans à-coups des terrains composés de millions voire de milliards de polygones sur une grappe de 8 PC.

1 Introduction

These last two decades, scientific visualization becomes an attractive way for gaining deep understanding in scientific data. Interactive rendering of these data gives researchers the ability to do faster visual data analysis. Due to strides in data acquisition technologies (high-resolution 3D scanners, cameras, scientific simulation), these data can reach gigantic sizes. For instance, USGS elevation data for Washington state at a horizontal resolution of ten meter and a vertical resolution of ten centimeter represent 1,4 billion elevation values [29]. Visualizing this kind of data in an interactive way is not a trivial task [19]. As the data obviously do not fit in the system memory, direct rendering cannot be used even with high-end graphics dedicated infrastructures such as SGI machines [22]. These architectures generally provide memory and compute capabilities that can satisfy many visualization needs. However, their restricted scalability and extensibility makes them not appropriate for the rapid increase of today's datasets. An alternative way is to use level of detail (LOD) algorithms. Given a model to be visualized, their goal is to decrease the size of the data sent through the graphics pipeline by discarding the unnecessary detail while preserving an acceptable visual quality. Although they induce some CPU load overhead, LOD algorithms give the ability to balance between interactivity and detail for large models visualization. First LOD algorithms require the data to fit entirely in the system memory [21, 16, 10]. This naturally bounds their usage for huge datasets. In order to bypass this bottleneck, many desktop-based LOD algorithms resort to out-of-core mechanisms. This often leads to considerable CPU load and memory footprint overheads. To ensure interactivity by hiding the latency due to the external memory access, they must implement complex prefetching mechanisms. However, any extra time spent in LOD algorithm inexorably results into frame rate degradation, thus into interaction disruption.

For interactive rendering of huge datasets, both LOD techniques and high-performance computing are required. Moreover, the anatomy of super-computers is quickly and deeply changing. Clusters of commodity components are becoming the leading choice architecture. They are now usual in the supercomputer top 500 [27]. They are scalable and modular with a high performance-price ratio. These architectures are proved efficient for classical (not interactive) intensive computations. In scientific visualization field, several work [1, 14] showed the effectiveness of using PC cluster as a means of complex computations (simulation) while the rendering is performed in a multi-display environment. Our goal is to define a similar model for LOD algorithms aiming at massive terrain rendering. Distributing terrain data and LOD computations among cluster nodes avoids the two limiting factors, CPU time and memory. The number of nodes being theoretically unbounded, there is no reason to systematically resort to external memory solutions as for desktop-based algorithms. However, as in the out-of-core case, one must face the communications overhead induced by bus and network traffics. Currently, efficient inter-connecting networks exist for clusters. Their throughput now outperforms that of todays hard discs. Furthermore, in the particular case of graphics applications, in addition to the LOD algorithm which itself is a data reduction process, ways for further reducing the data transfer can be implemented by harnessing intrinsic graphics properties.

LOD algorithms usually require building a hierarchical structure that encompasses all levels of detail. As this preprocessing may be very time-consuming, the first parallelization attempts have been used to tackle this issue [12, 8]. Other approaches really parallelize LOD computations [3, 15]. In [3] a master/slaves paradigm is proposed. In this scheme, slaves perform the LOD computations and the resulting data are gathered on master for rendering. The master node becomes a bottleneck for the scalability of this approach. In a previous work [15], we present a cluster-based parallelization framework for height fields that does not suffer from this bottleneck. However, this work presents some weaknesses. No load balancing support is provided. It does not propose a satisfactory solution for the partition boundary simplification issue (the *border problem*) nor it benefits from frame-to-frame coherence. Moreover, there is no support for visual quality improvement such as geomorphing or texturing.

This paper describes a comprehensive parallelization framework built on the model described in [15]. Critical issues such as the border problem and load balancing are solved. Essential optimizations that benefit from graphics properties such as culling and frame-to-frame coherence are proposed to drastically reduce the communication overhead. In addition, techniques such as geomorphing and texturing, are used to improve the visual quality of the rendering. The paper is organized as follows: in section 2 we briefly recall our previous work. Section 3 describes the parallelization scheme in details. The way communication overhead is minimized is explained in 4. Geomorphing and texturing are described in 5. Algorithm performance is given through benchmarks in section 6. We conclude and plan for future work in section 7.

2 Background

In [15], we present a cluster-based parallel simplification framework for height fields (fig. 1). This framework partitions cluster nodes into visualization nodes and LOD nodes. The visualization nodes do only rendering whereas LOD nodes do only computations. At a preprocessing stage, initial height field data are partitioned into n partitions where n is the number of LOD nodes. During the execution, before each step of LOD computations, LOD nodes receive view parameters (view point and frusta) from visualization nodes. Then, they independently perform the LOD algorithm on their partition. Resulting data are culled against visualization nodes frusta before being sent for rendering. The way the initial data are partitioned does not allow to reach a satisfying load balancing. As partition is done on a large block per node basis, load balancing is strongly related to the user's interaction. Moreover, this work does not address the border problem because LOD computations are performed totally independently inside tiles whereas some overlapping data are needed. Indeed, [15] proposes a parallelization of Roettger's algorithm [24]. This algorithm dynamically constructs a hierarchical structure, called quadtree, that represents the most relevant data with respect to the current view

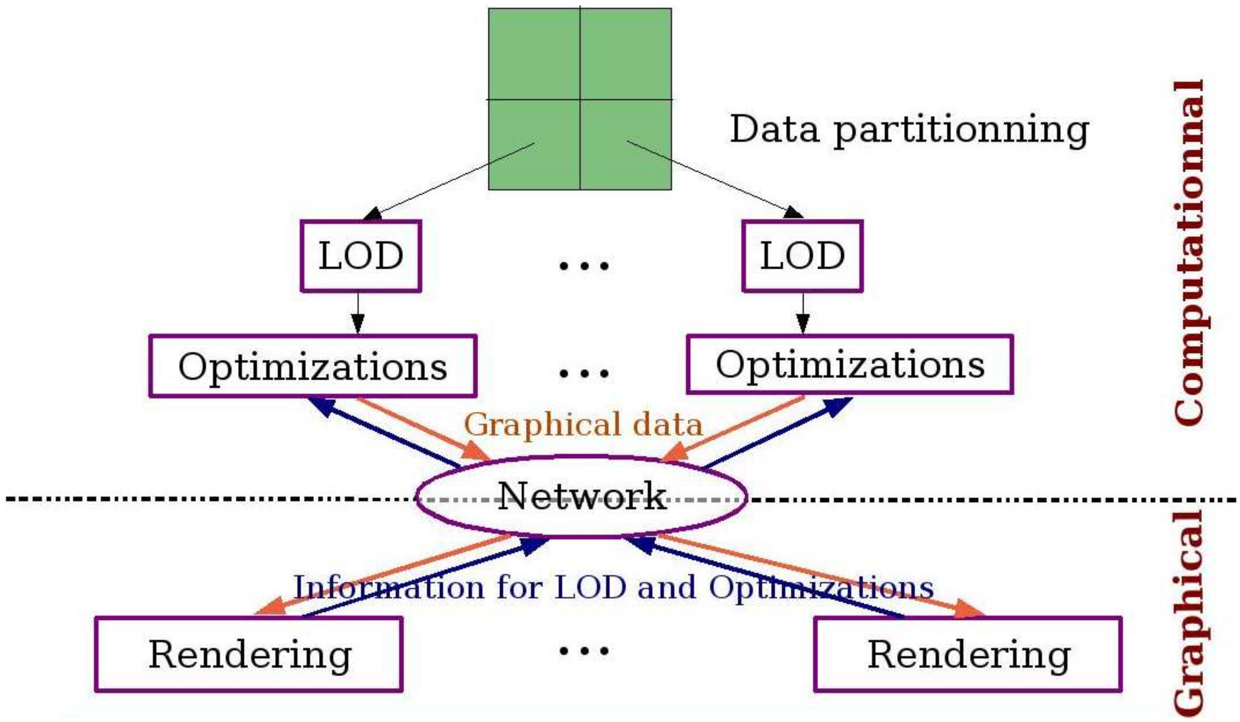

Figure 1: A parallelization model for LOD.

point and the local terrain topology. The subdivision process is controlled by a decision variable f evaluated at each quadtree node (for the sake of clarity, we will call it block). It depends on the block's current subdivision state as well as the local terrain topology. The local terrain topology is measured as the terrain roughness around the block (called its $d2$ *value*). Thus, the decision variable of a given block also depends on the adjacent blocks. This information is used to constrain the LOD difference between adjacent blocks to never exceed one, block's LOD value being its depth in the quadtree structure. We call this constraint the *fundamental constraint*. As long as this constraint is fulfilled, T-junctions, that cause cracks, are avoided by skipping drawing central vertices on shared edges between blocks of different LOD values (see figure 2). While using a sequential algorithm, we can easily satisfy this constraint as $d2$ values are computed in a consistent manner by taking into account neighbors ones. However, in a parallel framework, adjacent blocks may belong to different partitions. Thus, independently computing $d2$ values inside the partitions as done in [15], leads to inconsistencies in border blocks $d2$ values because a complete neighborhood information lacks for these blocks.

3 The Parallelization Scheme

As usual in parallel computing, our algorithm is made up of two phases: a data partitioning phase and a computations phase. Data partitioning is done statically in a preprocessing stage whereas computations step occurs at each frame and may lead to inter-tasks communications. The three key points in writing efficient parallel algorithms, namely inter-tasks communications, load balancing and global coherence, are taken into account. Inter-tasks communication is the topic of the next section. The global coherence issue, known as the border problem in the field of LOD, is discussed in subsection 3.2. The way data are partitioned takes care of load imbalance and it concerns the next subsection.

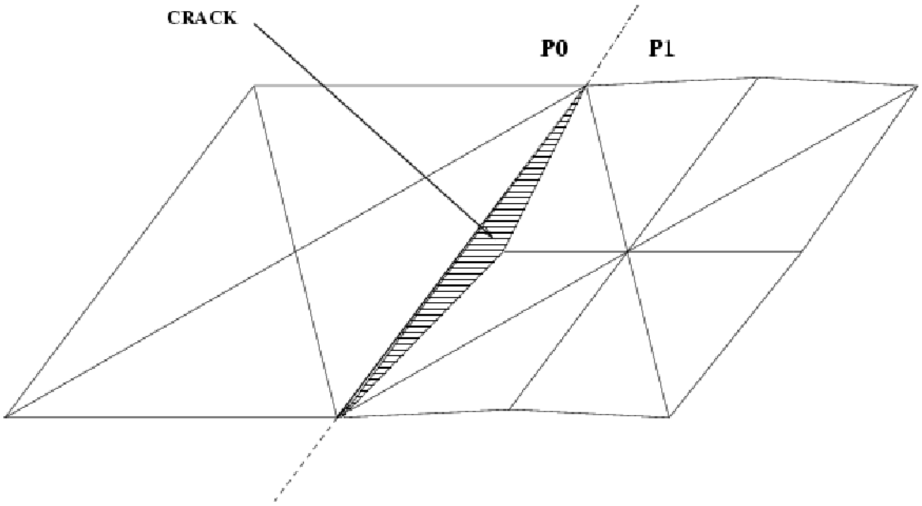

Figure 2: Lacking complete neighborhood information leads to cracks between adjacent blocks of different partitions.

3.1 Data Partitioning

Data distribution is a critical stage in parallel computing. The parallel algorithm efficiency heavily depends on it. The global task and data must be partitioned into subtasks and subsets that are ideally independent and balanced. Apart from embarrassingly parallel algorithms, these goals are difficult to reach. In the LOD field, partitions size must be carefully chosen: they must not be too small nor too large. In one side, if they are too large, the navigation can introduce load imbalance. On the other side, cutting them too smaller can limit the drastic simplification ability of the algorithm as the number of partitions determines its minimum achievable resolution. Let's assume our initial terrain is a square of side $(2^n + 1)$. On one hand, if terrain is not square, we extend its side to the power of 2 immediately greater than the actual side, the added data being skipped at drawing phase. Next, terrain data are partitioned into square *tiles* of side $(2^m + 1)$, where $m <= n$. Such tiles perfectly fit to Roettger's algorithm and can be processed on different machines. The number m is chosen such that load imbalance is minimized and the minimal resolution is acceptable. To achieve this, tiles are assigned to the computation nodes in a checkerboard manner. Thus, whatever the navigation is, each computations node has some work to do. Moreover, sending the simplified data to the visualization nodes is not done on a per-tile basis. Instead, all computations node data are gathered together before being sent, thus optimizing the network bandwidth use.

3.2 Addressing the Border Problem

Spatial decomposition is not a novel idea in the LOD field. Many out-of-core algorithms [5, 4, 18, 28, 11, 30] partition the initial data into tiles (some authors call them clusters, segments or chunks) to speed up external data access. A known issue related to this partitioning is the border problem. This problem occurs because LOD algorithms need to know about the neighborhood of a particular simplex in order to determine whether it has to be simplified, refined or left unchanged. However, when data are partitioned, this information lacks for border simplices inside the partitions. Several authors have been faced this issue. For instance, in [18], Hoppe describes one of the first out-of-core LOD algorithms which partitions initial terrain data into rectangular blocks. After simplification, adjacent blocks are stitched together in a hierarchical fashion to form larger blocks. To ensure a conforming stitching, blocks boundaries are left unchanged. While going up in the hierarchy, blocks boundaries are coarsen but at the price of an additional processing. Other authors simply hide the visual artifacts due to the border problem either by using vertical skirts to fill the cracks around the chunks [28] or by using texturing [2].

In fact, in order to solve the border problem, some information about neighboring simplices are needed for each border simplex [5]. There are several ways to do this. One way is to fetch the missing information at runtime. This could be costly, especially in a parallel computing context. So, we adopt an alternative way which consists in associating additional neighborhood information into partitions data at the preprocessing stage and then doing some computations to recover the missing information. As stated in section 2, in the LOD algorithm we used, when the fundamental constraint is fulfilled, the only information needed by a block about its neighbors is whether they are refined or not. This information is obtained by computing the decision variable f of those blocks. Thus, we must fulfill two conditions in order to guarantee a solution to the border problem:

1. satisfying the fundamental constraint across partitions boundaries
2. knowing foreign neighbor blocks subdivision state for each border block

To satisfy the first constraint, the $d2$ values must be computed as described in [24], but on the entire dataset and before the partitioning stage occurs. To satisfy the second condition for a given border block, we must be able to evaluate f for each adjacent block belonging to another tile. Except the $d2$ value, all parameters needed to evaluate the decision variable for adjacent blocks can be computed from the border block. So, in addition to its own d2 values, a tile must store the d2 values of neighboring tile blocks. These contiguous blocks follow a binary tree pattern, and as neighboring tiles number is at most four, the number of additional d2 values to record is $4 * \sum_{i=0}^{n-1} 2^i = 4 * (2^n - 1)$. This represents a few percent of the initial tile data (3% for n=7 for instance).

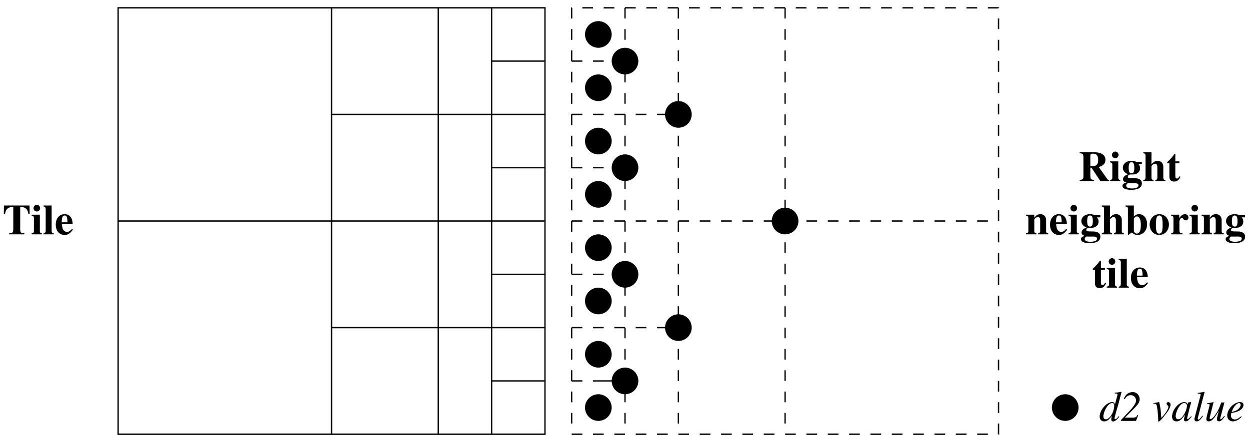

Figure 3: Binary tree of neighboring d2 values at the border of the tile right edge.

4 Minimizing communication overhead

Although they have evolved, networks still remain a bottleneck for data transfer-intensive applications on PC clusters. Therefore, sending all data at each frame from LOD nodes to visualization nodes would bound the model scalability. Instead, we propose a scheme which relies on culling and frame-to-frame coherence to considerably reduce the data transfer overhead by avoiding sending non visible and non modified data.

4.1 Hierarchical frustum culling

Large terrain visualization highly benefits from frustum culling since generally only a small part of the whole model is visible at a time. It is even more true in conjunction with a LOD algorithm because they are complementary: LOD is more effective for large views of the 3D object and frustum culling reaches its full potential at close ranges. In addition, in our scheme, all distribution features take advantage of this complementarity: faster LOD computations with early geometry pruning and lighter communication, both leading to an effective rendering. Frustum culling is as more efficient as it is performed early, so it is carried out in two stages : *coarse-grained* by not processing every tile fully outside the viewing frustum, and then *fine-grained* inside the LOD process when traversing each tile quadtree. Storing a complete hierarchy of bounding boxes over each tile quadtree would represent up to 66% of the original data. Actually a reasonable tradeoff is to limit the hierarchy to one or two level(s) below its maximum depth, the memory overhead is henceforth of 16% or 4% only whereas as the culling efficiency is still maintained.

4.2 Exploiting Temporal Coherence

Despite of LOD which continuously modifies the data visibility state, we observe that, the frame-to-frame coherence property still holds inside tiles. So, between two frames, large parts of the rendered data do not change even with interaction. On the basis of this observation, we implemented a mechanism allowing the modified data detection. It should be noted that this stage takes place after the culling step. Therefore, as it is applied to already reduced data, it does not induce a significant CPU overload. Furthermore, the introduced memory overhead is compensated by a substantial reduction of communication overhead . This mechanism is very similar to Unix commands *diff* and *patch*. We recall that after LOD algorithm is performed, data are sent to visualization node as triangle fans. For each tile, we record a copy of the list of fans being rendered which is the result of the previous LOD computations. At the end of current computations, a new list of fans is generated. A *diff* operation occurs between this list and the saved one. The delta is then sent to the visualization nodes which perform a *patch* operation between the incoming data and the previously rendered data. This operation results in updating the rendered data with respect to the current view parameters.

5 Rendering Quality

Thanks to advances in graphics technologies, graphics users become more demanding in interactivity and high visual quality even for huge models. LOD-specific visual issues are not tolerated anymore. Indeed, LOD obviously produces lower detail rendering in simplified regions. Moreover, as the view point moves, detail is suddenly added or removed. This leads to visual artifacts known as vertex popping. There are several ways to solve these issues. Terrain texturing can be used to maintain an acceptable detail even in simplified areas. Vertex popping is usually solved by using colors blending or by implementing a geometrical morphing technique (geomorphing) [17]. However, geomorphing is more widely used than blending. Instead of going from one level of detail to another in one step, geomorphing technique consists to smoothly morph vertex positions between two consecutive levels of detail. This smooth transition is gained at the cost of an additional processing. So, it naturally benefits from our parallel framework as the induced CPU workload is distributed on all computations nodes.

5.1 Geomorphing

Since its introduction by Fergusson et al. [13], many LOD algorithms have incoporated the morphing technique. Different approaches are used for its implementation. Distance-based approcaches use transition zones to blend the geometry between levels of detail [7, 23]. Time-based strategies use a fixed time interval as interpolating parameter [17, 10]. These approaches may induce important computational overhead because finding the appropriate interpolating parameter is not a trivial task [17]. Another approach consists in using the error metric used in the algorithm to parameterize the morphing process [24, 6] . What is common to all approaches is that they all may introduce cracks because of the need to early know the interpolation final positions before the subdivision that creates them occurs.

In [24], the morphing parameter (called the blending factor b) is computed using the decision variable f (the metric error). However, instead of computing the blending factor as specified in this paper, we use the formula $b = \frac{(1-f)}{(0.5+0.5*fcmin-f)}$ [26] ($fcmin$ being the minimum of the child blocks f values) that takes into account children states. This is essential in order to prevent a block having a child while still morphing. Moreover it allows more linearity and more smoothness to the morphing process.

Our parallel framework perfectly fits to this technique (see figure 4). However, due to the border problem, the cracks occurrence is even more important in our model than the sequential ones. But as the blending factor depends on *f* and that the global consistence issue regarding this parameter is solved with the border problem, our implementation is free of geomorphing cracks.

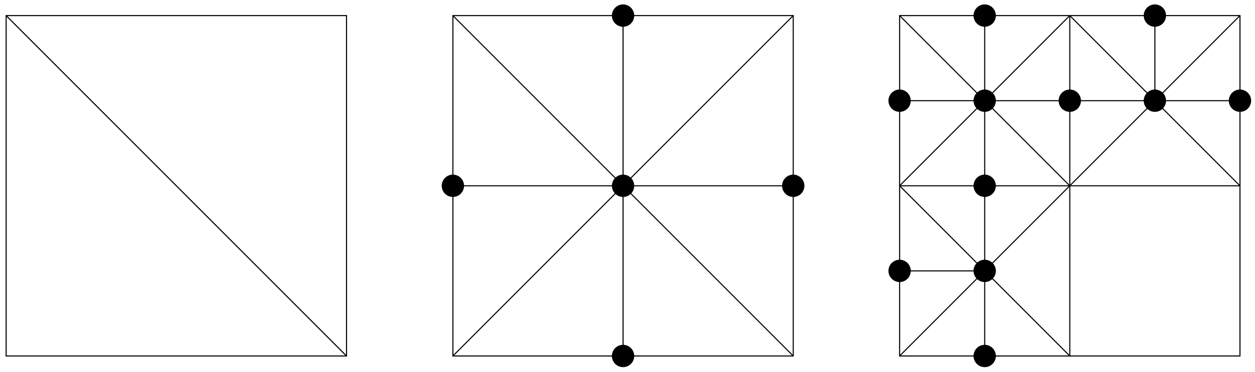

Figure 4: New vertices at each level of refinement. Only these vertices have to be geomorphed, while corner ones stay at the same 3D position.

5.2 Texturing

Texture data represent another important category of terrain data. In many applications, terrain data are associated with another layer of data such as a large satellite image, generally in the form of a texture handled by the graphics card [9, 2].

To fit the texturing to our tiling framework, we also adopt a tiled texturing approach [28]. At preprocessing stage, we compute a texture tile for each terrain tile using lightmapping and multi-texturing. These additional tiles are managed by the visualization nodes at rendering time.

Using lightmaps leads us to a static lighting based on the original terrain model. We found this technique more suitable to our framework than dynamic lighting that could require more graphics resources and lead to disturbing effects as we do not provide yet morphing mechanism for lighting attributes. In contrast, our approach avoids this drawback and it requires only one or two bytes per texel which can be compressed using advanced features of modern graphics cards. An alternative way would be to distribute lighting information on computations nodes, packed with height data. But lighting precision will follow the LOD simplification scheme, which leads to unexpected appearance in most simplified areas.

Texturing may also be used to manage missing values by exploiting alpha texturing. Missing data are present in many areas such as GIS. In this framework, we use them to fulfill the tile size constraint by setting to 'missing' all values beyond the real boundary of the tile.

Finally, large scenery textures may be difficult to handle as we do not provide distribution mechanism for texture data. However, since our framework frees visualization nodes from unnecessary geometry storing and processing, the whole graphics resources are available for addressing a maximal video memory, using AGP memory at full extent or even running out-of-core techniques such as clip-mapping [25].

6 Performance

We perform our tests on a cluster of 8 PCs equipped with Pentium 4 processors, 1 Gb of RAM and NVIDIA GeForce FX 5900 graphics cards with 256 Mb of video memory. The nodes are connected by a Gigabit Ethernet network. The same screen resolution of 1024x768 is used in all configurations. We use the publicly available USGS Washington state dataset [29] for all of our tests. These data represent 1.3 billion elevation values. The flythrough on the terrain is fixed in order to keep the interaction almost identical in all cases. As our framework is a tile-based system, we can load a variable number of tiles in order to change our dataset size, a tile being of size 257x257. With this setup which leads to a fine approximation of the model, an extended version of the whole USGS dataset representing 3.2 Gigabytes of data was successfully rendered in a 1 visualization node and 7 computation nodes setup, at an average framerate of 28. Each of the 7 computation nodes was loaded with 3600 tiles of 256x256 actual points, wich equals to a complete map of 1.65 billions 16 bits height values.

The purpose of our work is to increase the size of datasets that can be rendered using level of detail and commodity clusters. So, our first test naturally consists in checking the algorithm data scalability. For this purpose, we start with a dataset composed of 3600 tiles. For rendering this dataset, we use one machine for the computations and another for the visualization. Next, we render the double by adding a second computation node with another 3600 tiles and so on. As shown in the figure 5, the average frame rate is approximatively constant which corresponds to our expectations. The bending of this curve is due to the graphics board which becomes more loaded as the dataset size increases. Given this observation, we would expect to speed up the processing of a fixed-size model by adding more computations nodes. We check this over by running a test consisting in rendering a dataset made up of 3600 tiles using different setups, all using one visualization node. In the first setup, we use only one computations node, so that it processes all the 3600 tiles. The second setup uses 2 computations nodes, each one dealing with 1800 tiles. The others setups follow the same scheme. Results being drawn in figure 5, we can observe that the average frame rate increases as the number of computation nodes augments.

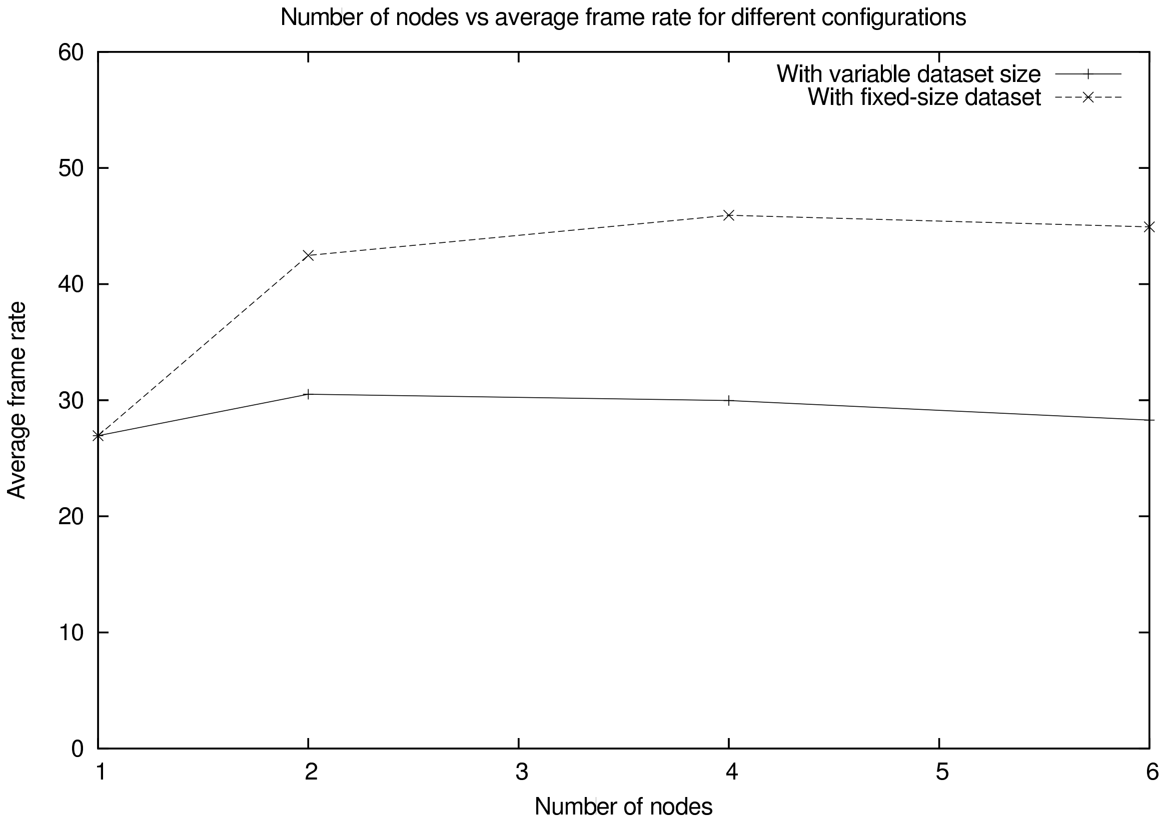

Figure 5: Data scalability test. We observe that the frame rate increases with the number of computation nodes (the top curve), it remains constant by multiplying the dataset size and the number of computation nodes by the same factor.

Tests on the data compression are very conclusive while not activating geomorphing. 80% to 90% of the data remain unchanged on the visualization nodes. However, when activating geomorphing this ratio drops to 50-60%

which is not surprisingly as geomorphing introduces more changing vertices (see figure 6). Using our framework, we distribute these additional computations on the cluster nodes. As shown in figure 6, the more they are, the lighter the extra cost.

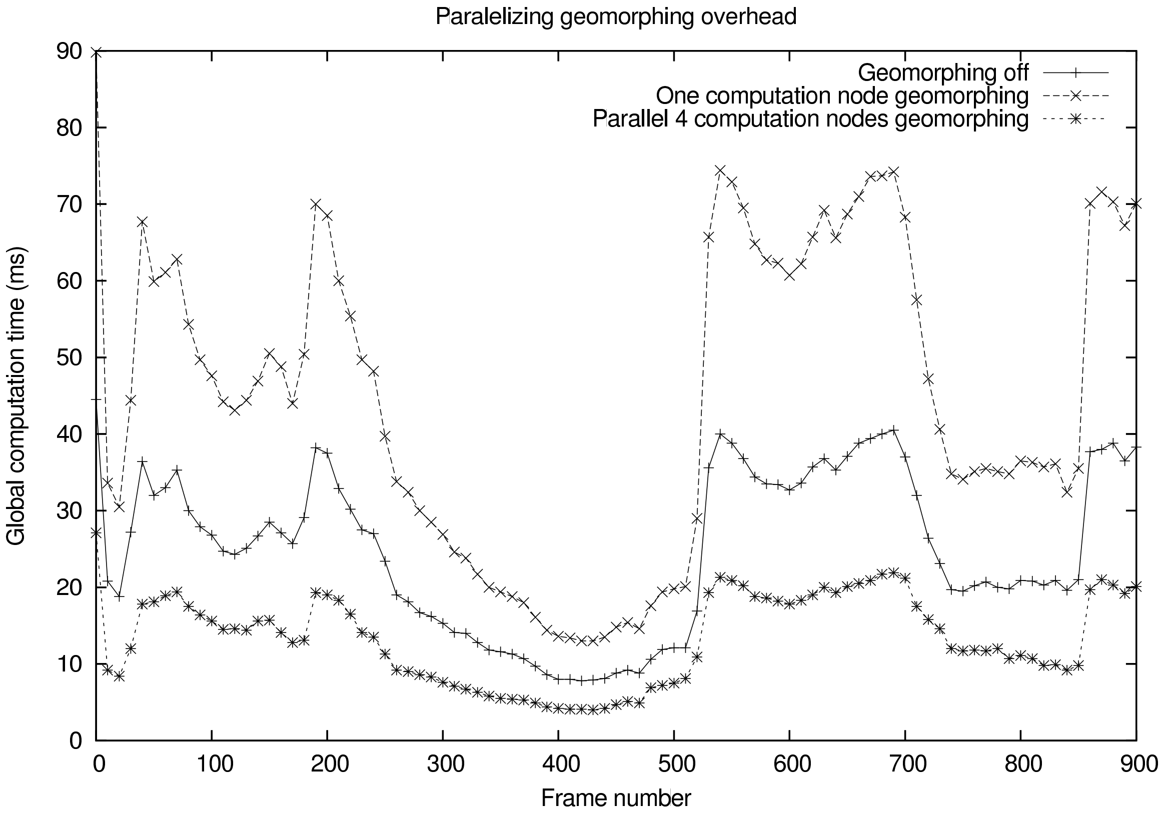

Figure 6: Implemeting geomorphing introduces important computation overhead. Observe the distance between the 2 top curves. Thes curves represent the global computation time when geomorphing is on (curve on the bottom) an off (curve on the top).

Finally, we do not use texturing for the tests (see figure 7). But as they induce no substantial overhead on the rendering, using them do not alter the given results.

7 Conclusion

We presented a parallel framework for interactive smooth rendering of massive terrains. The framework based on the model presented in [15], gives several improvements on this previous work. The main contributions concern addressing unsolved issues in [15] by solving the border problem and reducing load imbalance, adding more optimizations to further reduce the communication overhead by providing a hierarchical view frustum culling and a data compression mechanisms. Moreover, geomorphing and texturing features are given to improve the rendering quality. As shown by our benchmarks, this allows our framework to support smooth interactive rendering of very huge terrains.

However, texture data bound our parallel scheme because they are centralized on the visualization nodes. A solution will be to distribute those data among computation nodes which process them and then transfer the resulting color values to the rendering nodes. Furthermore, terrain data may be described as 'triangle soup', our framework is

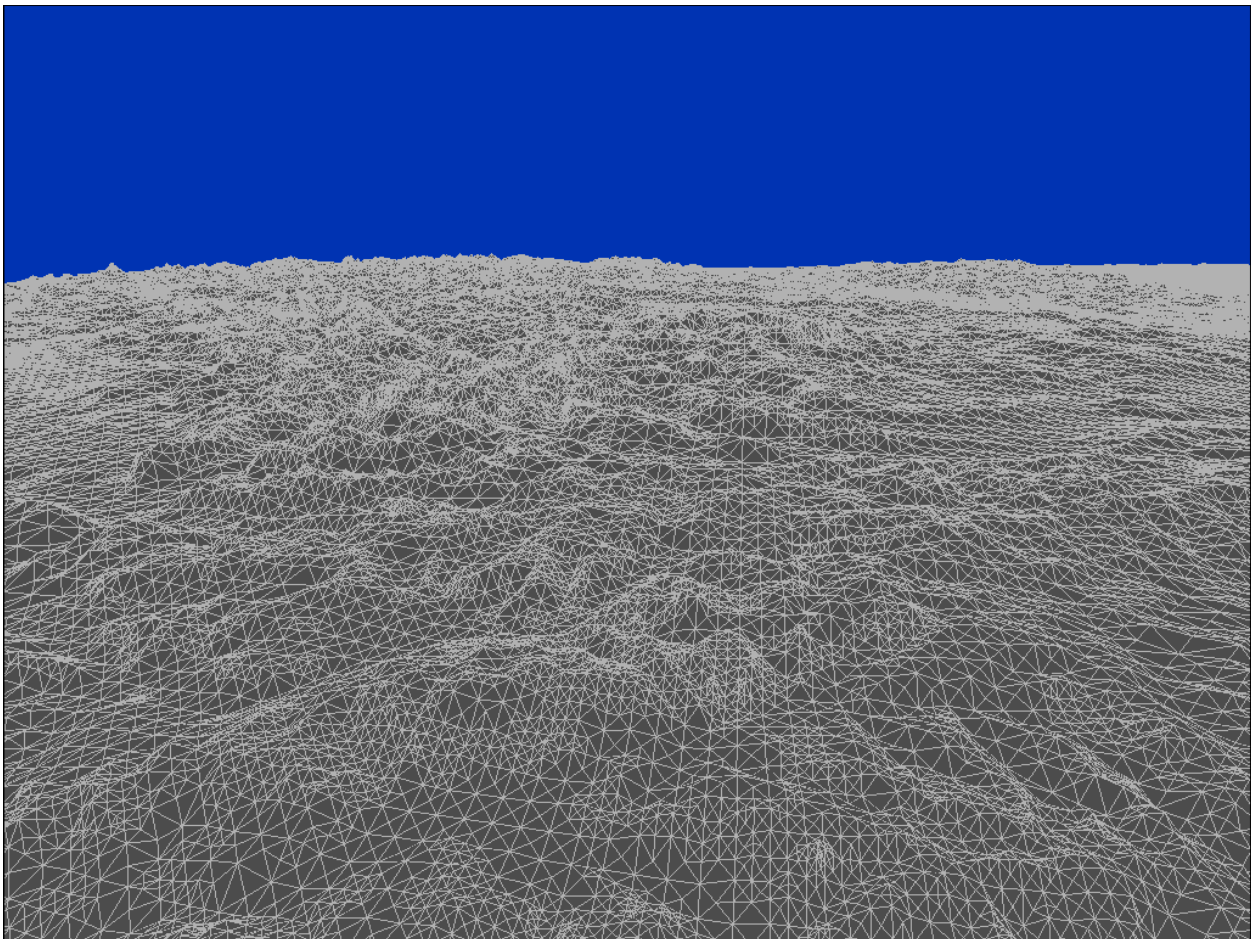

Figure 7: Our parallel framework running on USGS Washington dataset.

then not usable. In order to extend its applicability to other types of dataset such as triangulated irregular networks (TIN) or time-varying data, we aim at using scattered data approximation techniques [20] to adapt such models to our framework. In addtion, to speed up the local computations, the high stream processing capabilities of recent graphics processors can be exploited.

Acknowledgments

We would like to acknowledge the BRGM (French Geological survey) and Région Centre for supporting part of this work. We also would like to acknowledge Sylvain Jubertie for setting up the environment for our tests.

References

[1] J. Allard, B. Raffin, and F. Zara. Coupling parallel simulation and multi-display visualization on a pc cluster. In *Euro-par 2003*, Klagenfurt, Austria, August 2003.

[2] K. Baumann, J. Dollner, and K. Hinrichs. Integrated multiresolution geometry and texture models for terrain visualization. *Proceedings joint EuroGraphics-IEEE TCVG 2000*, May 2000.

[3] D. Brodsky and B. Watson. Model simplification through refinement. *Graphics Interface'00*, 2000.

[4] P. Cignoni, F. Ganovelli, E. Gobetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam: Batched dynamic adaptive meshes for high performance terrain visualization. *Proceedings EG2003*, pages 505–514, September 2003.

[5] P. Cignoni, C. Rocchini, C. Montani, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transaction on Visualization and Computer Graphics*, 2002.

[6] D. Cline and P. Egbert. Terrain decimation through quadtree morphing. *IEEE Transaction on Visualization and Computer Graphics 2001*, 7(1):62–69, January 2001.

[7] D. Cohen-Or and Y. Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. *IEEE Visualization '96*, pages 37–42, October 1996.

[8] F. Dehne, C. Langis, and G. Roth. Mesh simplification in parallel. *ICA3PP'00*, pages 281–290, 2000.

[9] J. Dollner, K. Baumann, and K. Hinrichs. Texturing techniques for terrain visualization. *Proceedings IEEE Visualization'00*, pages 227–234, 2000.

[10] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Mille, C Aldrich, and M. B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes. *IEEE Visualization*, pages 81–88, 1997.

[11] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 3(19):139–150, August 2000.

[12] J. El-Sana and A. Varshney. Parallel processing for view-dependent polygonal virtual environments. *Proceedings SIGGRAPH'99*, 1999.

[13] R. Fergusson, R. Economy, and P. Kelly, A.and Ramos. Continuous terrain level of detail for visual simulation. *ACM Symposium on Interactive 3D Graphics*, pages 111–120, March 2001.

[14] R. Gaugne, S. Jubertie, and S. Robert. Distributed multigrid algorithms for interactive scientific simulations on clusters. *ICAT 2003, Japan*, 2003.

[15] V. Gouranton, S. Limet, S. Madougou, and E. Melin. a scalable cluster-based parallel simplification framework for height fileds. *EuroGraphics/ACM SIGGRAPH, Proceedings Parallel Graphics and Visualization'04*, pages 59–65, June 2004.

[16] H. Hoppe. Progressive meshes. *In proceedings SIGGRAPH'96*, pages 99–108, 1996.

[17] H. Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics (In proceedings SIGGRAPH'97)*, pages 189–198, August 1997.

[18] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE Visualization '98*, 31:35–42, October 1998.

[19] L. Hwa, M. Duchaineau, and K. Joy. Adaptive 4-8 texture hierarchies. *IEEE Visualization '04*, pages 219–226, October 2004.

[20] S. Lee, G. Wolberg, and Y. Shin. Scattered data interpolation with multilevel b-splines. *IEEE Transaction on Visualization and Computer Graphics*, 3(3), July-September 1997.

[21] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Real-time, continuous level of detail rendering of height fields. *Computer Graphics, Proceedings SIGGRAPH'96*, pages 109–118, 1996.

[22] J.S. Montrym, D.R. Baum, D.L. Dignam, and C.J. Migdal. InfiniteReality: A Real-Time Graphics System. In *Computer Graphics (SIGGRAPH 97)*, pages 293–303. ACM Press, August 1997.

[23] R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. *IEEE Visualization (Proc. IEEE Visualization '98)*, pages 19–26, 1998.

[24] S. Rottger, W. Heidrich, and P. Slussallek. Real-time generation of continuous levels of detail for height fields. *Proceedings in 6th International Conference in Central Europe on Computer Graphics and Visualization*, pages 315–322, 1998.

[25] C. Tanner, C. Migdal, and M. Jones. The clipmap: A virtual mipmap. *SIGGRAPH'98 proceedings*, pages 151–158, 1998.

[26] http://home.planet.nl/ monstrous/.

[27] http://www.top500.org.

[28] T. Ulrich. Rendering massive terrains using chunked level of detail control. *SIGGRAPH Course Notes*, 2002.

[29] http://rocky.ess.washington.edu/data/raster/tenmeter/onebytwo10/index.html.

[30] S. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: Interactive view-dependent rendering of massive models. *IEEE Visualization 2004*, pages 131–138, October 2004.