Processing Large Geometric Datasets in Distributed Environments

Daniela Cabiddu(✉) and Marco Attene

CNR-IMATI, Genova, Italy
{daniela.cabiddu,marco.attene}@ge.imati.cnr.it

Abstract. We describe an innovative Web-based platform to remotely perform complex geometry processing on large triangle meshes. A graphical user interface allows combining available algorithms to build complex pipelines that may also include conditional tasks and loops. The execution is managed by a central engine that delegates the computation to a distributed network of servers and handles the data transmission. The overall amount of data that is flowed through the net is kept within reasonable bounds thanks to an innovative mesh transfer protocol. A novel distributed divide-and-conquer approach enables parallel processing by partitioning the dataset into subparts to be delivered and handled by dedicated servers. Our approach can be used to process an arbitrarily large mesh represented either as a single large file or as a collection of files possibly stored on geographically scattered servers. To prove its effectiveness, we exploited our platform to implement a distributed simplification algorithm which exhibits a significant flexibility, scalability and speed.

Keywords: Distributed environments · Parallel computation · Geometry processing · Large meshes · Out-of-core

1 Introduction

In life science areas, several applications exist that allow remotely processing input data [38,40]. Such applications exploit the computational power of geographically scattered servers that communicate through traditional Internet connection. Each server exposes one or more remote services that can be invoked sequentially or in parallel to process a dataset received as an input.

This approach is scarcely considered in geometry processing literature where input datasets are easily made of millions of geometric elements and files encoding them may be larger than hundreds of gigabytes. Transferring these extremely large datasets on a distributed environment would slow the process down too much [6]. For this reason, geometry processing is usually performed by exploiting stand-alone tools and applications locally installed. When the main memory available on the local machine is not sufficient to load the input, divide and conquer approaches are used to subdivide the input into subparts, each of them sufficiently small to be processed [35]. Sometimes, multi-core technologies (e.g. GPUs) are exploited to process different subparts of the input simultaneously. Nevertheless, the memory shared among the concurrent processes imposes a sequentialization of I/O operations in any case.

Herewith, a Web-based platform is described to remotely run geometry processing workflows. The computational power of geographically distributed servers (i.e. processing nodes) is exploited to perform the actual computation. Our contribution is twofold: first, an optimized mesh transfer protocol is described that reduces the amount of data sent through the network and avoids possible bottlenecks; second, a divide-and-conquer approach is proposed that enables the possibility to run distributed parallel algorithms and guarantees efficiency. As a proof–of–concept, an innovative distributed mesh simplification algorithm is described that exploits our divide-and-conquer approach to distribute the computational load across multiple servers.

For the sake of simplicity, in the first part of the paper we assume that the input is stored as a single file on the disk of one of the servers. However, specific applications [4,17] acquire data from the real world and generate 3D models as collections of files, each representing a subpart of the whole. When such a collection is too large, it can be distributed on multiple machines. Although these datasets are natively partitioned, such a partition may not be compatible with the hardware limitations of the available processing nodes. In these cases, an input re-partitioning is required. In the second part of the paper (Sect. 6) we propose a novel approach to enable the possibility to run distributed parallel algorithms even on these extremely large data sets.

Summarizing, we propose an innovative approach to process arbitrary large geometric datasets. Thanks to our optimized transfer protocol and our divide-and-conquer method, well-known geometry processing workflows can be run efficiently on large datasets. To test our methods, a set of in-house Web services have been deployed on our servers and exploited. Each Web service is able to run a different geometric algorithm. Then, a former experimental phase has been focus on evaluate our mesh transfer protocol, while a second experimental phase has been focus on evaluating the distributed divide-and-conquer approach. Both experiments have been run a an heterogeneous dataset composed of meshes coming from public repositories [1,3] and from different research projects on processing large geometric datasets [2,4]. Both the computational time and the quality of the output meshes have been considered as a matter of comparison with the existing approaches. Our results demonstrate that distributed technologies can be actually exploited to efficiently run geometry processing even on extremely large datasets.

2 Related Work

Polygon meshes are the standard de-facto representation for 3D objects. A polygon mesh is a collection of polygons or “faces”, that form the surface of the object. To describe a mesh, both geometric and topological information are required. The former includes the position of all the vertices, while the latter describes which vertices are connected to form edges and faces (i.e. triangles). While processing a mesh, either the geometry or the topology (or both) may be involved. Due to this complex structure, distributively processing meshes is a non-trivial task.

In the reminder, we focus on *triangle meshes*. This specific representation is used to describe objects coming from diverse industrial and research areas (e.g. design, geology, archaeology, medicine and entertainment).

2.1 Mesh Processing

Traditionally, mesh processing is performed by exploiting existing tools and applications that need to be installed on the local machine. Among them, MeshLab [11] and OpenFlipper [29] allow editing a mesh, saving the sequential list of executed operations and locally re-executing the workflow from their user interfaces. Pipelines can be shared in order to be rerun on different machines where the stand-alone applications need to be installed.

Campen and colleagues published WebBSP [8], an online service which allows to remotely run a few specific geometric operations. The user is required to select a single geometric algorithm from a set of available operations and upload an input mesh. Then, the algorithm is actually run on the server and a link to the output is sent to the user. Unfortunately, only a single operation can be run at each call and the service is accessible only from the WebBSP interface.

Geometric Web services were previously considered by Pitikakis [31] with the objective of defining semantic requirements to guarantee their interoperability. Though in Pitikakis's work Web services are stacked into hardcoded sequences, users are not allowed to dynamically construct workflows, and possible bottlenecks due to the transmission of large models are not dealt with.

Distributed parallelism has been exploited in [28,30] to provide both analysis and visualization tools. The possibility to exploit distributed parallelism for processing has been proposed in [32] but, due to the use of a distributed shared memory, the approach proposed is appropriate only on high-end clusters where local nodes are interconnected with particularly fast protocols.

2.2 Processing Large Polygon Meshes

Out-of-core approaches assume that the input does not need to be entirely loaded into main memory, and the computation operates on the loaded portion at each time [14,23,24,41]. Similarly, the external memory data structure proposed in [10] provides support for generic processing under the constraint of limited core memory. These methods are very elegant, but pre-processing operations required to pre-sort the input and generate the data structures require a significant time. Also, they are based on the idea of repeatedly loading parts of the input; thus, they are not suitable for distributed environments.

To speed up the computation, parallel approaches are often exploited [5,15,18,36]. Typically, a "master" processor partitions the input mesh and distributes the portions across different "slave" processors that perform the partial computations simultaneously. When all the portions are ready, the master merges the results together. The many slave processors available in modern GPU-based architectures are exploited in [34], while multi-core CPUs are exploited in [37]. Both methods are based on a memory shared among parallel processes to allow efficient communication. Distributed architectures are not provided with shared memory and, thus, different approaches are required to allow parallel processes to efficiently communicate. In [32], a hybrid architecture is described, where both shared and distributed memory are exploited. Parallel algorithms involving significant communication among processes can be implemented, but the communication costs will eventually limit the scaling.

Other effective out-of-core partitioning techniques are described in [25,26]. These methods typically require their input to come as a so-called "triangle soup", where the vertex coordinates are explicitly encoded for each single triangle. Since this representation is highly redundant, the most diffused formats (e.g. OFF, PLY, OBJ, ...) use a form of indexing, where vertex coordinates are encoded only once and each triangle refer to them through indexes. When the input is represented using an indexed format, it must be dereferenced using out-of-core techniques [9], but this additional step is time-consuming and requires significant storage resources. As an exception, the method proposed in [33] is able to work with indexed representations by relying on memory-mapped I/O managed by the operating system; however, if the face set is described without locality in the file, the same information is repeatedly read from disk and thrashing is likely to occur.

When the partial computations are comprehensively small enough to fit in memory, incore methods are exploited to merge the final result. To guarantee an exact contact among adjacent regions, slave processors are often required to keep the submesh boundary unchanged [36]. If necessary and if the final output is small enough, the quality of the generated mesh is enhanced by exploiting traditional incore algorithms in a final post-processing step. Differently, the external memory data structure [10] allows keeping the boundary consistent at each iteration. Depending on the specific type of geometric algorithm, different approaches may be exploited to guarantee boundary coherence. Vertex clustering is just an example used in mesh simplification [25]. Such a method has a cost in terms of output quality, when compared with more "adaptive" methods: the clustering distributes vertices uniformly on the surface, regardless the local morphology, hence tiny features are not guaranteed to be preserved.

3 The Web-Based Platform

The framework architecture is organized in three layers [21]: a graphical user interface that allows building new workflows from scratch, and uploading and invoking existing workflows; a set of Web services that wrap geometry processing tools; a workflow engine that handles the runtime execution by orchestrating the available Web services.

The Graphical User Interface. The graphical interface allows building geometric workflows and remotely running them on a selected input model. While building a new workflow, the user is asked to provide the list of geometry processing algorithms that constitute the pipeline, each to be selected from a list of available ones. Also, conditional tasks or loops can be defined. Once the whole procedure is ready, the user can turn it into an actual experiment by uploading an input mesh. If no input is associated, the workflow can be stored on the system as an "abstract" procedure that can be selected later for execution.

The Web Services. A Web service can be considered as a black box able to perform a specific operation on the mesh without the need of user interaction. A single server (i.e. a provider) can expose a plurality of Web services, each implementing a specific algorithm and identified by its own address. The system supports the invocation of two types of Web services, namely "atomic" and "boolean". An atomic service runs a simple operation on a mesh using possible input parameters, and produces another mesh as an output. Conversely, a boolean service just analyzes the mesh and returns a true/false value. Boolean Web services are used to support the execution of conditional tasks and loops.

Since input models may be stored on remote servers, we require that Web services are designed to receive the address of the input mesh and to download it locally; also, after the execution of the algorithm, the output must be made accessible through another address to be returned to the calling service.

The Workflow Engine. The workflow engine is the core of the system and orchestrates the invocation of the various algorithms involved. From the user interface it receives the specification of a geometry processing workflow and the address of an input mesh. The engine analyses the workflow, locates the most appropriate servers hosting the involved Web services, and sequentially invokes the various algorithms. For each operation, such a list of registered Web services is queried to retrieve which ones can perform the task, and the best performing one is selected [13] based on appropriate metadata to be provided upon registration of the service on our system. When the selected Web service is triggered for execution, it receives from the engine the address of the input mesh and possible parameters, runs its task and returns the address of the generated output to the engine. This latter information is sent to the next involved Web service as an input mesh or returned to the user interface when the workflow execution terminates.

4 Mesh Transfer Protocol

Not surprisingly, we have observed that the transfer of large-size meshes from a server to another according to the aforementioned protocol constitutes a bottleneck in the workflow execution, in particular when slow connections are involved.

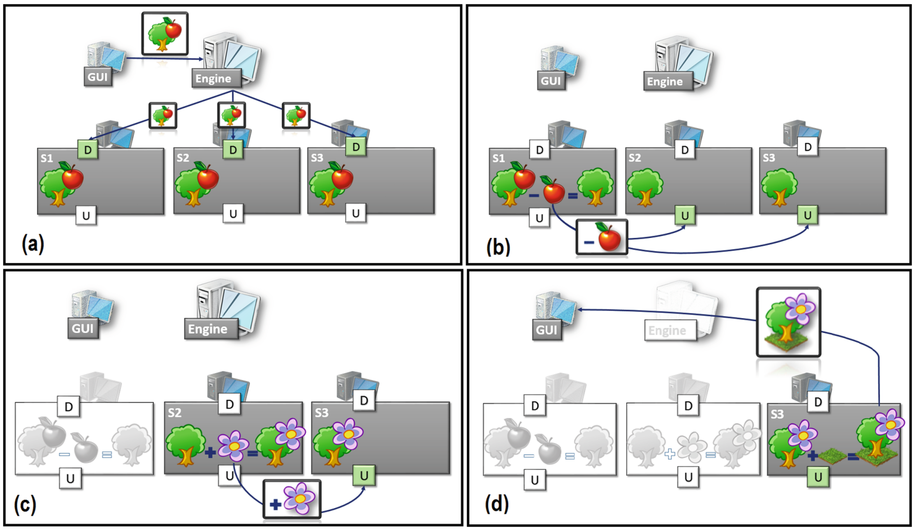

Fig. 1. Mesh transfer protocol example. The workflow is built by combining three operations. Thus, three servers are involved into the workflow execution. Each of them is able to download (D) meshes and update (U) the previously downloaded mesh by applying the corrections. (a) The address of the input mesh is broadcasted to all the involved servers that proceed with the download. (b) The first operation is run by the appropriate service that produces the corrections and returns the corresponding address to the engine. Such an address is shared in parallel to the successive servers, so that they can download the file and correct the prediction. (c) The second service runs the task and makes the correction available to allow the third server to update its local copy. (d) The last service is invoked to run the algorithm. The address of its output mesh is returned so that the user can directly download it.

Mesh compression techniques can be used to reduce the input size, but they do not solve the intrinsic problem [27]. In order to improve the transfer speed and thus efficiently support the processing of large meshes, we designed a mesh transfer protocol inspired on the prediction/correction metaphor used in data compression [39].

We have observed that there are numerous mesh processing algorithms that simply transform an input mesh into an output by computing and applying geometrical modifications. In all these cases it is possible to predict the result by assuming that it will be identical to the input, and it is reasonable to expect that the corrections to be transmitted can be more compactly encoded than the explicit result of the process.

The aforementioned observation can be exploited in our setting as shown in Fig. 1, where an example of execution of a simple workflow composed by three tasks is shown. Through the user interface, the user selects/sends a workflow and possibly the address of an input mesh to the workflow engine. The engine reads the workflow, searches for the available Web services, and sends in parallel to each of them the address of the input mesh. Each server is triggered to download the input model and save it locally. At the first step of the experiment, the workflow engine triggers the suitable Web service that runs the algorithm, produces the result, and locally stores the output mesh and the correction file (both compressed). Their addresses are returned to the workflow engine that forwards them to all the subsequent servers involved in the workflow. Each server downloads the correction and updates the local copy of the model according to it. Then, the workflow engine triggers the next service for which an up-to-date copy of the mesh is readily available on its local server. At the end of the workflow execution, the engine receives the address of the output produced by the last invoked Web service and returns it to the user interface, so that the user can download it.

In this scenario, the address of the input mesh is broadcasted to all the involved Web service once and Web services are able to download such a mesh simultaneously. Then, only correction files (which are sensibly smaller than the input mesh) travel through the network to allow each server to update its local copy of the mesh. In any case, each Web service produces both the correction and the actual result. When the correction is actually smaller than the results, this procedure produces significant benefits. Otherwise, the subsequent Web services can directly download the output instead of the corrections and no degradation is introduced. Note that lossless arithmetic coding is exploited by each Web service to compress either the output mesh or the correction file before making them travel the network.

5 Parallel Processing

Although our system theoretically allows processing any input mesh, remote servers have their own limitations and may not satisfy specific hardware requirements (eg. insufficient storage space, RAM, or computational performance) necessary to efficiently process large data. As a consequence, the remote server that is invoked may require a very long time to finish its task or, even worse, the process may be interrupted because of the insufficient main memory. In order to avoid these situations, the workflow engine is responsible for partitioning the input mesh into smaller subparts that can be elaborated by available processing services. When all the submeshes have been processed, they need to be merged to generate the final output. Both partitioning and merging operations are performed through out-of-core approaches. To allow final merging, an exact contact among adjacent regions must be guaranteed. Contrary to previous methods [36], our approach allows boundary modifications, while keeping the boundary consistent step by step.

For the sake of simplicity, our exposition assumes that all the servers have an equally-sized memory and comparable speed. Also, in the reminder of this section we describe the case where the input mesh is stored as a single input file. The treatment of pre-partitioned meshes whose parts are stored on different servers is described in Sect. 6.

5.1 Mesh Partitioning

We assume that the input mesh is encoded as an indexed mesh, since the most common file formats are based on this representation. Our mesh partitioning approach is mainly composed by the following sequential steps:

1. **Pre-Processing:** an initial binary space partition (BSP) is computed based on a representative vertex downsample;
2. **Vertex and Triangle Classification:** each vertex is assigned to the cell of the BSP where it falls, while each triangle is assigned to a selected BSP cell, based on the location of its vertices;
3. **Generation of independent sets:** each independent set includes submeshes that do not share any vertex, and thus they can be processed simultaneously;
4. **Optional post-processing:** depending on the specific geometry processing operation to be run by processing service.

Pre-processing. The mesh bounding box is computed by reading once the coordinates of all the input vertices. At the same time, a representative vertex down-sampling is computed and saved into main memory. Starting from the bounding box, an in-core binary space partition (BSP) is built by iteratively subdividing the cell with the greatest number of points. The root of the BSP refers to the whole downsampling. Each cell is split along its largest side. For each subdivision, each vertex in the parent cell is assigned to one of the two children according to its spatial location. If the vertex falls exactly on the splitting plane, it is assigned to the cell having the lowest barycenter in lexicographical order. The process is stopped when the number of vertices assigned to each BSP cell is at most equal to a given threshold, based on the number of vertices that each processing service is able to manage and the ratio between the number of input vertices and the downsample size.

Vertex and Triangle Classification. First, vertices are read one by one and assigned based on their spatial location as above. Some technical details are shown in Fig. 2.

Then, triangles are read one by one from T and assigned depending on their vertex position as follows:

- If at least two vertices belong to cell C_A, the triangle is assigned to cell C_A. In this case, if the third vertex belongs to a different cell C_B, a copy of the third vertex is added to C_A.
- If the three vertices belong to three different cells C_A, C_B, and C_C, the triangle is assigned to the cell having the smallest barycenter in lexicographical order (let it be C_A), and a copy of each vertex belonging to the other two cells is added to C_A.

At the end of the triangle classification, the BSP leaf cells represent a triangle-based partition of the input mesh geometry.

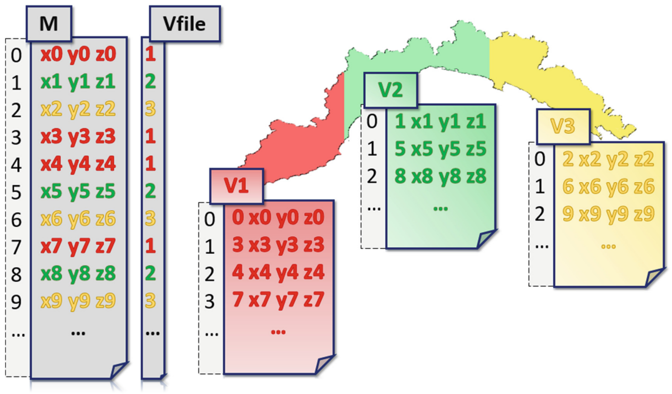

Fig. 2. Vertex classification. For each BSP cell, a corresponding file is created. Vertices are read one by one and assigned based on their spatial location. *Global* indexes are shown on the left of the original M, while *local* indexes are on the left of each V_i. For each vertex in M, both its global index and its coordinates are written on the corresponding V_i. V_{file} stores, for each vertex, the ID of the corresponding BSP cell. V_{file} is exploited during triangle classification to identify where the vertices of each triangle are located.

Independent Sets. An adjacency graph for the submeshes is defined where each node represents a BSP cell, and an arc exists between two nodes if their corresponding BSP cells are "mesh-adjacent". Two cells are considered to be mesh-adjacent if their corresponding submeshes share at least one vertex, that is, at least one triangle is intersected by the splitting plane between the two cells. Based on this observation, the adjacency graph is built during triangle classification and kept updated at each assignment. The problem of grouping together submeshes that are independent (e.g. no arc exists between the corresponding nodes) is solved by applying a greedy graph coloring algorithm [22]. Submeshes that belong to the same independent set can be processed simultaneously. Intuitively, the maximum number of nodes included in the same group is limited by the number of available processing services.

Post-processing. Depending on the specific geometric operation to be performed by processing services, some additional information from submesh's neighborhood may be required (e.g. the 1-ring neighborhood of boundary vertices is necessary to perform Laplacian smoothing). In these cases, a post-processing step is required to extract, for each submesh, the elements that constitute such a "support neighborhood". Such an information is then delivered to the processing service along with submesh to be processed.

5.2 Processing Services

Each processing service receives an input submesh and is asked to return an output mesh. If required, the submesh's support neighborhood is also provided. Processing services can modify both inner and boundary elements, while any possible support neighborhood must be kept unchanged. When boundary elements are modified, such modifications must be returned, so that the boundary of adjacent submeshes can be synchronized.

Besides the output mesh and possible modifications on the boundary, each processing service also encodes the list of boundary vertices of the output mesh into an additional file. Such a boundary information is used by the engine to efficiently merge the processed submeshes within a single model (Sect. 5.3).

Parallel Processing. When the same geometric operation is provided by more than one processing service, the engine exploits the generated independent sets to enable parallel processing. Each processing service is required to follow the rules described above. In the first iteration, each submesh in the current independent set is processed. Besides its output submesh, each processing service produces an additional file describing which modifications have been applied on the submesh boundary. This information is appended to adjacent submeshes and used a constraint during the next iterations (Fig. 3).

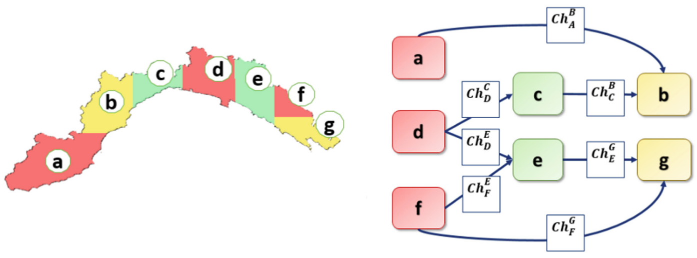

Fig. 3. Boundary synchronization. As an example, M_a and M_b are two neighbor submeshes. M_a is processed first. During the processing of M_a, all the changes introduced on the part of its boundary which is shared with M_b are stored in a file, namely Ch_b^a. When the turn of M_b comes, its processing service receives Ch_b^a and constrains M_b's boundary to change according to these instructions. Submeshes with the same color belong to the same independent set and can be processed simultaneously.

5.3 Output Merging

The engine is responsible for merging all the processed submeshes to generate a single indexed mesh. Mainly, the engine has two issues to deal with. First, vertices shared among two or more neighbor submeshes have to be identified and merged into a single point. Second, triplets of indexes representing triangles have to be rebuilt according to the final output indexing.

Since the final output may be too large to be loaded into main memory, an out-of-core merging method is proposed. As aforementioned, each processing service is also required to return the list of boundary vertices of the output mesh. Such a list is exploited to identify boundary vertices with no need to load the entire submesh. Algorithm 1 shows a more technical overview of our merging method.

Algorithm 1. Merge algorithm. n processed submeshes M'_i are merged into a single output M'. For each M'_i, the list of boundary vertices is stored in BV_i, encoded as a pair $\langle l, g \rangle$, where l is the local index and g the global index. Each BV_i is sorted by local index. An in-core map Map is used to store, for each boundary vertex already written to the final output, a mapping between its global index and its position in the merged mesh.

```
1:  procedure MERGE($M'_1, ..., M'_n, BV_1, ..., BV_n$)
2:      Create $V_f$ and $T_f$ files
3:      Create empty $Map$
4:      $V_c \leftarrow 0$                                  ▷ number of vertices added to final output
5:      for each pair $\langle M'_i, BV'_i \rangle$ do
6:          $\langle l, g \rangle \leftarrow$ first pair in $BV_i$
7:          Allocate $V(M'_i)$                                          ▷ an empty vector
8:          for each $v \in M'_i$ do
9:              $l_v \leftarrow$ local index of $v$
10:             if $l_v \neq l$ then                                   ▷ $v$ is an inner vertex
11:                 Write $v$ coordinates in $V_f$
12:                 Append $V_c$ to $V(M'_i)$ and increment $V_c$
13:             else                                                ▷ $v$ is a boundary vertex
14:                 $f_v \leftarrow Map.find(g)$
15:                 if $g$ is not found then                              ▷ $v$ is not in $V_f$
16:                     Write $v$ coordinates in $V_f$
17:                     Append $V_c$ to $V(M'_i)$
18:                     $Map.add(\langle g, V_c \rangle)$ and increment $V_c$
19:                 else                                             ▷ $v$ is already in $V_f$
20:                     $V(M'_i)[l_v] \leftarrow f_v$
21:                 $\langle l, g \rangle \leftarrow$ next pair in $BV'_i$
22:         for each $t := (v_1, v_2, v_3) \in M'_i$ do
23:             Write $V(M'_i)[v_1]$, $V(M'_i)[v_2]$ and $V(M'_i)[v_3]$ in $T_f$
24:     $M' \leftarrow$ [header information] $+V_f + T_f$
25:     return $M'$
```

6 Distributed Input Dataset

When the input model is too large to be stored on a single machine, the mesh is stored as a distributed collection of files representing adjacent sections of the whole input model [4]. In this case, the engine may not have sufficient storage resources to download the whole input mesh on its own disk, and the existing sections of the model may not be compatible with the hardware limitations of the machines which host the processing services. Also, the final output may be too large to be stored on the engine's disk. Thus, a different approach is required to re-partition the input dataset (Fig. 4) and to generate the final output.

In principle, one could exploit the approach described in Sect. 5.1 to partition input subemeshes which are too large, while the smallest ones can be processed as they are. Nevertheless, such an approach is inefficient when the number of small input submeshes is too large (i.e. because submeshes are unnecessarily small for the sake of processing). We propose an input repartitioning approach that maximizes the exploitation of available processing services.

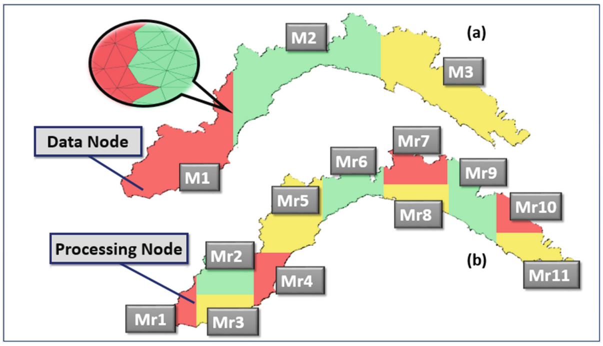

Fig. 4. Repartitioning. (a) The overall input mesh M. Each M_i is stored on a different data node. (b) Repartitioned M. M^r_js with the same color are included in the same independent set.

Our reference scenario is shown in Fig. 5. The engine manages the input re-partitioning and the final output generation by delegating part of the computation to the data nodes. When the re-partitioning has been completed, a new collection of adjacent submeshes $\langle M^r_1, M^r_2, ..., M^r_m\rangle$ representing the original M is distributedly stored on the data nodes. The engine is responsible for grouping the generated submeshes into independent sets and for orchestrating the processing nodes to enable parallel processing. The result of each processing service is delivered back to the data node that hosts the input. It is worth noticing that, in this scenario, the engine works as an interface among data nodes and processing nodes. When a node is triggered for execution, it receives from the engine the address of the input data to be processed.

6.1 Input Repartitioning

The input repartitioning method is an extension of the the previously described approach (Sect. 5.1), where part of the computation is delegated to the data nodes.

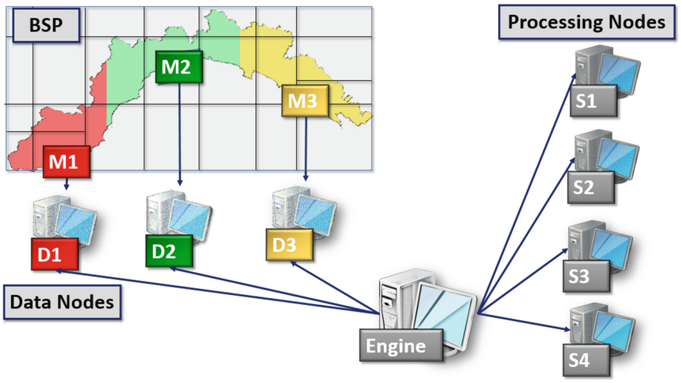

Fig. 5. Scenario. The original input mesh is defined as a collection of adjacent indexed submeshes $\langle M_1, M_2, M_3 \rangle$. Each M_i is stored on a different data node D_i. The engine manages the input re-partitioning and the final output generation by delegating part of the computation to the data nodes, while processing nodes are invoked for the actual computation.

Pre-processing. Each data node is required to compute both the bounding box and a representative vertex downsampling of its own original submesh. The engine exploits this information to build a BSP of the whole original mesh M. The BSP is stored on file to be distributed to the data nodes.

Vertex and Triangle Classification. Each data node assigns vertices and triangles of its original input portion to the corresponding BSP cell, according to their spatial location.

Generation of Independent Sets. The engine is responsible for building the adjacency graph for the generated submeshes and group them into independent sets. In some cases, a generated submesh may include portions of different original portion (e.g. M_2^r in Fig. 6). While building the independent sets, the engine is responsible to group together data coming from different data nodes and to send all of them to the same processing node.

6.2 Processing Services

When the input re-partitioning is completed, the dataset is ready to be processed. The engine is responsible of managing the actual processing by iteratively distributing each independent set to the available processing services. Note that processing services work as described in Sect. 5.2. Additionally, when a submesh is compose by portions coming from different data nodes (e.g. M_2^r in Fig. 6), a processing service is required to load all the portions into its main memory and merge them together before starting the actual computation. Since submeshes are guaranteed to be sufficiently small to be completely loaded, the merging

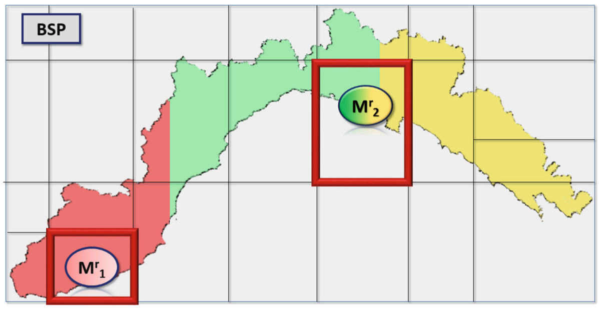

Fig. 6. The distributed BSP at the end of the repartitioning of $M = \langle M_1, M_2, M_3 \rangle$. As an example, M_1^r is a subpart of the original M_1 (red), while M_2^r is composed by two subparts of the original M_2 (green) and M_3 (yellow) respectively. (Color figure online)

operation is performed by an incore method. Consistently with the previous approach, each processing service generates an output mesh and an additional file listing its boundary vertices. Also, files storing the list of modifications applied on the submesh boundary are built and distributed to allow boundary synchronization among neighbor submeshes.

6.3 Distributed Output Merging

When all the submeshes have been processed by the available processing nodes, they should be merged to generate the final output. When the engine has not enough storage resources, the disk space of the data nodes is exploited. We assume that each data node has sufficient free storage resources to collectively store a final merged output.

Let D_i be the data node storing a set of generated submeshes. The outputs of the processing services responsible for their elaboration is returned to D_i, which is responsible for merging them into a single mesh by exploiting the previously described approach (Sect. 5.3) to perform the task. The final output is a distributed collection of processed submeshes, representing adjacent pieces of a huge mesh M', which is a modified version of the original M.

7 Mesh Simplification

The distributed simplification algorithm works as follows. In the first step, the engine partitions the mesh into a set of submeshes. Depending on the representation of the input dataset (distributed or not), one of the previously described algorithms (Sects. 5.1 or 6.1) is selected to perform the task. Generated submeshes are then grouped into independent sets. Each independent set is iteratively sent to the processing nodes for simplification. In the first iteration, each submesh is simplified in all its parts according to the target accuracy. Besides the simplified mesh, each processing service produces a set of additional files identifying which vertices on the submesh boundary were removed during simplification. Specifically, each file identifies vertices shared with a specific neighbor. When processing adjacent submeshes, this information is used as a constraint for their own simplification. When all the independent sets are processed, the final output is generated by joining the simplified submeshes along their boundaries, which are guaranteed to match exactly. If the engine has sufficient resources, the algorithm described in Sect. 5.3 is exploited. Otherwise, the approach described in Sect. 6.3 enables the possibility to distributedly store the final output.

Adaptivity. Each submesh is simplified by a single processing service through a standard iterative edge-collapse approach based on quadric error metric [19]. Every edge is assigned a "cost" that represents the geometric error introduced should it be collapsed. On each iteration, the lowest-cost edge is actually collapsed, and the costs of neighboring edges are updated. In order to preserve the appearance of the original shape, the simplification algorithm applied by each service stops when a maximum error max_E is reached. This approach provides an adaptively optimal result [7]. For each vertex, a quadric matrix is calculated without the need of any support neghborhood: if the vertex is on the submesh boundary, a partial quadric for boundaries [20] is calculated. To preserve the input topology, we constrain boundary vertices which are shared by more than two submeshes. By not simplifying these vertices, and by verifying the link condition for all the other vertices, we can guarantee that the resulting simplified submesh is topologically equivalent to the input.

Other Features. Our simplification algorithm proves the benefits provided by our partitioning/merging approach, but it also has other noticeable characteristics. Table 1 summarized the main features of such an algorithm and a comparison with the state of the art. However, their description would bring us too far from the scope of this paper, hence we refer the reader to [7] for details.

Table 1. Feature-based comparison with the state of the art.

	[26]	[36]	[5]	Ours
Out–Of–Core input	✓	✗	✓	✓
Out–of–Core output	✓	✗	✗	✓
Adaptivity	✗	✓	✓	✓
Distributable	✓	✓	✗	✓
Indexed mesh support	✗	✓	✓	✓

8 Results and Discussion

For the sake of experimentation, the proposed Workflow Engine has been deployed on a standard server running Windows 7, whereas other web services implementing atomic tasks have been deployed on different machines to constitute a distributed environment. However, since all the servers involved in our experiments were in the same lab with a gigabit network connection, we needed to simulate a long-distance network by artificially limiting the transfer bandwidth to 5 Mbps. All the machines involved in the experimentation are equipped with Windows 7 64bit, an Intel i7 3.5 GHz processor, 4 GB Ram and 1 T hard disk.

Then, to test such a system we defined multiple processing workflows involving the available web services. The dataset has been constructed by selecting some of the most complex meshes currently stored within the Digital Shape Workbench [3]. As an example, one of our test workflows is composed by the following operations: Removal of Smallest Components (RSC), Laplacian Smoothing (LS), Hole Filling (HF), and Removal of Degenerate Triangles (RDT). The same workflow was run on all the meshes in our dataset to better evaluate the performance gain achievable thanks to our concurrent mesh transfer protocol. Table 2 reports the size of the output mesh and the size of the correction file after each operation (both after compression) whereas Table 3 shows the total time spent by the workflow along with a more detailed timing for each single phase. As expected, the corrections related to tasks that locally modify the model (eg. RSC, HF, RDT) are significantly smaller than the whole output mesh by several orders of magnitude; corrections regarding more "global" tasks (eg. LS) are also smaller than the output mesh, although in this latter case the correction file is just two/three times smaller than the whole output. Nevertheless, these results confirm that the proposed concurrent mesh transfer protocol provides significant benefits when the single steps produce mainly little or local mesh changes.

For each mesh in our dataset, Table 3 shows the time required to be processed both in case the mesh transfer protocol is exploited (first line) or not (second line). Specifically, the time spent by each algorithm is reported in columns RSC, LS, HF, RDT, while columns $T_1 \dots T_3$ and columns $U_1 \dots U_3$ show the time needed to transfer the correction file to the subsequent Web service and the time spent to update the mesh by applying the correction respectively. For the sake of comparison, below each pair (T_i, U_i) we also included the time spent by transferring the whole compressed result instead of the correction file, and the overall relative gain achieved by our protocol is reported in the last column. It is worth noticing that, in all our test cases, the sum of the transfer and update times is smaller than the time needed to transfer the whole mesh, with a significant difference when the latter was produced by applying little local modifications on the input.

To test our partitioning and simplification algorithm, large meshes extracted from the Stanford online repository [1], from the Digital Michelangelo Project [2] and from the IQmulus Project [4] were used as inputs. Some small meshes have been included in our dataset to evaluate and compare the error generated by the part-by-part simplification.

For each input model, we ran several tests by varying the number of involved processing nodes and the maximum error threshold. We fixed the number N_v

Table 2. Output sizes (in KB). For each mesh and for each task, the first line shows the size of the compressed output mesh, while the second line reports the size of the compressed correction. Average compression ratio is 5:1. Acronyms indicate Removal of Smallest Components (RSC), Laplacian Smoothing (LS), Hole Filling (HF), and Removal of Degenerate Triangles (RDT). A modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (*) in order to preserve their largest boundary.

Mesh	RSC	LS	HF	RDT
Rome*	14915	15551	14915	13166
	1	1425	1	1
Dolomiti*	11146	11637	11146	10588
	1	1402	1	1
Isidore	20573	23333	23717	25497
	11	9433	154	2
Nicolo	19498	21447	20601	20171
	3	9296	48	2
Neptune	39881	40131	39891	39937
	1	15237	1	1
Ramesses	17484	19544	19934	19802
	3	8754	149	3
Dancers	16457	18037	18325	18116
	1	7220	80	1

of vertices that should be assigned to each submesh to 1 M for very large input meshes. Table 4 shows the time spent by the system to finish the whole computation. The achieved speedup S_i is also shown, computed as $S_i = \frac{Time_1}{Time_i}$, where $Time_1$ is the sequential time and $Time_i$ is the time required to run the simplification on i servers. As expected, speedups are higher when the number of available processing nodes increases. More noticeably, speedup increases as the input size grows. Table in Fig. 7 reports the relation between the size of the input, and shows the time needed to partition it and the benefits provided by our re-partitioning algorithm. As a summarizing achievement, our method could partition the 25 GB OFF file representing the Atlas model (≈0.5 billions triangles) in ≈25 min. As a matter of comparison, the engine's operating system takes more than 8 min to perform a simple local copy of the same file. Furthermore, the last experiment in Table 4 shows the time required to process the full-resolution Liguria model (1.1 Tb), represented as a collection of 10 indexed meshes stored on just as many data nodes. The repartitioning step requires less than 3 h. Note that more than 24 h would be required if the model is stored as a single OFF file on the engine hard disk.

To test the quality of output meshes produced by our algorithm, we used Metro [12] to measure the mean error between some small meshes and their

Table 3. Times (in seconds). Acronyms indicate Input Broadcast (**IB**), Removal of Smallest Components (**RSC**), Laplacian Smoothing (**LS**), Hole Filling (**HF**), and Removal of Degenerate Triangles (**RDT**). Cells labelled by $\mathbf{T}_i$ indicate the time needed to transfer the correction file. Cells labelled by $\mathbf{U}_i$ indicate the time needed to update the mesh by applying the correction. **Total** indicates the overall time required for the execution. **Benefits** indicates, for each experiment, how much the computation time decreases when our protocol is exploited. Computation times are reported both in case the mesh transfer protocol is exploited (upper line) and in case of "traditional" transfer (lower line). Note that a modified version of the Hole Filling algorithm has been run to process "2.5D" geospatial data (*) in order to preserve the largest boundary.

Mesh (# vertices)	IB	RSC	T_1	U_1	LS	T_2	U_2	HF	T_3	U_3	RDT	Total	Benefits
Rome* (957456)	20.4	5.8	0.0	0.0	8.4	2.3	9.4	5.5	0.0	0.0	6.9	58.7	104%
			23.9			24.9			23.9			119.7	
Dolomiti* (810000)	15.8	4.9	0.0	0.0	7.2	2.2	7.8	4.6	0.0	0.0	5.7	48.2	92%
			17.8			18.6			17.8			92.4	
Isidore (1071671)	33.0	7.7	0.0	5.8	12.4	15.1	7.1	8.4	0.2	6.0	13.8	109.5	67%
			32.9			37.3			37.9			183.4	
Nicolo (945924)	31.2	6.5	0.0	4.8	10.5	14.9	6.1	7.5	0.1	4.9	11.5	98.0	69%
			31.2			34.3			33.0			165.7	
Neptune (1321838)	63.8	13.0	0.0	0.0	18.6	24.4	11.0	12.6	0.0	0.0	14.4	157.8	99%
			63.8			64.2			63.8			314.2	
Ramesses (775715)	28.0	6.7	0.0	4.3	9.6	14.0	5.4	7.0	0.2	4.5	10.3	90.0	70%
			28.0			31.3			31.9			152.8	
Dancers (703207)	26.3	4.9	0.0	0.0	7.3	11.6	4.3	5.2	0.1	3.6	7.0	70.3	92%
			26.3			28.9			29.3			135.2	

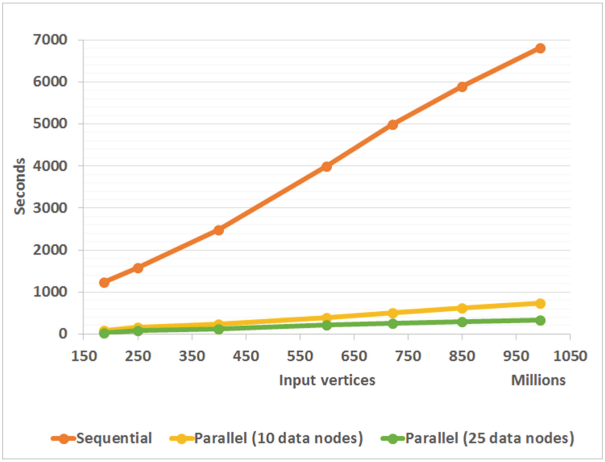

Fig. 7. Partitioning time vs input size: we can observe an approximately linear growth of the processing time as the input grows. When the input is pre-partitioned and scattered on different disks, the re-partitioning approach speeds up the input segmentation.

Table 4. Times (in seconds). Column labels: max_E is the threshold error (one thousandth of the bounding box diagonal of the input in all these experiments) expressed in absolute values, N_s is the number of available services, $\#ISs$ is the number of generated independent sets, while $\#V_o$ is the number of output vertices. Also, times are shown for each step: Partitioning (P), Simplification (S), and Merging (M). All the input meshes are stored as a single OFF file, except Liguria model (*) that is represented as a collection of 10 indexed meshes distributedly stored on 10 data nodes.

Input			$\#ISs$	$\#V_o$	Times				Speedup
Mesh (# vertices)	max_E	N_s			P	S	M	Total	
Terrain (67873499)	0.00006	1	117	12166	497	302	1	800	–
		10	13	11697		64.45		562.45	1.42
		25	6	11660		13.37		511.37	1.56
St. Matthew (186836670)	3.01716	1	285	119121	1225.5	805.65	2.5	2033.65	–
		10	29	119035		104.05		1332.05	1.53
		25	13	119308		47.65		1275.65	1.59
Atlas (245837027)	3.35350	1	395	234084	1441	1481.25	4.5	2926.75	–
		10	42	234081		157.05		1602.55	1.83
		25	18	234091		72.95		1518.45	1.93
Liguria* (12986836670)	0.00006	1	26077	12174	9647	67278.60	1	76926.60	–
		10	3276	12144		8452.08		18100.08	4.25
		25	896	12153		2311.70		11959.70	6.42

simplifications. Results show that the number of services does not significantly affect the quality of the output. Unfortunately, Metro is based on an incore approach that evaluates the Hausdorff distance between the input mesh and the simplified one. Such an approach cannot be used to evaluate the quality of simplified meshes when the original version is too large. In these cases, quality can be assessed based on a visual inspection only. Figures 8, 9, and 10 show that high quality is preserved in any case and is not sensibly affected by the number of involved services.

8.1 Limitations

We enabled the possibility to analyze and process large geometric datasets. Nevertheless, some limitations should be taken into account when designing a parallel algorithm that exploits our divide-and-conquer method. First, our approach supports algorithms that modify the existing geometry, but does not consider the possibility to generate new geometric elements based on non strictly local information (e.g. hole filling). Second, processing services are assumed to perform local operations by analyzing at most a support neighborhood. Our divide-and-conquer approach is not suitable for processing services requiring global information. In this latter case, our proposal can be exploited only if an approximated result is accepted.

Nonetheless, for some specific global operations, our system can be easily customized and exploited as well. As an example, small components (e.g. those with low triangle counts) of the original input may be partitioned by the BSP. In this case, each processing service can just count the number of triangles of each

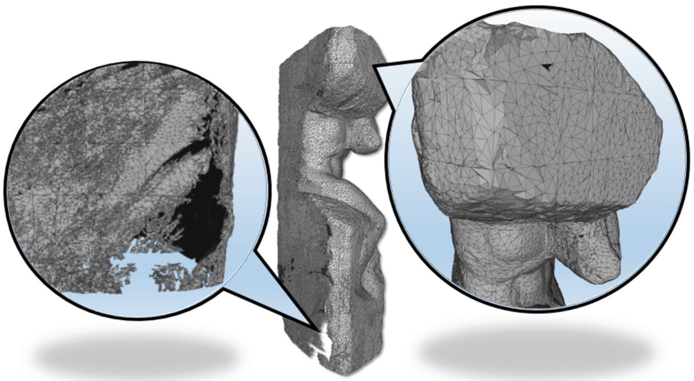

Fig. 8. Details of Atlas model simplified by exploiting 25 available services (original: ≈256 M vertices, simplified: ≈234 K vertices)

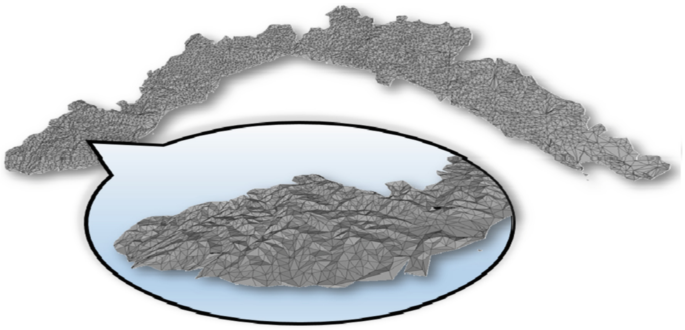

Fig. 9. Detail of simplified Terrain model (original: ≈68 M vertices, simplified: ≈115 K vertices). Nearly high fields are naturally supported

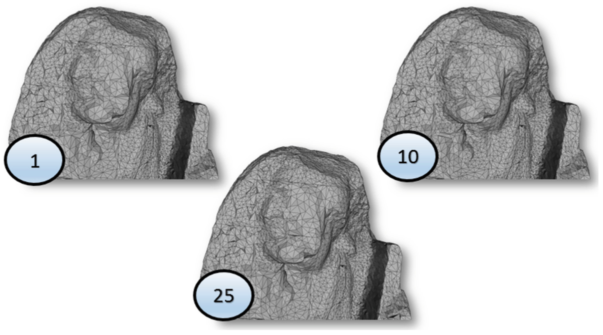

Fig. 10. Detail of St Matthew model simplified by 1, 10, and 25 servers (original: ≈187 M vertices, simplified: ≈1195 K vertices)

component which is connected in the submesh. Such an information is returned to the engine that, thanks to the BSP adjacency graph, can sum the partial counts for adjacent sub-components without the need to explicitly load mesh elements in memory. Thus, the engine can identify the smallest components and tell the services to remove them in a second iteration.

9 Conclusions

We proposed a workflow-based framework to support collaborative research in geometry processing. The platform is accessible from any operating system through a standard Web browser with no hardware or software requirements. A prototypal version is available at http://visionair.ge.imati.cnr.it/workflows/. Scientists are allowed to remotely run geometric algorithms provided by other researchers as Web services and to combine them to create executable geometric workflows. No specific knowledge in geometric modelling and programming languages is required to exploit the system.

As an additional advantage, short-lasting experiments can be re-executed on the fly when needed and there is no more need to keep output results explicitly stored on online repositories. Since experiments can be efficiently encoded as a list of operations, sharing them instead of output models sensibly reduces required storage resources. The architecture is open and fully extensible by simply publishing a new algorithm as a Web service and by communicating its URL to the system. Moreover, we have demonstrated that the computing power of a network of PCs can be exploited to significantly speedup the processing of large triangle meshes and we have shown that the overhead due to the data transmission is much lower than the gain in speed provided by parallel processing.

In its current form, our system has still a few weaknesses. First, experiments can be reproduced only as long as the involved Web services are available and are not modified by their providers. To reduce the possibility of workflow decay [42] a certain level of redundancy would be required. Second, our system does not allow to execute semi-automatic pipelines, that is with user interaction. Such a functionality would require the engine to interrupt the execution waiting for the user intervention.

Several future directions are possible, both in terms of improvement of the platform capabilities and enrichment of the geometry processing operations. One of the objectives of our future research is to simplify the work of potential contributor by enabling the engine to automatically compute the list of editing operations. A possible solution may be inspired on [16], even if the high computational complexity of this method would probably hinder our gain in speed.

Acknowledgements. This work is partly supported by the EU FP7 Project no. ICT–2011-318787 (IQmulus) and by the international joint project on Mesh Repairing for 3D Printing Applications funded by Software Architects Inc (WA, USA). The authors are grateful to all the colleagues at IMATI for the helpful discussions.

References

1. The Stanford 3D Scanning Repository (1996)
2. The Digital Michelangelo Project (2009)
3. DSW v5.0 - visualization virtual services (2012)
4. Iqmulus: A High-volume Fusion and Analysis Platform for Geospatial Point Clouds, Coverages and Volumetric Data Sets (2013)
5. Brodsky, D., Pedersen, J.B.: Parallel model simplification of very large polygonal meshes. In: Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA 2002), vol. 3, pp. 1207–1215 (2002)
6. Cabiddu, D., Attene, M.: Distributed processing of large polygon meshes. In: Proceedings of Smart Tools and Apps for Graphics (STAG 2015) (2015)
7. Cabiddu, D., Attene, M.: Large mesh simplification for distributed environments. Comput. Graph. **51**, 81–89 (2015)
8. Campen, M.: WebBSP 0.3 beta (2010). http://www.graphics.rwth-aachen.de/webbsp
9. Chiang, Y.J., Silva, C.T., Schroeder, W.J.: Interactive out-of-core isosurface extraction. In: IEEE Visualization 1998, pp. 167–174 (1998)
10. Cignoni, P., Montani, C., Rocchini, C., Scopigno, R.: External memory management and simplification of huge meshes. IEEE Trans. Vis. Comput. Graph. **9**(4), 525–537 (2003)
11. Cignoni, P., Corsini, M., Ranzuglia, G.: Meshlab: an open-source 3d mesh processing system. ERCIM News **73**, 45–46 (2008)
12. Cignoni, P., Rocchini, C., Scopigno, R.: Metro: measuring error on simplified surfaces. Comput. Graph. Forum **17**(2), 167–174 (1998)
13. Claro, D.B., Albers, P., Hao, J.: Selecting web services for optimal composition. In: Proceedings of the 2nd International Workshop on Semantic and Dynamic Web Processes (SDWP 2005), pp. 32–45 (2005)
14. Cuccuru, G., Gobbetti, E., Marton, F., Pajarola, R., Pintus, R.: Fast low-memory streaming MLS reconstruction of point-sampled surfaces. In: Proceedings of Graphics Interface, GI 2009, pp. 15–22. Canadian Information Processing Society, Toronto (2009)
15. Dehne, F., Langis, C., Roth, G.: Mesh simplification in parallel. In: Proceedings of Algorithms and Architectures for Parallel Processing (ICA3P 2000), pp. 281–290 (2000)
16. Denning, J.D., Pellacini, F.: Meshgit: diffing and merging meshes for polygonal modeling. ACM Trans. Graph **32**(4), 35: 1–35: 10 (2013)
17. Farr, T.G., Rosen, P.A., Caro, E., Crippen, R., Duren, R., Hensley, S., Kobrick, M., Paller, M., Rodriguez, E., Roth, L., Seal, D., Shaffer, S., Shimada, J., Umland, J., Werner, M., Oskin, M., Burbank, D., Alsdorf, D.: The shuttle radar topography mission. Rev. Geophys. **45**(2), RG2004 (2007)
18. Franc, M., Skala, V.: Parallel triangular mesh reduction. In: Proceedings of Scientific Computing, ALGORITMY 2000, pp. 357–367 (2000)
19. Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: Proceedings of SIGGRAPH 1997, pp. 209–216 (1997)
20. Heckbert, P.S., Garland, M.: Optimal triangulation and quadric-based surface simplification. J. Comput. Geometry Theory Appl. **14**(1–3), 49–65 (1999)
21. Hollingsworth, D.: Workflow management coalition - the workflow reference model. Technical report, January 1995

22. Hutter, M., Knuth, M., Kuijper, A.: Mesh partitioning for parallel garment simulation. In: Proceedings of WSCG 2014, pp. 125–133 (2014)
23. Isenburg, M., Lindstrom, P.: Streaming meshes. In: Visualization (VIS 2005), pp. 231–238. IEEE, October 2005
24. Isenburg, M., Lindstrom, P., Gumhold, S., Snoeyink, J.: Large mesh simplification using processing sequences. In: Visualization (VIS 2003), pp. 465–472, October 2003
25. Lindstrom, P.: Out-of-core simplification of large polygonal models. In: Proceedings of SIGGRAPH 2000, pp. 259–262 (2000)
26. Lindstrom, P., Silva, C.T.: A memory insensitive technique for large model simplification. In: IEEE Visualization, pp. 121–126 (2001)
27. Maglo, A., Lavoué, G., Dupont, F., Hudelot, C.: 3d mesh compression: survey, comparisons, and emerging trends. ACM Comput. Surv. 47(3), 44: 1–44: 41 (2015)
28. Meredith, J.S., Ahern, S., Pugmire, D., Sisneros, R.: EAVL: the extreme-scale analysis and visualization library. In: Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association (2012)
29. Möbius, J., Kobbelt, L.: OpenFlipper: an open source geometry processing and rendering framework. In: Boissonnat, J.-D., Chenin, P., Cohen, A., Gout, C., Lyche, T., Mazure, M.-L., Schumaker, L. (eds.) Curves and Surfaces 2010. LNCS, vol. 6920, pp. 488–500. Springer, Heidelberg (2012). doi:10.1007/978-3-642-27413-8_31
30. Moreland, K., Ayachit, U., Geveci, B., Ma, K.L.: Dax toolkit: a proposed framework for data analysis and visualization at extreme scale. In: IEEE Symposium on Large Data Analysis and Visualization (LDAV 2011), pp. 97–104 (2011)
31. Pitikakis, M.: A semantic based approach for knowledge management, discovery and service composition applied to 3D scientif objects. Ph.D. thesis, University of Thessaly, School of Engineering, Department of Computer and Communication Engineering (2010)
32. C. Sewell, Lo, L.T., Ahrens, J.: Portable data-parallel visualization and analysis in distributed memory environments. In: IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV 2013), pp. 25–33 (2013)
33. Shaffer, E., Garland, M.: Efficient adaptive simplification of massive meshes. In: Proceedings of Visualization 2001, pp. 127–134 (2001)
34. Shontz, S.M., Nistor, D.M.: CPU-GPU algorithms for triangular surface mesh simplification. In: Jiao, X., Weil, J.-C. (eds.) Proceedings of the 21st International Meshing Roundtable, pp. 475–492. Springer, Heidelberg (2013)
35. Silva, C., Chiang, Y., Corra, W., El-sana, J., Lindstrom, P.: Out-of-core algorithms for scientific visualization and computer graphics. In: Visualization 2002 Course Notes (2002)
36. Tang, X., Jia, S., Li, B.: Simplification algorithm for large polygonal model in distributed environment. In: Huang, D.-S., Heutte, L., Loog, M. (eds.) ICIC 2007. LNCS, vol. 4681, pp. 960–969. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74171-8_97
37. Thomaszewski, B., Pabst, S., Blochinger, W.: Parallel techniques for physically based simulation on multi-core processor architectures. Comput. Graph. **32**(1), 25–40 (2008)
38. Tiwari, A., Sekhar, A.K.T.: Workflow based framework for life science informatics. Comput. Biol. Chem. **31**(56), 305–319 (2007)
39. Touma, C., Gotsman, C.: Triangle mesh compression. In: Graphics Interface, pp. 26–34 (1998)

40. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva de la Hidalga, A., Balcazar Vargas, M.P., Sufi, S., Goble, C.: The Taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucl. Acids Res. **41**(Web Server issue), gkt328–W561 (2013)
41. Jianhua, W., Kobbelt, L.: A stream algorithm for the decimation of massive meshes. In: Proceedings of the Graphics Interface 2003 Conference, Halifax, Nova Scotia, Canada, pp. 185–192, June 2003
42. Zhao, J., Gomez-Perez, J.M., Belhajjame, K., Klyne, G., Garcia-cuesta, E., Garrido, A., Hettne, K., Roos, M., De Roure, D., Goble, C.: Why workflows break: understanding and combating decay in Taverna workflows, pp. 1–9 (2012)