Interoperability of multiscale visual representations for satellite image big data

François Merciol, Antoine Sauray, Sébastien Lefèvre

INTEROPERABILITY OF MULTISCALE VISUAL REPRESENTATIONS FOR SATELLITE IMAGE BIG DATA

François Merciol, Antoine Sauray and Sébastien Lefèvre

Université Bretagne Sud – IRISA
Campus de Tohannic, BP 573, 56017 Vannes Cedex, France

ABSTRACT

In this paper, we propose an interoperable solution for dealing with hierarchical representations of satellite images. Computationally intensive construction of the tree representation is performed on the server side, while the need for computational resources on the client side are greatly reduced (tree postprocessing or visualization). The proposed scheme is interoperable in the sense that it does not impose any constraints on the client environment, and API for C++, Java and Python languages are currently available. The communication is performed nodewise in a binary format using array structure for limiting the memory footprint.

***Index Terms*—** Hierarchical image representation, Tree, Client-server framework, Interoperability.

1. INTRODUCTION

Recent years have seen Earth Observation entering in the Big Data era. This brings new challenges related to the V's: volume, variety, velocity, as well as veracity and visibility. Very high spatial resolution (e.g., a Pleiades scene is $40,000 \times 40,000$ pixels) as well as high temporal revisiting frequency (e.g., Sentinel missions will offer worldwide updates of a scene every 5 days) lead to massive datasets to be processed in order to produce land cover map, detect objects of interest or changes, retrieve specific patterns, or perform (manual or assisted) visual interpretation. To do so, very efficient solutions have to be designed, requiring algorithms of appropriate complexity relying on efficient data structures and operating on adequate architectures (e.g., cloud, HPC, etc). Representing images through multiscale representations based on tree structures has been proved to be a relevant framework for efficient processing of large image data [1]. Besides, trees have been successful in addressing various tasks in remote sensing [2, 3, 4].

However, one of the major bottlenecks of such a hierarchical image analysis strategy is the need for prior computation of the tree representation. The computational cost of building a tree from a large dataset is indeed significantly high, even in the case of efficient algorithms [5]. In this paper, we propose a new strategy based on a client-server model, where the tree is built on a server before being used by the client. Such a strategy presents many advantages: extensive (and possibly scalable) computational resources are needed only on the server side, since the client will only process the tree structure (requiring much less resources than dealing with raw data); similarly to computing resources, network resources are required on the server side between the data source and the processing server, not on the client side; if the satellite image comes with access restrictions, which is the case for most commercial satellites that do not allow distributing the raw data, it is possible to provide a tree-based product (non bijective image representation) to the client; any compatible client can be used, even running in some environments not known for their performance; finally, if the client is only interested by some part of the image (and not the full data), only a subset of the tree can be sent by the server.

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR) under reference ANR-13-JS02-0005-01 (Asterix project).

2. INTEROPERABILITY

The client-server strategy is designed here in such a way that there is no need for a unique environment (software, coding language, operating system) on both sides. Interoperability is ensured through several wrappers allowing to access the API from the client side using various environments (C++, Java, Python, etc.). Any postprocessing or visualization tool can then be used on the client side. Let us note that the proposed solution can also be used on the client side only allowing different tools for tree construction and manipulation. The only requirement is a standard scheme to support the distribution of the (full or partial) tree from the server to the client.

There are many interoperable formats, the most famous being XML. However, exchanging large datasets using a verbose language such as XML is not efficient. So we have rather used byte arrays containing scalar values (either integers or real numbers). It requires some appropriate interface between client and server components (i.e. to use the same coding format: number of bits, byte ordering). To illustrate the interoperability offered by our framework, we provide 3 code snippets for C++, Java and Python in Fig. 1. We can observe that the calls are very similar from one language to the other.

```
// C++ snippet
CppSession<Component, Leaf> *cppSession
  = CppSession<Component, Leaf>::getInstance
    (serverUrl, imageUrl);
cppSession->buildTree (algo, metric);
cppSession->createLeaves ();
cppSession->copyTree ();

// Java snippet
Session s = Session.getInstance (urlServer, urlImage);
s.buildTree (algo, metric);
s.createLeaves ();
root = s.copyTree (connectedComponentCreator);

// Python snippet
s1 = PythonSession (urlServer, path)
s1.buildTree (algo, metric)
s1.createLeaves (leafCreator)
s1.copyTree (connectedComponentCreator)
```

Fig. 1. Client requests for tree construction in C++, Java and Python.

3. IMPLEMENTATION

For the sake of performances, the server component has been coded in C++. It benefits from significant optimizations with no overhead for tree coding (only the minimal number of quartets needed is required). API for tree communication with C++, Java, and Python-compliant clients have been designed. The system is made freely available to the scientific community[1].

The proposed framework is illustrated through an example (Fig. 3). We assume here that nodes contain scalar data, but the solution can be easily adapted to more advanced representations.

First, a given client has to choose the appropriate server based on its needs. Indeed, a pool of servers might be available, each server coming with its specific algorithms (kind of tree, kind of data to be processed, etc). The client establishes a connection to a server through a Kernel object. Once the communication link has been established, messages are exchanged between the client and the server through a dedicated protocol. To do so, it relies on the interface given in Fig. 2. Let us observe that the communication is here of type unconnected socket (e.g. with one TCP connection by HTTP request). This protocol relies on the BOOST library, both for socket management and vector serialization.

```
class UTP {
public:
  static UTP *getInstance (const string &url);

  int openSession (const string &url);
  void closeSession (int sessionId);
  void closeAllSession ();

  void buildTree (int sessionId, int tree, int weight);
  vector<int> getInfos (int sessionId);
  double getMillisec (int sessionId);
  vector<float> getValues (int sessionId);
  vector<int> getNodeChildren (int sessionId, int id);
  vector<int> getNodePixels (int sessionId, int id);
 //
};
```

Fig. 2. Interface available to the client (C++ example).

Once the Kernel object has been created, the input image has to be loaded on the server, resulting in a new Session object. The protocol for iterative communication of array structures is actually implemented within this Session object. As shown in the Snippets from Fig. 1, the creation of both Kernel and Session objects are embedded in a single function call. Depending on the application context, various data sources are to be considered. Indeed, the client might already store the image to process. In this case, the image is to be sent to the server over the network, before receiving back its tree representation. More interestingly, the server might already store the image data (e.g. in some business or research environments). As such, there is no network cost to transfer the original image. A last and more realistic scenario, especially when considering the current initiatives such as the EU Copernicus programme, considers that original data are actually made available through dedicated geospatial hubs (such as the one from ESA or from national space agencies). In this last case, we can reasonably assume that there exist a fast network connection between the server and the data hub.

Once the image is stored on the server, the latter computes its tree representation. To do so we assume that some efficient tree construction algorithms [5, 6] are available on the server. Besides the tree, several data structures are stored: the list of nodes in the image (array with 10 cells in the figure), the list of inner nodes or nodes with children (array with 4 cells, noted C*), the list of leaves or nodes with pixels (array with 8 cells, noted L*). These additional structures increase the overall storage cost on the server side. But let us recall that once the tree has been built, there is no need to store the initial image on the server anymore. Indeed, the tree can then be used at any time to answer a client request (both for the full image or only a part of it). If the tree is used in a single user session, the server will store the tree and the additional data structures until the end of the user session. If the image data shall remain available (through its tree representation) for future users, then all structures but the original data are kept.

After tree construction, the server sends a first array providing the image height and width (3×3), the number of nodes in the tree (10), the number of inner nodes or nodes with children (4), and the number of leaves or nodes with pixels (8). The client is then able to allocate the right amount of memory and initialize the local data structure (root, nodes, pixels). On the client side, an iterator is used to request the server to send information for each node of the tree. Inner nodes are considered first ($C0$ to $C3$). The server then send iteratively the information for each C_i through an array containing the ID of the node followed by the ID of its children (e.g., node 0 has children 1, 5, and 2). Once all inner nodes have been transferred, the client iterates on leaves ($L0$ to $L7$). It thus requests the server to provide information that consists for each node of an array containing the ID of the node followed by the ID of the pixels (e.g., node 2 points to pixel 3). The (full or partial) tree is then available on the client side.

4. EXPERIMENTS

It is rather difficult to assess the impact of network performances on the overall system. Thus we have decided to run the client and the server on the same computer to avoid being affected by unpredictable network behavior. Furthermore, we have also discard the effect of image loading over the network by storing the images directly on the server. We report the results obtained on a standard workstation (8 GB RAM, 4 cores Intel i5-3320M@2.60GHz).

We have compared between several configurations. The server is running in C++, and various clients are used: C++, Java and Python. For reference we have also included a Java standalone application (not client-server). We have considered various satellite image size (from 80,000 to 13,000,000 pixels) and types (panchromatic, multispectral), and use the notation $X \times N$ to denote an image containing X pixels with N spectral bands. Results are given in Fig. 1. We use here the α-tree implementation from Havel et al [5].

We can derive several observations from these experimental results. First, when small images are considered, the proposed client-server framework is not mandatory. Indeed, a standalone application is not too costly in this case, and can achieve a similar level of performance than the client-server mode (e.g. very similar times obtained for the full Java and C++/Java mode). Even on such small images, we can notice the strong impact of the core algorithm (e.g. 0.11 for the C++ construction, while 0.40 for the Java one). When image size increases, computation cost becomes an issue and the proposed framework achieves satisfying performances. Indeed, it can benefit from a significant optimization on the server side (i.e., compare between the full Java and C++ server). Furthermore, the cost for array communication and tree reconstruction on the client side is reasonable w.r.t. the overall cost. This allows the proposed system to outperform a standalone application for both C++ and Java clients. Let us remark that if a more powerful server would have been considered (and not run on the same computer than the client), the improvement would have been much higher. For a given client, the overall cost

[1] See `http://www.irisa.fr/obelix` for more details.

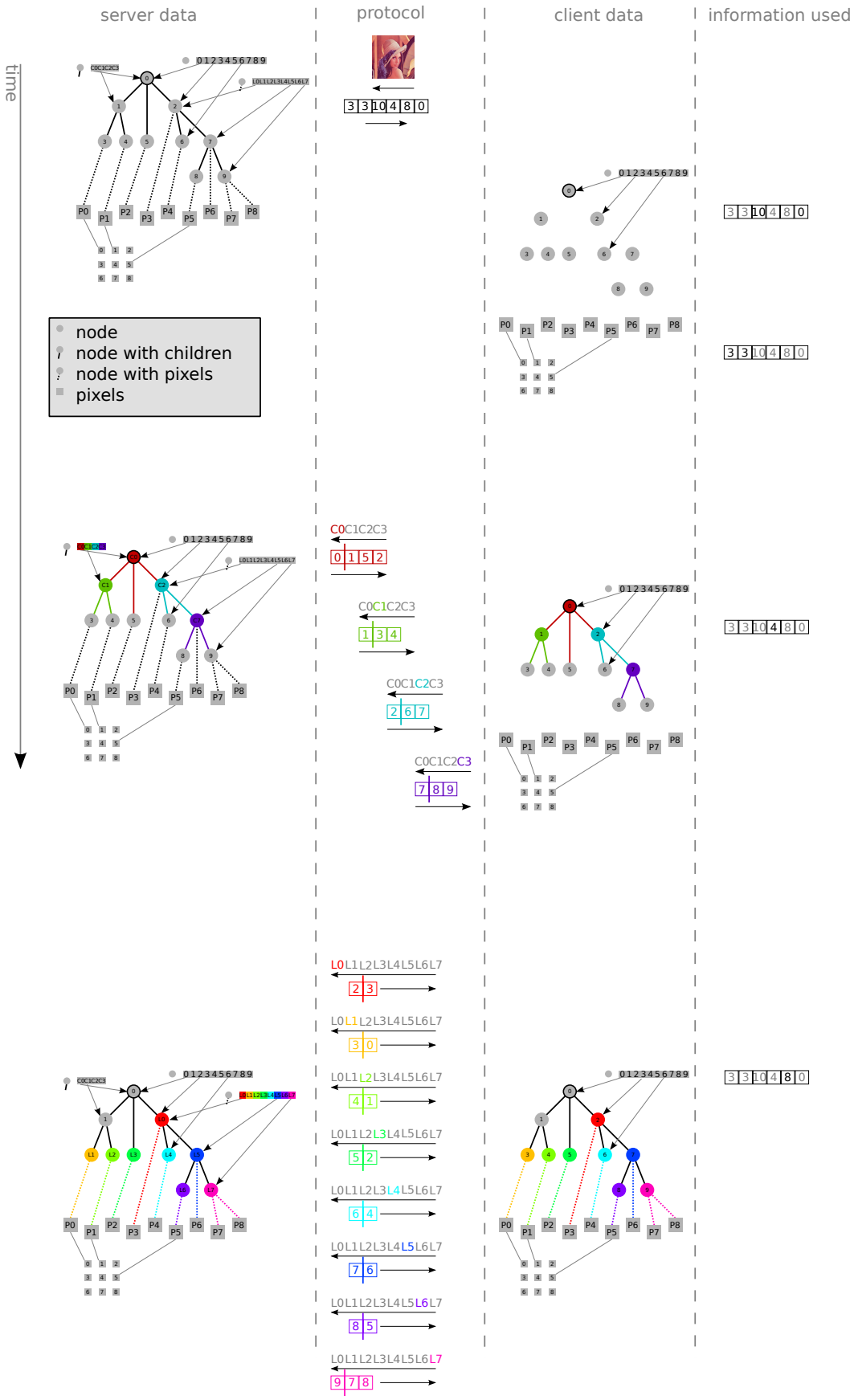

Fig. 3. Illustration of the communication process for tree transfer.

Method	$80k \times 4$	$200k \times 4$	$8M \times 1$	$13M \times 1$
full Java	0.40	0.85	22.11	—
C++ server	0.11	0.31	15.57	26.36
C++ client	0.05	0.13	4.26	7.05
C++/C++	**0.16**	**0.44**	**19.83**	**33.41**
Java client	0.31	0.31	6.84	19.87
C++/Java	**0.42**	**0.62**	**22.41**	**46.23**
Python client	1.03	2.88	99.64	—
C++/Python	1.14	3.19	115.21	—

Table 1. Comparison between standalone computation and the proposed client-server framework with various clients.

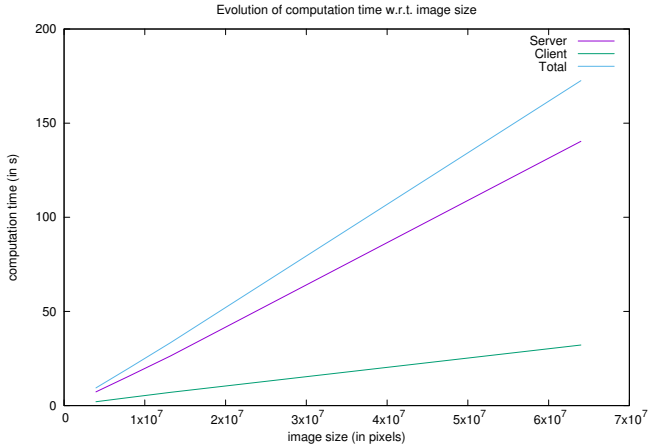

Fig. 4. Quantitative evaluation of the most efficient architecture (both server and client are in C++).

then tends to the client cost only.

Furthermore, we can notice some significant differences between the clients. Indeed, each client comes with its own memory management system, and its performance has a direct influence on the cost needed to create data structures on the client side. As such, we can see that Python might not be able to address very large images, even in this client-server mode. But let us note that for the largest image in consideration, the full Java application also fails due to memory limitations. Only the proposed framework, both with C++ and Java clients, is able to address the large image.

Finally, we have measured the computation time for both C++ server and client, for another set of satellite images with increasing size. We can see in Fig. 4 that the overall process has a linear complexity, and the client cost is always much lower than the server one. This shows the relevance of the proposed framework.

5. DISCUSSION

The proposed solution has appealing properties in terms of memory cost. As already indicated, a naive approach would have been to perform a full transfer of a tree through an XML representation. But the textual representation, as well as the data embedding through XML tags, would have led to a prohibitive cost. Byte coding appears as a relevant alternative. It is indeed possible to consider coding the tree in a proprietary binary format. Unfortunately, there is no native serialization scheme in all languages, and worse no interoperability between the existing serialization formats. Without serialization, byte coding can still be implemented to ensure the communication process between the client and the server. But this would also require an additional cost if interoperability is sought. Indeed, in this case the end-user aims to manipulate the tree in an environment different from the one on the server side. Thus a new tree structure is needed on the client side to map the tree sent from the server in a format compliant with the client environment. The proposed scheme (see Fig. 3) introduces node-based messages. Such messages have a size that is negligible w.r.t. the tree size, and can be removed as soon as they have been used on the client side to update the tree reconstruction. Since a tree representation might come with a very large memory footprint (e.g. in case of large images), the proposed strategy allows allocating most of the client memory to the storage of the tree with no significant overhead due to the server communication.

We have considered a complete transfer of the tree from the server to the client. If the latter is interested in only a part of the image, two different strategies can be followed (top-down and bottom-up). On the one side, the user can select a subtree, but it requires to identify the node of the tree acting as the subtree before transferring the subtree. While this second step is easily achievable with the framework proposed in this paper, performing node selection on a client-server paradigm is not trivial. On the other side, the selection can be done in a bottom-up way. In this case, the user defines the area of interest directly in the image, from which the related subtree (or forest, i.e. set of (sub)trees) is extracted. Among the available solutions for allowing such user selection, we can rely on our previous work [7] where the user input consists in a bounding box. The selected tree is then the largest subtree (or set of subtrees) totally included in the bounding box.

Future work will include demonstration of this framework within realistic remote sensing scenarios, as well as enrichment of the server component with additional tree construction algorithms.

6. REFERENCES

[1] G.K. Ouzounis, V. Syrris, L. Gueguen, and P. Soille, "The switchboard platform for interactive image information mining," in *Image Information Mining Conference*, 2012.

[2] S. Lefèvre, L. Chapel, and F. Merciol, "Hyperspectral image classification from multiscale description with constrained connectivity and metric learning," in *IEEE Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, 2014.

[3] F. Merciol and S. Lefèvre, "Fast building extraction by multiscale analysis of digital surface models," in *IEEE International Geoscience and Remote Sensing Symposium*, 2015, pp. 553–556.

[4] E. Aptoula, M. Dalla Mura, and S. Lefevre, "Vector attribute profiles for hyperspectral image classification," *IEEE Transactions on Geoscience and Remote Sensing*, 2016, to appear.

[5] J. Havel, F. Merciol, and S. Lefèvre, "Efficient schemes for computing α-tree representations," in *International Symposium on Mathematical Morphology*, 2013, pp. 111–122.

[6] E. Carlinet and T. Geraud, "A comparative review of component tree computation algorithms," *IEEE Transactions on Image Processing*, vol. 23, no. 9, pp. 3885–3895, 2014.

[7] F. Merciol and S. Lefèvre, "Buffering hierarchical representation of color video streams for interactive object selection," in *International Conference on Advanced Concepts for Intelligent Vision Systems*, 2015, pp. 864–875.