CHAPTER 3

LAR FRAMEWORK

As seen in the introductory chapters, technological advances in histology and imaging made it possible to acquire massive sets of 3D biomedical data. The collected knowledge has to be formalized, organized and combined in many ways. Furthermore, simulations and interactive explorations are needed. This chapter is aimed to introduce the Linear Algebraic Representation (LAR)[1] framework - developed by Prof. Paoluzzi and his team at Roma Tre University - and its approach to the problems of model extraction and visualisation starting from 3D medical images.

LAR is a general-purpose framework for solid and geometric modelling, based on a representation scheme[1] that uses Combinatorial Cell Complexes (CCC)[3] as its mathematical domain, and various compressed representations of sparse matrices as its codomain. The LAR framework provides then a wide set of operations, each based on the representation scheme.

LarVolumeToObj is a software module of the LAR framework designed to generate a 3D mesh from a stack of 2D images. Generally speaking, the structure of a *d*-dimensional image is mapped to a cellular complex of *(d-1)*-cuboids. The generated model can be smoothed and visualized.

3.1 LAR model

Definition 3.1. *A* LAR model *of a cell complex is either a pair* ***V,FV*** *or a triple* ***V,FV,EV****, where:*

1. ***V*** *is the list of vertices, given as a list of coordinates;*

2. ***FV*** *is a faces-vertex relation, given as a list of faces, where each face is given as a list of vertex indices;*

[1] A representation scheme is a mapping between the mathematical spaces to be represented by a computer system and their symbolic representation in computer memory.

3. ***EV*** *is an edge-vertex relation, given as a list of edges, where each edge is given as a list of vertex indices.*

Note that this is a *decompositive* representation of the *boundary*, in which the boundary is decomposed into *faces*, with face boundaries represented by a decomposition into *edges*, given as pairs of *vertices*. A *geometric model* is then a list of *cells* as unordered lists of vertex indices. [2]

In this and the next chapters, without loss of generality, will be often used a *python*-like notation to refer to the *list* structures, since this is the original notation defined by the authors of LAR. However, this same concepts can be easily adapted and implemented in different languages using similar data structures, as we'll see during the course of the treatment.

Example 3.1. The first example shows the LAR model of a standard 3-cuboid:

```
V=[[0,0,1],[1,0,1],[0,0,0],[1,0,0],[0,1,1],[1,1,1],[0,1,0],[1,1,0]]
FV=[[0,1,2,3],[0,1,4,5],[1,3,5,7],[2,3,6,7],[0,2,4,6],[4,5,6,7]]
EV=[[0,1],[1,3],[2,3],[0,2],[0,4],[1,5],[3,7],[2,6],[4,5],[5,7],[6,7],[4,6]]
CV=[0,1,2,3,4,5]
```

The visualization is given in fig. 3.1. The conventions for the axis are: red(x), green(y), blue(z).

1. In purple are shown the indices of the vertices, as defined in V and used to create FV;

2. In light blue are shown the indices of the edges, as defined in EV.

3. In green are shown the indices of the faces, as defined in FV and used to create CV.

3.2 Matrix representations

The binary *characteristic matrix* of the cell complex is efficiently retained in memory using some common matrix representations such as *Compressed Sparse Row (CSR)*, *Compressed Sparse Colums(CSC)*, *Coordinate representation (COO)* and *Binary Row Compressed (BRC)*. Obviously it's possible to *convert* the data from one representation to another using some simple library functions.

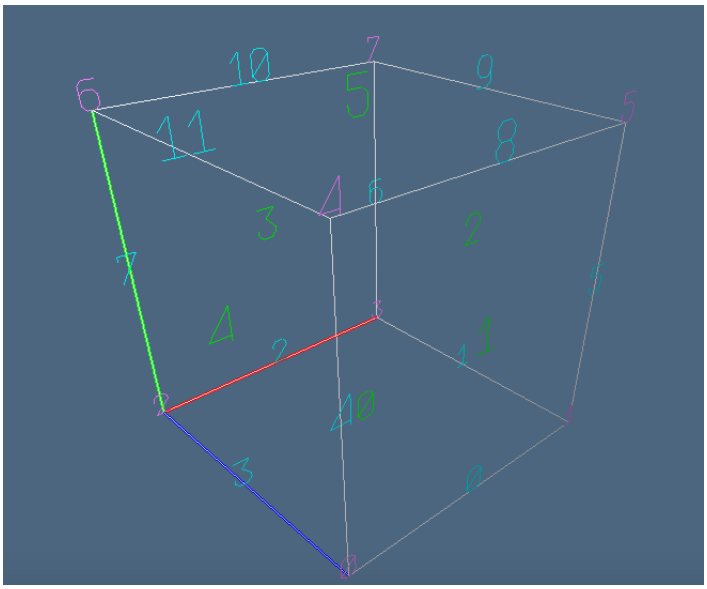


Figure 3.1. LAR model

3.2.1 Binary Row Compressed (BRC)

BRC is the standard input representation in LAR. It's used to represent the binary characteristic matrix, typically sparse. Each component array corresponds to a matrix row, and contains the indices of columns that store a 1 value. Zeros are not stored. In Example 3.1. the model was given in BRC.

Example 3.2. Here is presented the binary characteristic matrix M_3 for the standard cube and the equivalence to the BRC array is highlighted.

$$M_3 = \begin{bmatrix} 11110000 \\ 11001100 \\ 01010101 \\ 00110011 \\ 10101010 \\ 00001111 \end{bmatrix} \leftrightarrow FV = [[0,1,2,3],[0,1,4,5],[1,3,5,7],[2,3,6,7],[0,2,4,6],[4,5,6,7]]$$

3.3 Boundary and coboundary operators

There is a first distinction between:

- *Oriented* or *signed* boundary operator matrix $\Leftrightarrow \partial_{ij} \in \{+1, 0, -1\} \; \forall \; row \; i, \; column \; j$
- *Non-oriented* or *unsigned* boundary operator matrix $\Leftrightarrow \partial_{ij} \in \{0, +1\} \; \forall \; row \; i, \; column \; j$

The coboundary operator $[\delta_d]$ is in a duality relationship in respect of the boundary operator, since it allows the transition from a *(d - 1)*-dimensional complex to a d-dimensional one as shown in 3.2[6]:

$$\delta^d : C^d \to C^{d+1}$$

$$\partial_{d+1} : C_{d+1} \to C_d$$

This is the relation between boundary and coboundary operator is:

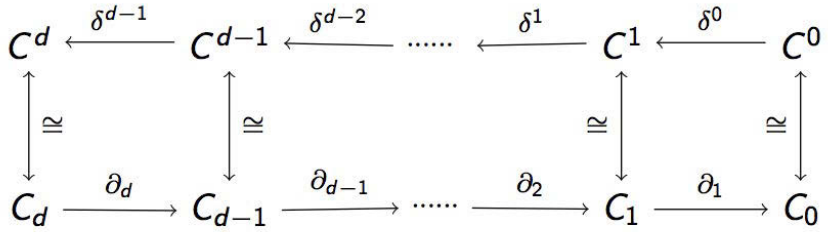


Figure 3.2. Boundary and coboundary operators.

$$[\delta_{d-1}] = [\partial_d]^T$$

For several computations, the knowledge of the matrices of non-oriented boundary operators is sufficient and much faster in terms of computing time.[2]

3.4 Geometric model extraction from 3D medical images using LAR: overview of the *LarVolumeToObj* package

The `LarVolumeToObj` package exposes basically three interfaces, that match with the three primary steps in the computation methodology: *data preparation*, model generation and visualization (see 3.3[5]). Here it will be given an overview of the entire process.

3.4.1 Data preparation

These are the main steps in data preparation:

- Acquisition of the 3D data - consisting of a stack of 2D images - into a 3D array.

[2]In the next part will be discussed examples of computations based on the different operators.

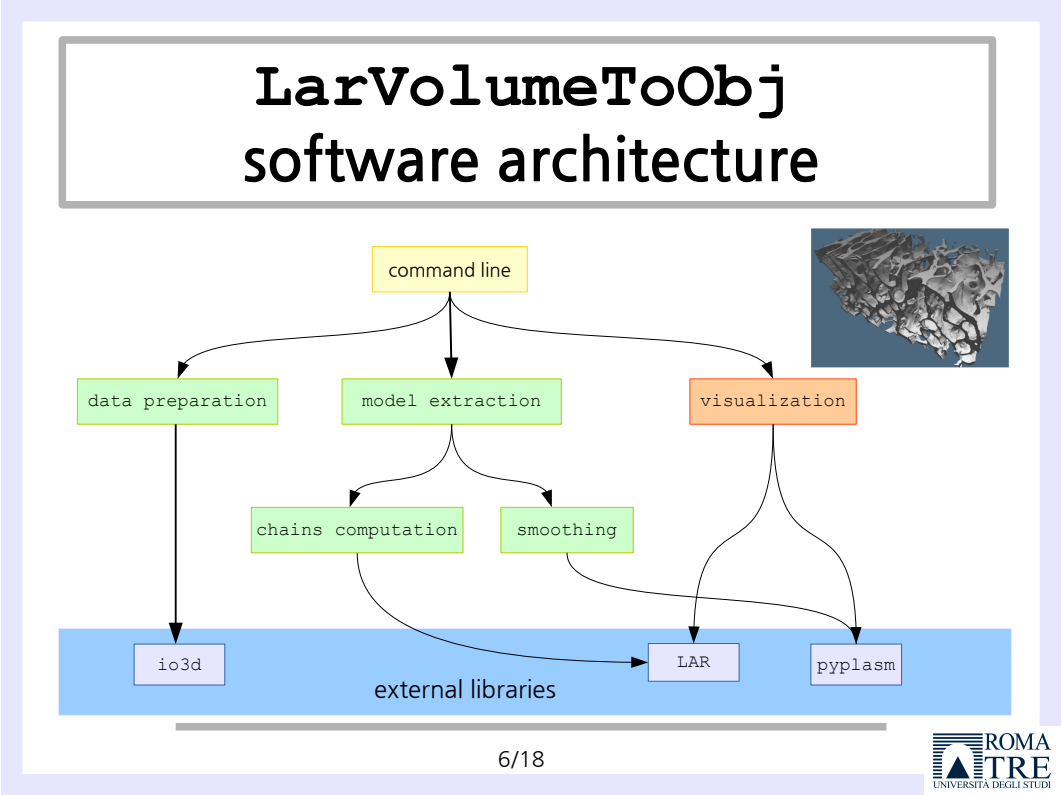


Figure 3.3. LarVolumeToObj package architecture.

- Possible crop of the data, to select a *volume* of interest.
- Application of a *threshold* on the *pixel intensity* values, in order to *clusterize* the data into two groups, the pixels that will belong and not belong to the model. This operation results in the data becoming a binary 3D matrix.
- Possible denoising via *median filter*[3].
- Use of *pklz*, a *python object serializator*, to retain the intermediate result.

The input/output operations, so basically the acquisition of the images and the writing of the data preparation outcome, are delegated to the `io3d` package (see 3.3[5]), developed by the same authors and able to manage the different image formats.

[3]In respect to the other traditional filters, the *median filter* is the one that better preserves the image edges. The main idea is to iterate over the pixels and replace each value with the median of the neighbors. The pattern of neighbors is called *window*. The window slides over the entire image, covering all the possible patterns. [8]

3.4.2 Model generation

- *Linearization* of the 3D binary data matrix to obtain a vector.
- Generation of the *oriented boundary operator* matrix $[\partial_3]$.
- Generation of a *basis*.
- Division of the volume of data into a *grid of voxels*, each of the size of $[\partial_3]$.
- *SpMV*[4] multiplication between $[\partial_3]$ and the data vector. This operation is performed in parallel on the voxels.
- *Filtering* of the basis on the base of the result of the multiplication. The outcome is the *boundary 2-chain*[5] of the model.
- Stitching of the voxels to obtain the complete model.
- Removal of the double facets, situated at the boundary of the voxels, through a dictionary based algorithm.
- *Laplacian smoothing*[6].
- Computation of the vertices for the faces belonging to the model.

---

[4] *Sparse Matrix-Vector*

[5] The *d-chains* are sets of *d-cells*

[6] The *smoothing* in a mesh is aimed to compute a new position for each vertex according to local information, here the coordinates of the neighbors. This procedure can be repeated many times, to smooth even more the resulting mesh. In *laplacian smoothing*, for each vertex i, the new position is updated as follows:

$$\bar{x}_i = \frac{1}{N} \sum_{j=1}^{N} \bar{x}_j$$

where N is the number of adjacent vertices to vertex i, $\bar{x}_j$ is the position of the j-th adjacent vertex and $\bar{x}_i$ is the new position for i. Thanks to LAR, to compute the set of the adjacent vertices for each vertex that's enough to derive the VV relation. [8]

Example 3.3. For the cube in the example 3.1.: `VV = [[1, 2, 4], [0, 3, 5], [0, 3, 6], [1, 2, 7], [0, 5, 6], [1, 4, 7], [2, 4, 7], [3, 5, 6]]`

- Final retain into the `wavefront obj` format[7].

The core of the algorithm lies in the SpMV multiplication between the boundary operator and the data vector aimed to obtain the boundary model or 2-chain from the 3-chain. This part is delegated to the LAR library. The smoothing algorithm instead is implemented using `pyplasm` as support library.

3.4.3 Visualization

The outcome of model extraction is a 3D mesh of quads representing the geometric model of the boundary of the structures of interest (the boundary 2-complex). Note that this is an *enumerative* schema. The quads can be easily triangulated to achieve visualization. To conclude the computation methodology, the final steps are:

- Triangulation of the quads.
- `pyplasm`[8] visualization.

---

[7]Note the strong analogy between the LAR model, made of the couple $< V, FV >$ and the *obj* format, in which the list of vertices and the list of faces - expressed in terms of vertex indices - are represented in the following way:

Example 3.4. Geometric model of the standard cube:

```
v 0 0 1
v 1 0 1
v 0 0 0
v 1 0 0
v 0 1 1
v 1 1 1
v 0 1 0
v 1 1 0
f 3 4 2 1
f 1 2 6 5
f 2 4 8 6
f 7 8 4 3
f 5 7 3 1
f 5 6 8 7
```

Only few points should be took into consideration: first, while the numbering of the indices starts from 0 in LAR, it starts from 1 in the obj format. Second, while in LAR the order of the indices with which the faces are specified is not relevant, since the FV relation is retained as a matrix, on the other side, in the obj format one should traverse and annotate the indices of vertices in a specified order. Furthermore, the direction of traversal can influence the normals and the visualizations depending of the viewer, as will be discussed in the next part.

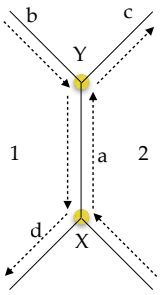
[8]*Programming LAnguage for Solid Modeling*: https://github.com/plasm-language/pyplasm

Since the *obj* is an open and widely adopted format, obviously you can visualize the final model using your favourite viewer, for instance `paraview`.

3.4.4 Compactness of representation

The amount of memory required to store a LAR model is compared to the well known *Baumgart's scheme*[9][4] in this article[2]:

> The common reference term for comparing the memory requirement of solid boundary representations in 3D is the *Winged-Edge* scheme by Baumgart, which makes use of relation tables with a storage occupancy $8|E| + |V| + |F|$, where F, E, V stand for the sets of boundary faces, edges and vertices, respectively. An equivalent LAR representation of topology of the boundary of a 3D solid (*B-rep*) needs only the storage of the $CSR(M_2)$ sparse matrix, corresponding to the FV incidence relation, and the computation of the $CSR(M_1)$ sparse matrix, to obtain the EV relation, for a total memory size of $2|E| + 2|E|$.



Edge	Vertices		Faces		Left Traverse		Right Traverse	
Name	Start	End	Left	Right	Pred	Succ	Pred	Succ
a	X	Y	1	2	b	d	e	a

Figure 3.4. Winged-edge representation

[9] Assumed that there aren't holes in the faces, consider the vertices, edges and faces of a polyhedron (see 3.4). In the *winged-edge* representation, for each edge, the following information are retained:

- its vertices;
- its left and right faces;
- predecessor and successor when traversing in clockwise ordering (viewing from outside of the polyhedron) its left face;
- predecessor and successor when traversing in clockwise ordering (viewing from outside of the polyhedron) its right face.

3.4.5 Parallelization and performances

`LarVolumeToObj` has been implemented in *python* using the following libraries, intended to improve performances: `Scipy`, `Numpy` and `Cython`.

As previously said, parallel programming is achieved in `LarVolumeToObj` partitioning the data volume into a *cuboidal grid* (fig. 3.5) made of *voxels*. This part of the problem is *embarassingly parallel*[10]: then it has been possible to divide the data into independent subsets, that are processed in parallel. In this phase, the SpMV multiplication between $[\partial_3]$ and the data vector is performed as well as the filtering of the basis. This is the core of the computation and is linear in the size of the output. To summarize, in this first phase, the boundary model is obtained by *decomposition* into a *queue* of independent tasks, in a *multiprocessing* context.

Note that the basis and the boundary operator of a certain size are the same for each voxel and for each computation, so a unique boundary operator is generated offline and written permanently.

Figure 3.5. Parallelization idea.

Smoothing is performed on the complete model after the stitching of the voxels and hasn't been parallelised.

3.4.6 Previous applications

Previous applications of LAR in the biomedical field include the study of the hepatic portal system, conducted in collaboration with the *University of West Bohemia* and the

[10] In *parallel computing*, an *embarrassingly parallel* problem is characterized by the fact that little or no effort is needed to separate the problem into a number of parallel tasks. This is often the case where there is little or no dependency or need for communication between those parallel tasks, or for results between them.

Charles University. The porcine hepatic microvessels were injected with *Biodur E20* resin, and the resulting corrosion casts were scanned with 1.9-4.7 μm resolution. [7] The imaging technique was *SEM*. The italian team focused on one of the datasets, made by 994 *DICOM* images with sizes 992×1013 and resolution 4.6823μm in each dimension. From this dataset has been obtained a model with size $370 \times 228 \times 237$ through `LarVolumeToObj`[2]. Further results have been achieved in [8]: the project, developed by Eng. Salvati, concerned the implementation of a fast and parallel *julia* version of the library to be tested on a *24 core cluster* belonging to *Istituto Nazionale Fisica Nucleare (INFN)*(see fig. 3.6)[11]. The biggest model extracted was of size $992 \times 1012 \times 100$.

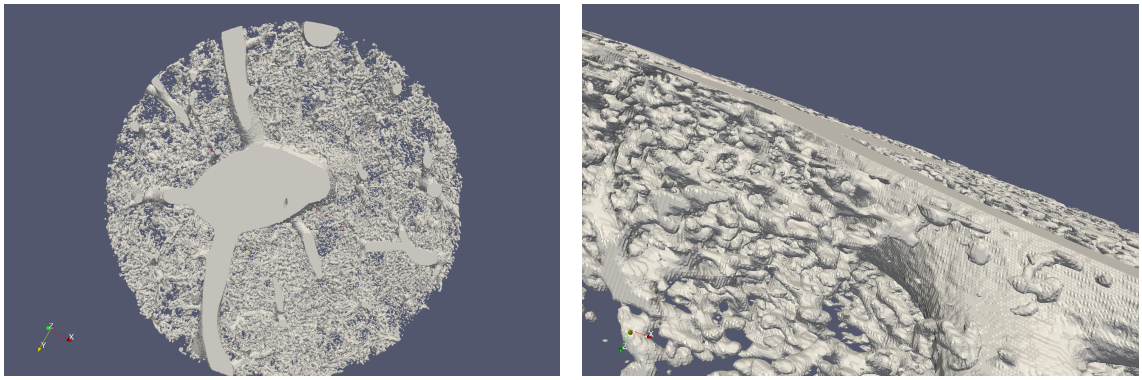


Figure 3.6. Liver portal vein system.

3.5 Summary and conclusions

LAR is a general-purpose framework for solid and geometric modelling, based on a representation scheme that uses cell complexes as its mathematical domain, and various compressed representations of sparse matrices as its codomain. The LAR framework provides then a wide set of operations, each based on the representation scheme. LAR includes also the `LarVolumeToObj` prototype, that has been used in different applications to extract geometric models from medical images. However, it has never been tested on massive datasets. Further results in this way has been achieved by a completelly parallel julia version of the library, that relays on a 24 cores cluster. The aim of the thesis is to try

[11] National institute of nuclear physics, Italy

to obtain similar results in respect of the julia implementation, but without the need for a supercomputer, opening the research results to a wider range of scientists.

3.6 References

[1] V. S. A. DiCarlo and A. Paoluzzi, *Linear algebraic representation for topological structures*, Computer-Aided Design, 46 (2014), pp. 269–274.

[2] F. F. M. J. A. Paoluzzi, A. Di Carlo, *Cad models from medical images using lar*, CAD15, (2015).

[3] T. Basak, *Combinatorial cell complexes and poincar duality*, Geometriae Dedicata, 147 (2010), pp. 357–387.

[4] B. G. Baumgart, *Winged edge polyhedron representation*, (1972).

[5] G. Clementi, *Geometric model extraction from 3d medical data*.

[6] A. F.Furiani, C.Paoluzzi, *Algebraic extraction of models and properties from images*.

[7] A. K. L. E. H. M. P. K. T. G. P. H. M. S. E. R. M. K. V. L. Miroslav Jirik, Zbynek Tonar, *Stereological quantification of microvessels using semiautomated evaluation of x-ray microtomography of hepatic vascular corrosion casts*, CARS2016, (2016).

[8] D. Salvati, *Models from massive biomedical images via parallel computational topology*, thesis, Roma Tre University, 2016.

CHAPTER 4

VISUS FRAMEWORK

The ViSUS visualization framework, developed by prof. Pascucci and his team at the University of Utah, allows on-the-fly interactive exploration and visualization of massive datasets, afferent to a wide range of scientific disciplines including *neuroscience*, for which provides a viewer and a neurotracker. The framework has been designed to *scale* on a wide variety of hardware, from mobile devices to powerwall displays.

In the course of this chapter will be exposed some key aspects of ViSUS that are particularly significant with reference to this thesis, that is aimed, among other goals, to critically analyse the ViSUS approach and to investigate the possibility of an integration solution between the LAR and the ViSUS frameworks.

4.1 Overview of the ViSUS software framework

The ViSUS infrastructure is composed of three major components, that here are briefly introduced:

- *data management* framework: regards the management of dataset and data storage, and data access.

- *dataflow* framework: concerns the processing of the data and the management of interactivity, real-time exploration and movement.

- Visualization layer: provides a viewer, based on OpenGL, offering standard visualizations such as slicing, *volume rendering*[1] and *isosurfacing*.

[1] *Volume rendering* is a set of techniques used to display a 2D projection of a 3D *scalar field*, a discrete volumetric grid.

4.2 Data access layer

When dealing with HPC and Big-Data applications, the first issue to be faced is *I/O cost*.[2]

This problem is brilliantly solved in ViSUS through *out-of-core*[3], *data streaming*[4] algorithms, where each *stream* has an associated level of resolution in the hierarchy.[5]

The solution relies on cache and a compressed disk file format with associated *metadata*. Compression is provided by zlib. The use of the cache is quite standard: a block number is queried to the cache. If the block is in the cache, the sample is accessed and returned, otherwise, an asynchronous I/O operation for that block is added to an I/O queue and the point is marked as pending. The read compressed block from disk is decompressed into the cache, so timings for acquisition t_a need to consider for each frame disk access time for reading t_r and data decompression time t_d: $t_a = t_r + t_d$.

The grid data, stored in secondary memory, has been reorganized to maximize *data locality* - exploiting the *cache hierarchies* of data storage architectures - and minimize the I/O operations, on the base of the *data access patterns* in the algorithms to be executed on the data.

---

[2]

> Even if you can speed up the computational aspects of a processor infinitely fast, you still must load and store the data and instructions to and from a memory. Today's processors continue to creep ever closer to infinitely fast processing. Memory performance is increasing at a much slower rate. Many of the interesting problems in HPC use a large amount of memory. As computers are getting faster, the size of problems they tend to operate on also goes up. The trouble is that when you want to solve these problems at high speeds, you need a memory system that is large, yet at the same time fast: a big challenge. *C. Severance* [1]

[3] *Out-of-core computation* deals with large data-structures that don't fit in the main memory of a single computer and try to adapt the computations to operate mainly on the small loaded parts.

> Real time processing of very large volumetric meshes introduces specific algorithmic challenges due to the impossibility of fitting the input data in the main memory of computer. [...] The performance of most algorithms does not scale well in the transition from in-core to the out-of-core or in external memory. The performance degradation is due to the high frequency of I/O operations that may start dominating the overall running time. *V. Pascucci* [6]

[4] *Data streaming* algorithms are designed to address the issue of the limited memory available -much less than the input size- and limited processing time. The input is presented as a continuous sequence of *data streams*(items). Streams can be denoted as an ordered sequence of points (or updates) that must be accessed in order and can be read only once or a small number of times. The resulting representation is in form of a vector:

$$\overrightarrow{a} = (a_1, a_2, ..., a_n)$$

.

To achieve real-time visualization and interaction, a *multi-resolution* approach has been set up: starting from an overview of the dataset, progressively, as soon as new data have been read and processed, the visualization in updated offering ever higher *level of detail*. This avoids the system to be stalled by the amount of data.

To summarize, the dataset is reorganized in a *hierarchical* way and the data are read from *coarse* to *fine* resolution and are provided in streaming-mode, allowing for *progressive*[5] update of the output data structures and visualization, derived from this data. Data at the same level of resolution are stored in close proximity in memory and accessed at the same time.

This is the only possible solution, since *precomputation* is infeasible, as explained here:

> Many of the parameters for interaction, such as display viewpoint, are determined by users at run time and therefore precomputing this levels of detail optimized for specific queries is infeasible. *V. Pascucci* [5]

4.2.1 Lebesgue's space filling curve

The reorganization of the dataset is achieved through the use of a hierarchical variant of a *Lebesgue space filling curve* (fig. 4.1) - commonly referred as *HZ (Hierarchical Z) order* - and without data replication. The conversion from the *Z-order*, adopted in classical databases, to the `IDX` format can be accomplished with simple sequence of bit-string manipulation. [5] Within the hierarchy, at each level, only the new data are stored (fig. 4.2). As can be seen in fig. 4.2, the i^{th} level of resolution is defined as the curve obtained by replacing the vertices of the $(i-1)^{th}$ level of resolution with Z shapes of size $(1/2)^i$. The 3D version of this space filling curve has the same hierarchical structure with the only difference that the basic Z shape is replaced by a connected pair of Z shapes lying on the opposite faces of the cube (fig. 4.1(f)). [6]

To accomplish a new static hierarchical indexing, a mapping from the 3D index(i,j,k) of an element in the grid, expressed in row-major order[6] to a 1D index I on the disk has to be

---

[5] *Progressive* and *streaming* are often used synonymously in *computer graphics*. [2]

[6] *Row-major* and *column-major order* are methods for storing multidimensional arrays in linear storage. For example, for

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

we have, in *Row-major order*:

provided. [6] In this way is generated a translation from data in cartesian coordinates to a one-dimensional array. [3]

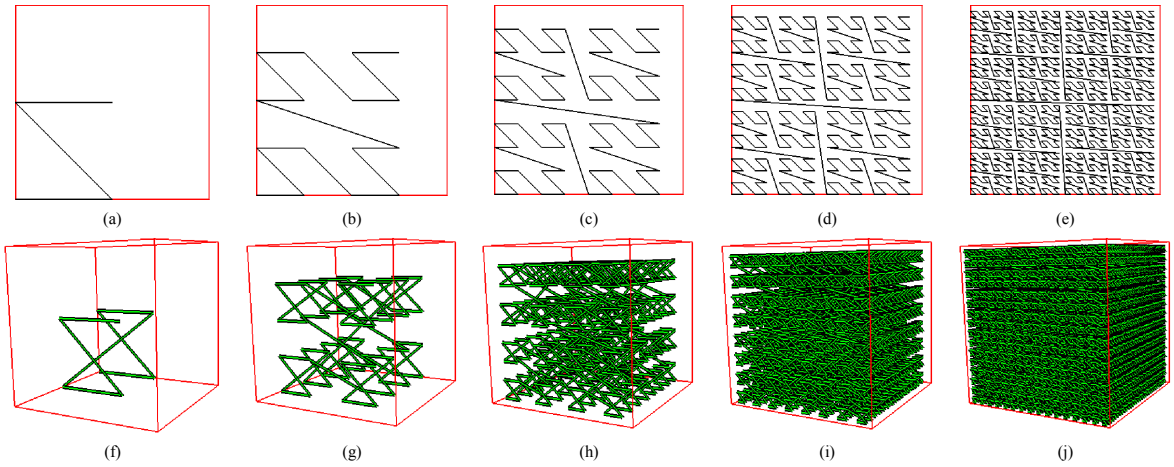


Figure 4.1. Lebesgue's space filling curve.

4.2.2 IDX format

To summarize, IDX is a format for visualizing large-scale HPC simulation results because of its ability to access multiple levels of resolution with low latency time for interactive exploration. HZ ordering is the key idea behind the IDX format. With each increasing level of resolution, the number of elements grows by a 2 factor. IDX blocks are

Address	Access	Value
0	A[0][0]	a_{11}
1	A[0][1]	a_{12}
2	A[0][2]	a_{13}
3	A[1][0]	a_{21}
4	A[1][1]	a_{22}
5	A[1][2]	a_{23}

and in *Column-major order*:

Address	Access	Value
1	A(1,1)	a_{11}
2	A(2,1)	a_{21}
3	A(1,2)	a_{12}
4	A(2,2)	a_{22}
5	A(1,3)	a_{13}
6	A(2,3)	a_{23}

Row-major order is used in C/C++ and `Numpy`. Column-major order is used in Fortran, `OpenGL` and `OpenGL ES`, Julia and `Eigen`.

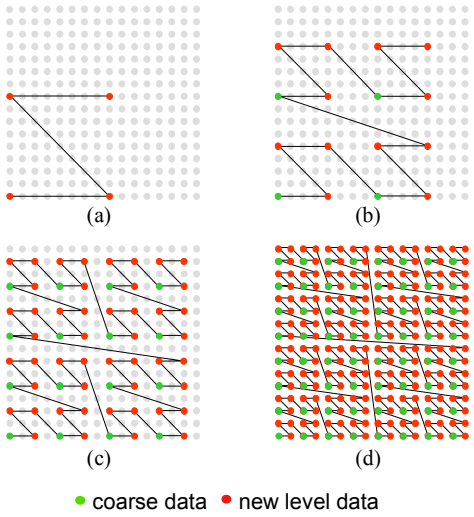


Figure 4.2. Data streaming

grouped together into a collection of files, which are organized into a directory hierarchy and accompanied by a metadata file describing the various properties of the data. [3]

4.2.3 Parallel IDX (PIDX)

The *Parallel IDX* `PIDX` library provides efficient methods for accessing and writing standard IDX datasets by using a parallel MPI based API. It's able to manage both *regular* and *irregular* volumes, that is volumes having power-of-2 dimensions (e.g. 32^3) or not. In fig. 4.3 is depicted an example of 8×8 2D regular grid. The colors correspond to the division of the data between 4 processes. Numbers in the cells stand for HZ indices of the highest resolution while empty cells stand for lower-resolution data. The 1D disk layout for the grid is below.

4.3 Progressive isocontouring and streaming meshes

4.3.1 Introduction

Until now, a progressive data-structure has been presented, giving a hierarchical representation of the data at multiple levels of detail. The next step is the introduction of progressive algorithms, that is techniques for *on-line* construction and smoothing of pro-

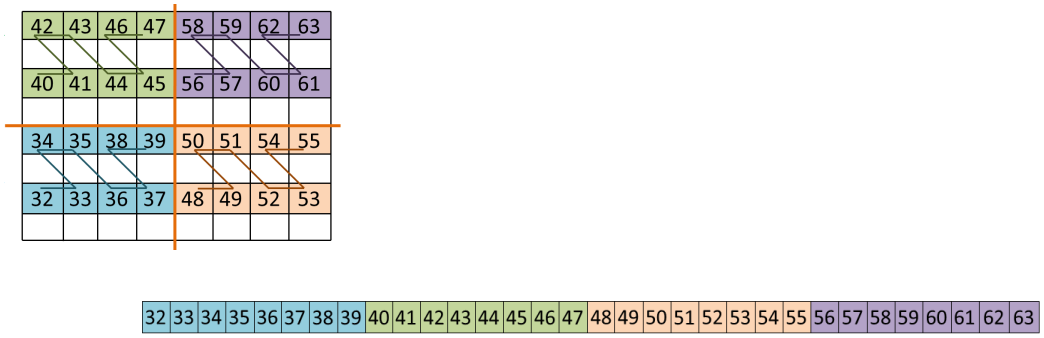


Figure 4.3. PIDX mapping schema.

gressive *isosurfaces*[7]. In ViSUS the visualized meshes are computed as isocontours of 3D scalar fields starting from volumetric input.

Definition 4.1. *An* isocontour *or* isosurface C(w) *of a scalar field* $\mathcal{F}(x)$ *is the locus of points where the field has a value* w:

$$C(w) = \{x | \mathcal{F}(x) - w = 0\}$$

The output data structures to be rendered are expressed in terms of multi-resolution streaming meshes, generated by a progressive algorithm. The output is built by several refinement steps, that update a portion of the output meshes, followed by smoothing.

4.3.2 Marching cubes

The basic isocontour computation algorithm is *marching cube*, introduced in 1987 by Lorensen and Cline in the SIGGRAPH journal. This first formulation is known to be inefficient due to the waste of time in exploring the empty regions of the volumetric data. Until now, many optimization techniques have been set up:

[7]Informally, an *isosurface* is a 3D surface that represents points of a constant value within a volume of space; in other words, it is a level set of a continuous function whose domain is 3D-space. Isosurfaces are an important form of visualization for volume datasets since they can be rendered by a simple polygonal model, which can be drawn on the screen very quickly.

4.4 Topological analysis

As seen before, topological analysis enables the detection and extraction of features from the data leading to generation of abstract models that are orders of magnitude smaller than the raw data. In fig. 4.4 is shown the ViSUS analysis, based on *Morse theory*[8] and visualization pypeline starting from the raw data. [4]

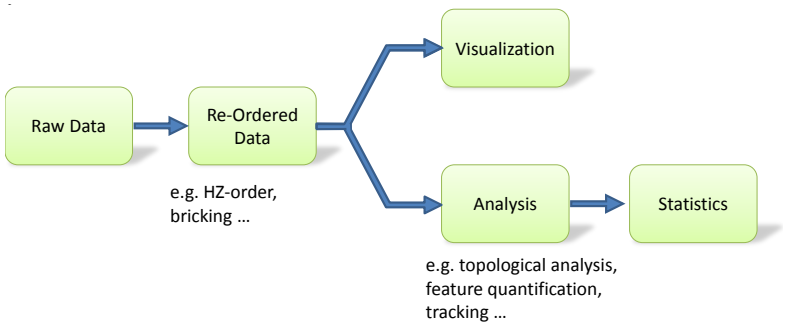


Figure 4.4. ViSUS analysis and visualization pypeline.

4.5 Computation of reeb graphs

4.6 Summary and conclusions

4.7 References

[1] K. D. C. Severance, *High Performance Computing*, CONNEXIONS, Rice University, Houston, Texas, 2011.

[2] P. L. M. Isenburg, *Streaming meshes*.

[3] P. H. C. J. A. L. R. L. G. S. H. K. R. W. G. R. B. R. M. E. P. J. C. V. P. Sidharth Kumar, Venkatram Vishwanath, *Efficient data restructuring and aggregation for i/o acceleration in pidx*, SC Conference for High Performance Computing, Networking, Storage, and Analysis, (2012).

[4] A. G. G. S. C. C. B. S. S. K. V. Pascucci, P.-T. Bremer, *Scalable Visualization and Interactive Analysis Using Massive Data Streams*, vol. 23, Cloud Computing and Big Data, Advances in Parallel Computing, IOS Press.

[8] *Morse theory* enables the analysis of the topology of a manifold by studying differentiable functions on that manifold. According to the basic insights of Marston Morse, a typical differentiable function on a manifold will reflect the topology quite directly. Morse theory allows one to find CW structures and handle decompositions on manifolds.

[5] B. S. P. B. A. G. C. S. P. S. K. V. Pascucci, G. Scorzelli, *Chapter 19 The ViSUS Visualization Framework*, Chapman and Hall/CRC Computational Science, 2012.

[6] R. J. F. V. Pascucci, *Global static indexing for real-time exploration of very large regular grids*, SuperComputing 2001, (2001).

PART III

DESIGN OF SOLUTION

PART IV

APPLICATION

BINOMIAL NOMENCLATURE INDEX

SOFTWARE INDEX

TOPIC INDEX