Parallel & Distibuted Computing: Lecture 18

Alberto Paoluzzi

November 13, 2019

Program B for individual projects

1. Extraction of boundary surface from 3D medical imaging
2. Parallel workflow
3. Browsing the code

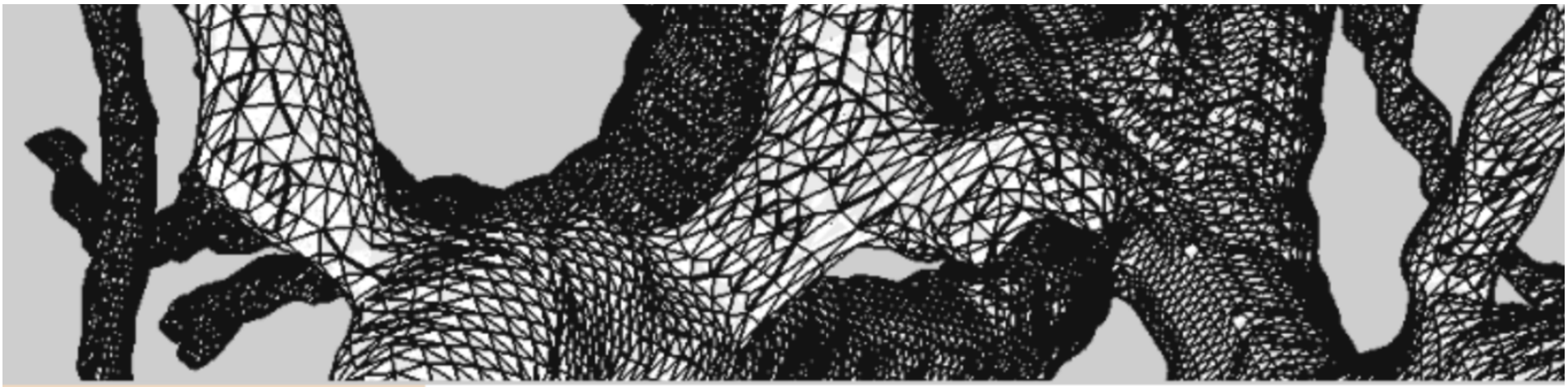
Extraction of boundary surface from 3D medical imaging

Computer modeling for research on liver perfusion

Department of Surgery and Biomedical Center, Faculty of Medicine in Pilsen, Charles University

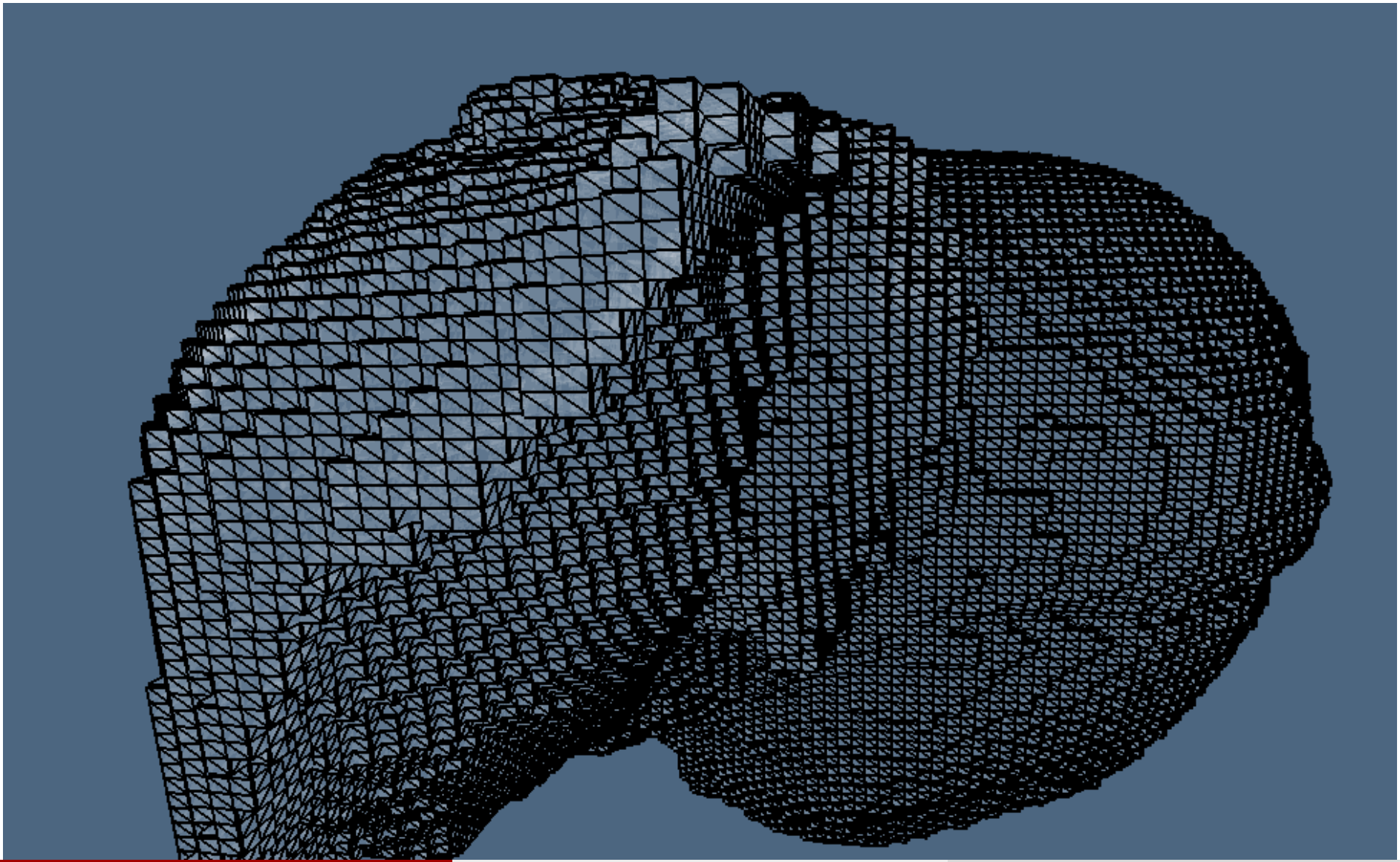
Joint project with

Department of Mathematics and Physics, Roma Tre University



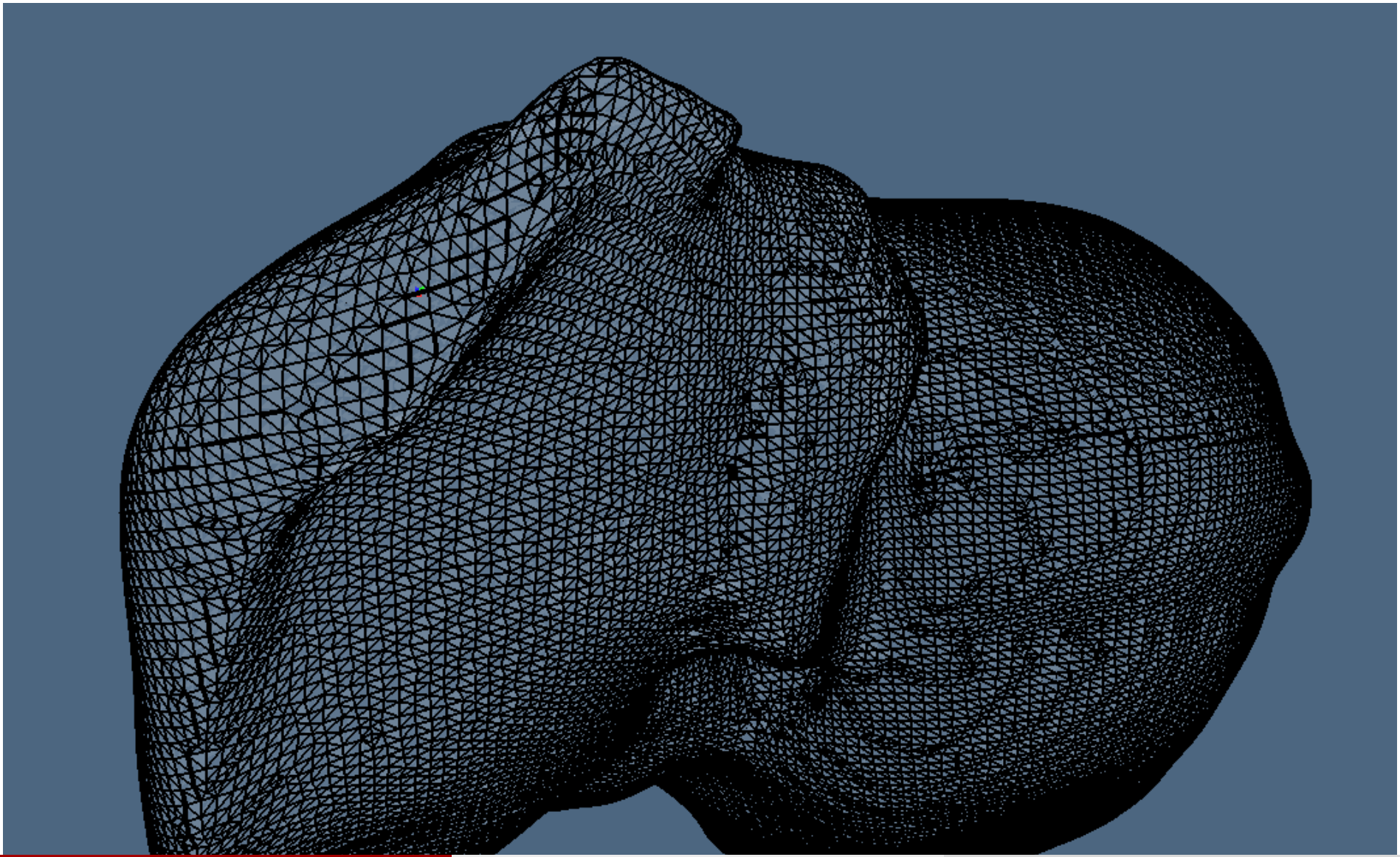
Liver digital surface

Triangulated surface extracted by LAR from 3D digital image

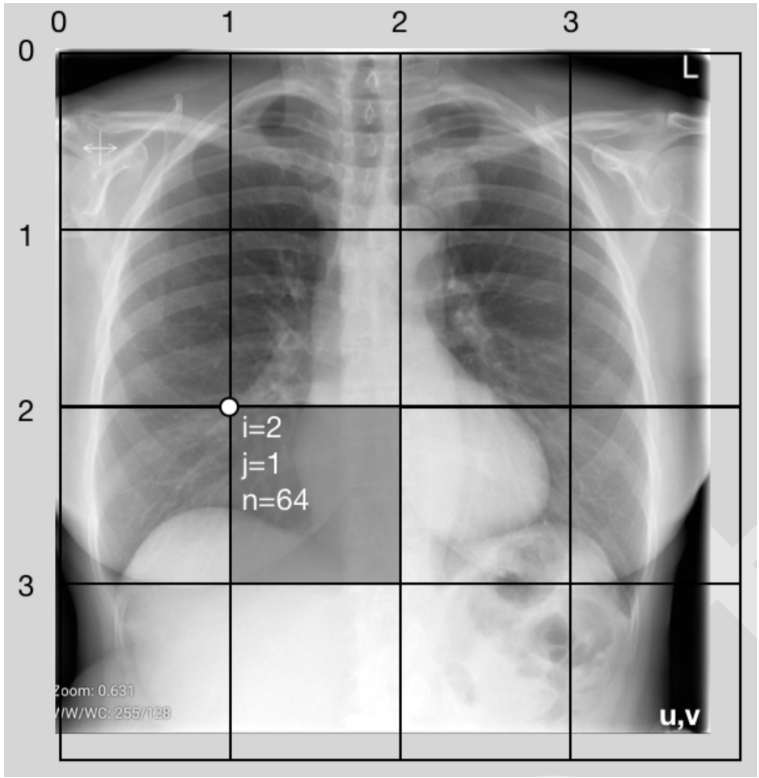


Smoothed liver surface

Smoothing by Taubin method



Block decomposition



A possible block partitioning of a radiologic image. The evidenced 2D block, of size $n^d = 64^2$, is sliced by $\mathbf{B}([2, 1, 64]) = \mathit{Image}([128 : 172], [64 : 128)]$

Linear index from Cartesian index

The special `CartesianIndex{N}` object represents a scalar index that behaves like an N-tuple of integers spanning multiple dimensions.

For example:

```
julia> A = reshape(1:32, 4, 4, 2);

julia> A[3, 2, 1]
7

julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7
true
```

https://docs.julialang.org/en/v1/manual/arrays/#Cartesian-indices-1

Construction of boundary matrix $\partial_3^\top = \delta_2$

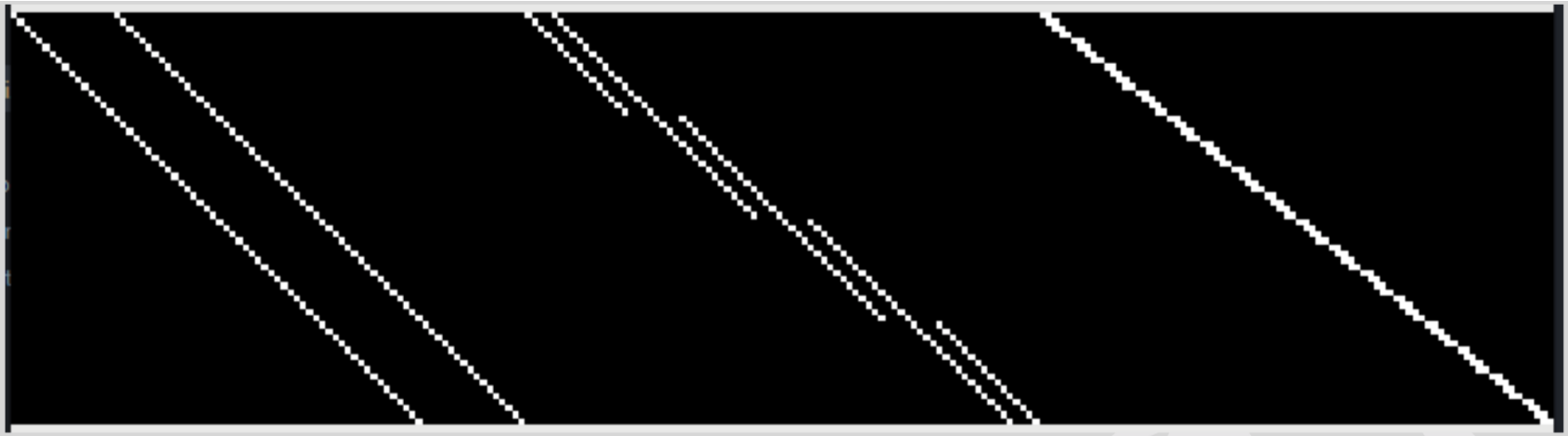
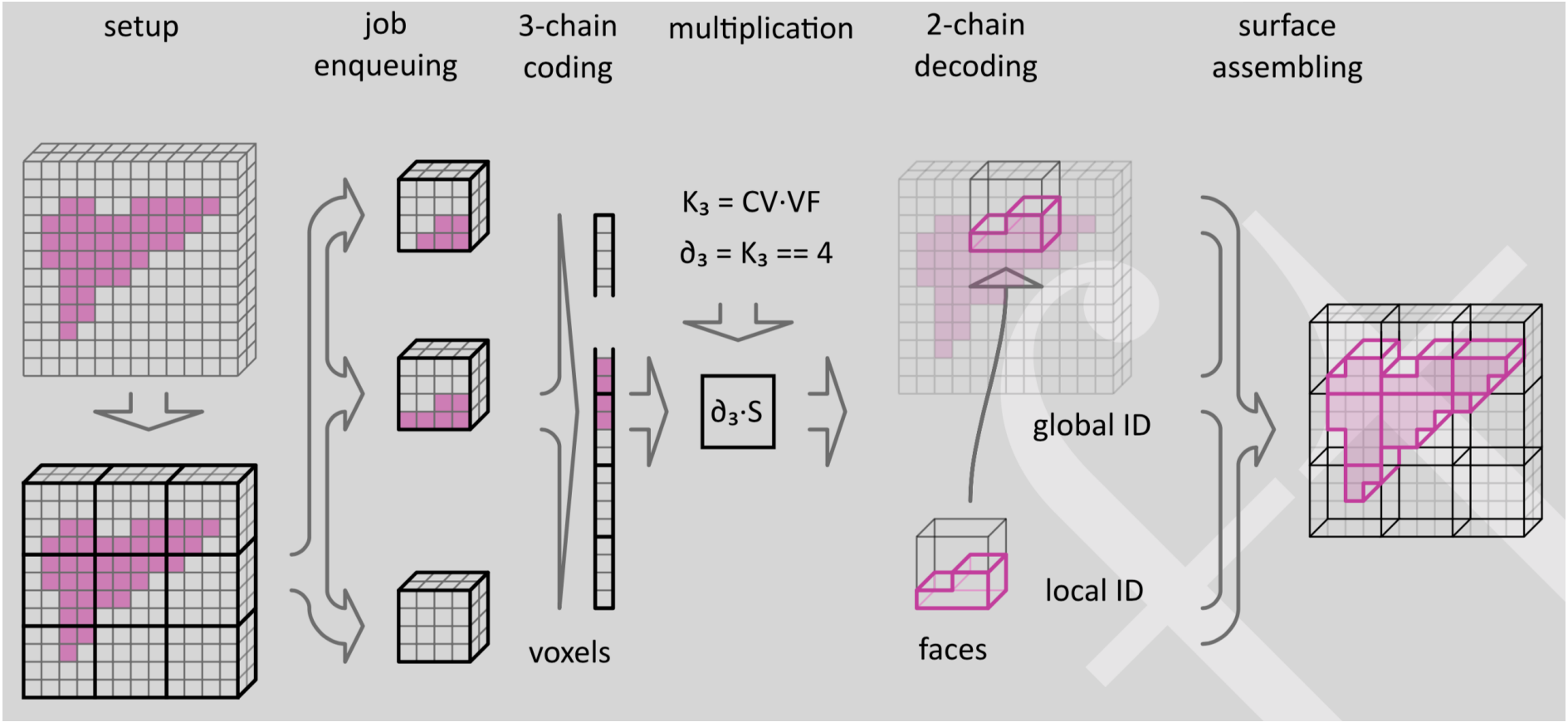


FIGURE 1. A binary image of the coboundary operator $\delta_2 = \partial_3^\top : C_2 \to C_3$, built for a small 3D image with shape $(4, 4, 4)$. Note that the number of rows equates the cardinality $4 \times 4 \times 4 = 64$ of the voxel set; the number of columns is $d\, n\, (1 + n)^{d-1} = 3 \times 4 \times 25 = 300$. Of course, the number of non-zeros per row (cardinality of the facet set of a single voxel) is six, whereas the number of non-zeros per column is two, but on boundary facets.

Parallel workflow

Workflow



Workflow of Lar-surf algorithm

Workflow setup

1. input of 3D medical image $\mathcal{I}$ with shape (ℓ_1, ℓ_2, ℓ_3), such that: $\mathcal{I} = [\ell_1] \times [\ell_2] \times [\ell_3]$, where $[\ell_k] = [1, 2, \ldots, \ell_k]$;
2. analysis of resources available in the computational environment, including operating system, type and number of compute nodes (processors, cores, GPUs), number of cores per node, RAM and caches amounts;
3. depending on the above, best decision for size of 3D image block (or brick) $\mathcal{B}$. With defaults $size = 64$, the number of bricks will be $n = \lceil \ell_1/size \rceil \times \lceil \ell_2/size \rceil \times \lceil \ell_3/size \rceil$. Hence the default number of bricks is $n = 256$, for standard images $512 \times 512 \times 256$;
4. computation of the Julia's sparse boundary matrix $[\partial_B]$, returning a value of type `SparseMatrixCSC{Int8}{Int64}`, where `Int8` and `Int64` are the types for values and indices, respectively, stored by Compressed Sparse Column (CSC) format; the storage of $[\partial_B]$ (for $size = 64$) requires about 45 MB;
5. creation of either a local or distributed `channel` to implement a producer/consumer model of parallel/distributed computation, depending on available resources;
6. distribution of matrix $[\partial_B]$, of default size 45 MB, to all available nodes/cores (Julia workers) using the Julia macro `@eveywhere`.

Job enqueuing

Encoding subsegments as 3-chain

Communication and data synchronization may be managed through Channels, FIFO conduits that may provide producer/consumer communication.

Overall execution time can be improved if other tasks can be run while a task is being executed, or while waiting for an external service/function to complete.

The work items of this stage:

1. extraction of block views from image array, depending on $([i, j, k], n)$;
2. transform each block from global $[\ell_1] \times [\ell_2] \times [\ell_3]$ to local $[n] \times [n] \times [n]$ coordinates;
3. encode the sub-segment as 3-chain by transforming the block data $[\nu] \subseteq \mathcal{S} \subseteq \mathcal{I}$ from Cartesian to linear coordinates, using Julia's library functions.
4. enqueing the job (as a sequence of integer positions for the non-zeros image elements aligned in a memory buffer of proper `Channel` type).

3-Chain encoding

1. each encoding task produces either a full or sparse binary vector. With full or sparse arrays depending by one index, we get either 262 KB or less per job;
2. special format for sparse CSC (Compressed Sparse Column) vectors can be used, since the value data for non-zeros does not need storage. Hence only a single 1-array of `Int64` row positions (with total length equal to the number of non-zeros in the block, with $8 \times \texttt{nnz}$ kB storage) is needed;
3. prepare such data vectors (non-zero linear row indices), in order to feed efficiently the available processors or threads.

In case of presence of one/more GPUs, a smaller size of the block—and hence of the boundary matrix and the encoded 3-chain vectors—and then much higher vector numbers, might be preferable for speed (to be tested)

SpMM Multiplication

Various multiplication algorithms are being tested, using packages for sparse linear algebra and/or custom implementations;

1. the total speed of this stage will strongly depend on the hardware available, on the granularity of blocks, and on the choice between dense/sparse storage of encoded 3-chains;
2. anyway, the compute nodes or threads will be fed without solution of continuity in a dataflow process.

This parallel operation is, according to our preliminary experiments, the critical one of the whole workflow, in the sense that any ΔT (either positive or negative) in this stage will contribute to the total time T.

2-Chain decoding

Each multiplication of $[\partial_B] : C_3 \to C_2$, times a 3-chain $[\nu] \in C_3$, produces a 2-chain $[\sigma] \in C_2$, i.e.~the coordinate representation of the boundary vector $\sigma \in C_2$.

The inverse of the coding algorithm is executed in the present stage. This process can also be executed sequentially with the previous ones on workers, depending on the size of the memory buffers used to feed the CPU cores or the GPUs and get their results.

Some elementary steps follow:

1. conversion from position of ones (i.e., non-zeros) in the 2-chain to linear indices of rows;
2. conversion from linear indices to Cartesian indices in local coordinates of the $\mathcal{B}$ block, using library functions;
3. conversion from Cartesian indexvalue to a suitably oriented (i.e.~with proper attitude) geometry quadrilateral (or to pair of triangles) in local coordinates.

Assembling and artifact filtering 1/2

The previous stages produce a collection of sets of geometric quadrilaterals (quads), each one encoded as an array of quadruples of integer indices, pointing to a `Float64` 2-array of grid vertices associated to the image block $\mathcal{B}$.

In other words, all quads of each job are now given in the same global coordinates.

Besides to put each partial surface $\mathcal{S}(B) = (\mathtt{V}_B, \mathtt{FV}_\sigma)$ in the global coordinate system of the image, the present stage must eliminate the redundant boundary cells generated at the block boundaries of the partial surface $\mathcal{S}(B)$ within each block $\mathcal{B}$ such that $B \cap \mathcal{I} \neq \emptyset$:

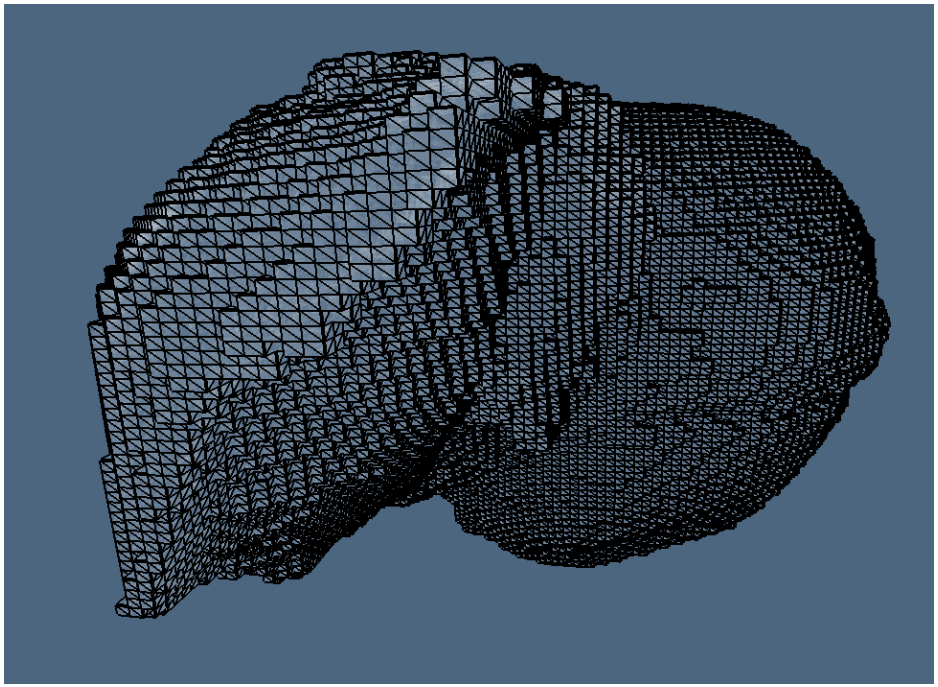
Remarks

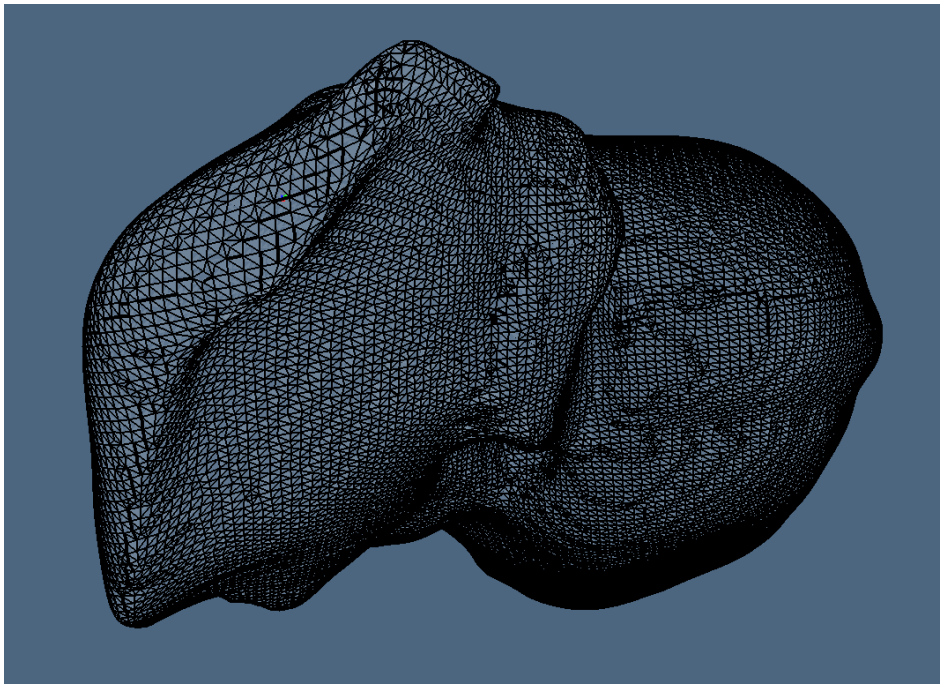
All computations concern the topological structure denoted by $[\partial_3]$.
None computation modifies the `Int64` coordinates of vertices, generated implicitly.

Assembling and artifact filtering 2/2

1. translate each array $\texttt{FV}_\sigma$, of type `Lar.Cells`, by summing to each vertex index the linearized offset of the Cartesian coordinates$(n, m, p)(B)$ of the $\mathcal{B}$'s reference vertex, i.e.~the one with (all) lowest Cartesian coordinates within the $\mathcal{B}$ block.
2. remove both instances of double quads generated by `Lar` software at the block boundaries (see Figure~**??**). They are artifacts generated by the decomposition of the whole image into a number of blocks of tractable size. (In canonical format they are double and equal!!)
3. a smart strategy of removal of such artifacts (by not enabling their generation) may be used, which does not require any sorting nor searching on the assembled array of quads. It will consist in arranging each block with all three dimensions decreased-increased by one, so that each 2-adjacent pair of blocks will be covering each other for a full side extent of blocks of depth one. The details of this artifact filtering are elucidated in Section~**??**.

Taubin smoothing





- First presentation of method
  Curve and surface smoothing without shrinkage
- More readable and general article, with pseudocode
  Geometric Signal Processing on Polygonal Meshes

Browsing the code

Github LarSurf repository

https://github.com/mjirik/LarSurf.jl

aaaaaa

aaaaaa

aaaaaa