Smart Slicing with Q.Blender

ROADMAP

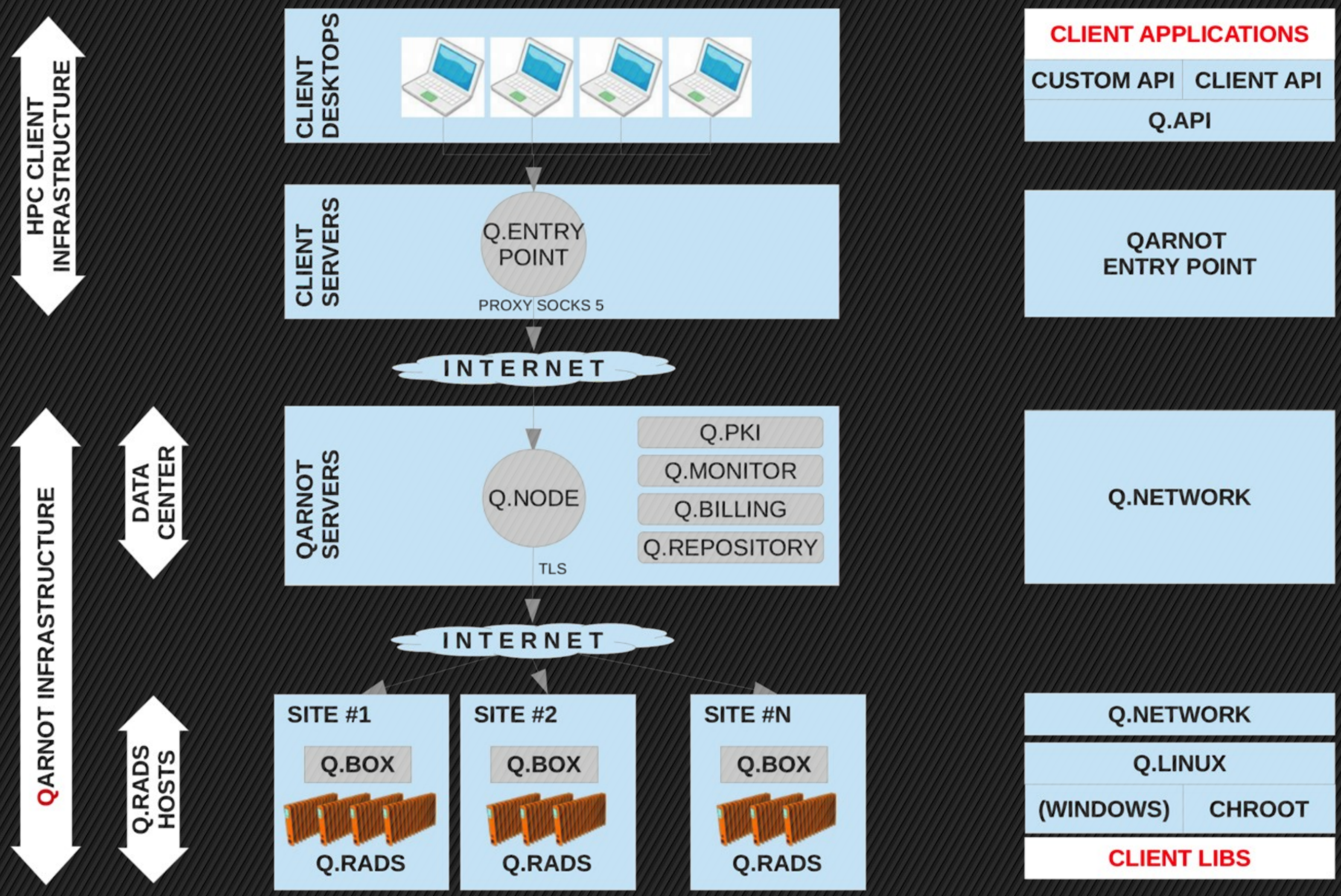
- Part I: Q.Blender
- Part II: Smart Slicing in Q.Blender

Part I: Q.Blender

DUAL SIDED BUSINESS MODEL

ARCHITECTURE

END-TO-END OVERVIEW

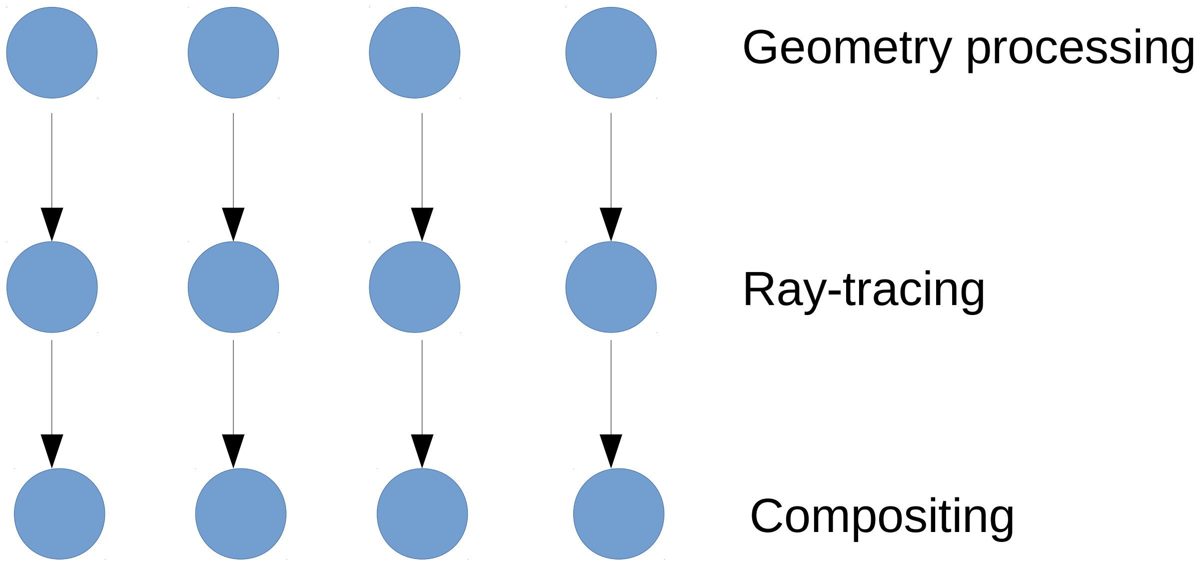


Python SDK for distributed rendering

```python
import qapy
import sys

print("Loading config...")
api = qapy.QApy('qarnot.conf')

print("Creating task...")
with api.create_task("My Blender Job", "blender", 1) as task:
     print("Sync resources from 'input' directory")
     task.resources.sync_directory("input/")

     print("Setting constants...")
     task.constants['BLEND_FILE'] = "model.blend"
     task.constants['BLEND_FORMAT'] = "PNG"
     task.constants['BLEND_ENGINE'] = "CYCLES"
     task.constants['BLEND_CYCLES_SAMPLES'] = 1000

     print("Submitting task...")
     task.submit()

     print("Waiting for task completion...")
     task.wait()

     print("Retrieving results in 'output' directory")
     if not os.path.exists("output/"):
             os.makedirs("output/")
     task.download_results("output/")
```

Part II: The smart slicing problem

Q.Blender parallel workflow



In Q.Blender we use a sort-first parallelization: the geometry processing task is duplicated between the nodes. In the ray-tracing phase, each node works on a sub-region of the image and the compositing task consists of joining the sub-images

Why sort-first ?

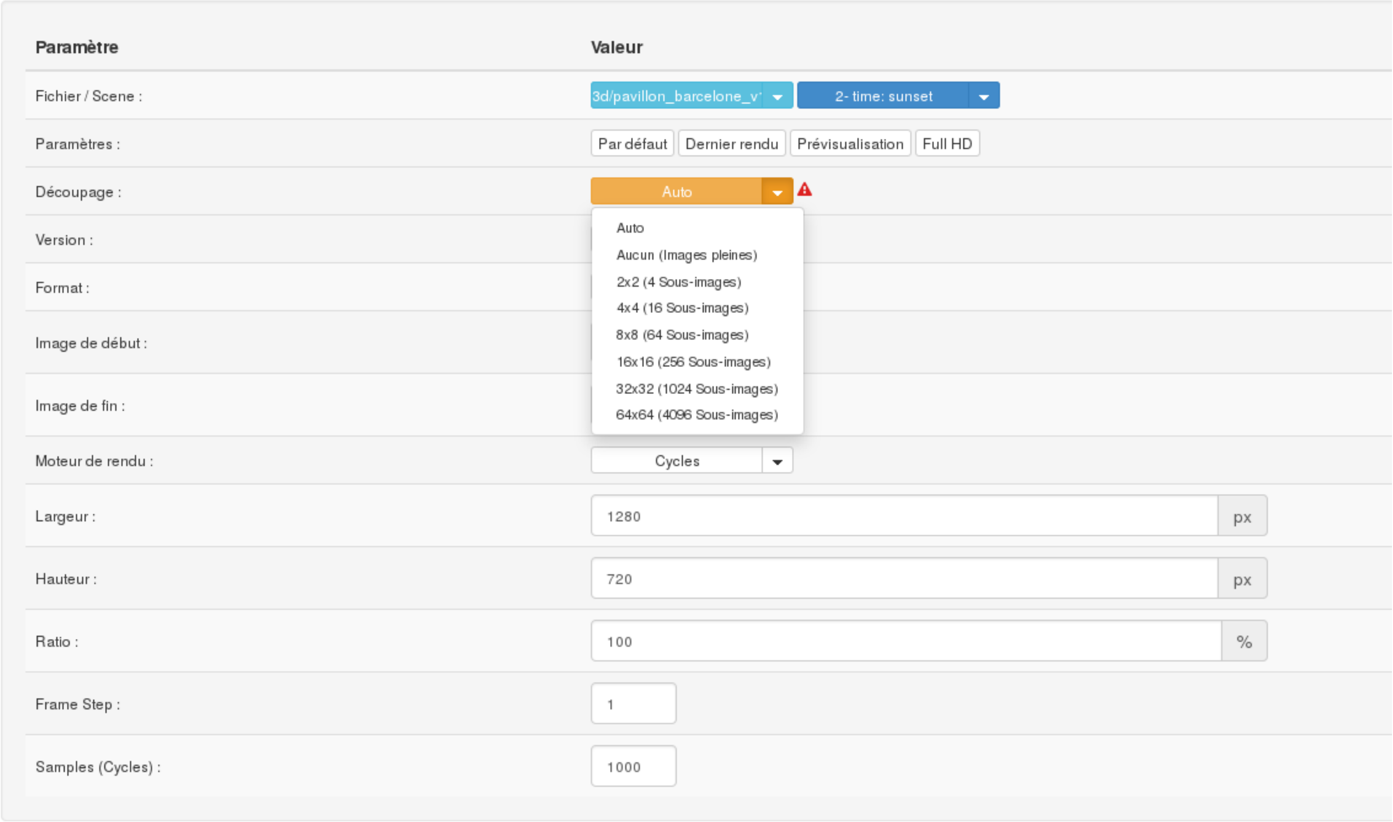
- **Bad point:** duplicated work in geometry processing
- **Good point:** We avoid expensive communication costs in a geo-distributed context

Creation of parallelism

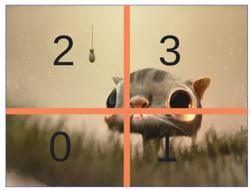
- The decomposition in region is based on **rectangle** tiling
- Q.Blender supports various generic decompositions (2*2, 4*4 ….)

Goal

- Improve the « auto» mode



Computational Problem



$t0 = 1516s$ $t1 = 1947s$
$t2 = 408s$ $t3 = 713s$

$T0 = 1947s$

CPus: 4

Rendering time of a region: ti

$t0 = 912s$ $t1 = 1012s$
$t2 = 1721s$ $t3 = 939s$

$T1 = 1721s$

Total rendering time:

$Ti = \max\{ti\}$

$t0 = 1698s$ $t1 = 1765s$
$t2 = 788s$ $t3 = 333s$

$T2 = 1765s$

Question: How to tile an image for parallel processing such as to minimize Ti? Two sub-problems:

- What is the processing time on the different «regions» of an image ?
- How do we tile the image given the runtime of each region?

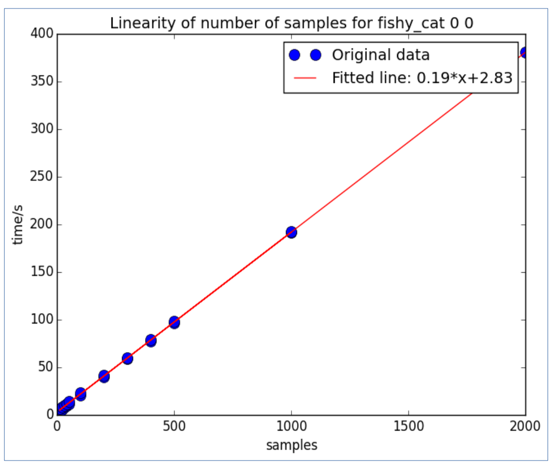
What impacts the load ?

- Assumptions :
  - #Sample
  - Ratio
  - # cells (octree, polygons ..)
  - #frames
  - Pixel intensity ...

Load Estimation

Approach: Process the image with a small number of samples and ratio and project the measured runtime (we assume an initial fine and squared tiling in regions)

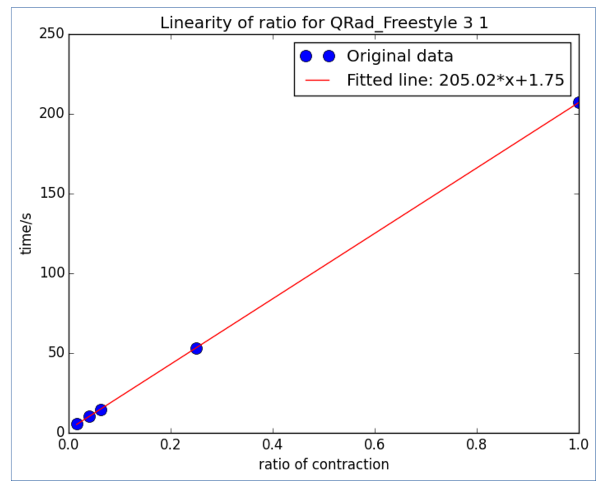
Main result: Linear regression of the runtime on the #number of samples and on the ratio



x: number of sambles

y: rendering time

Example of a linear regression



x: ratio

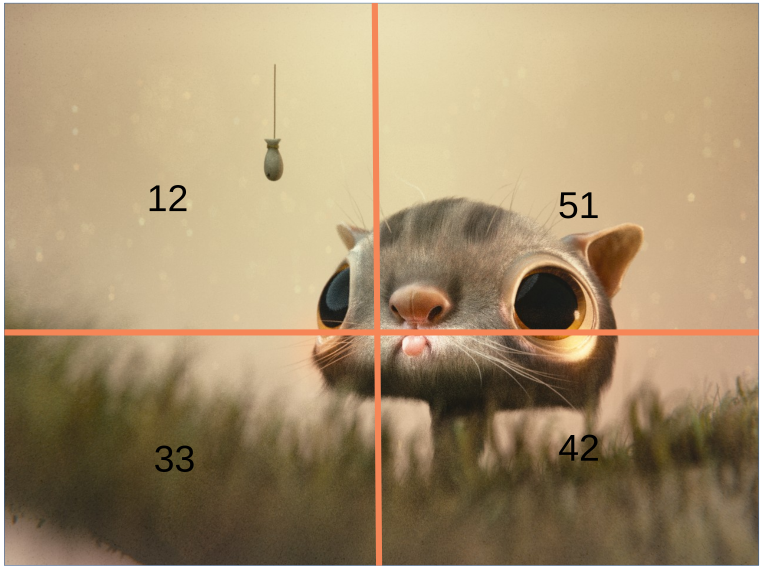
y: rendering time

A regression was found for more than 96 images. The constant term is very small

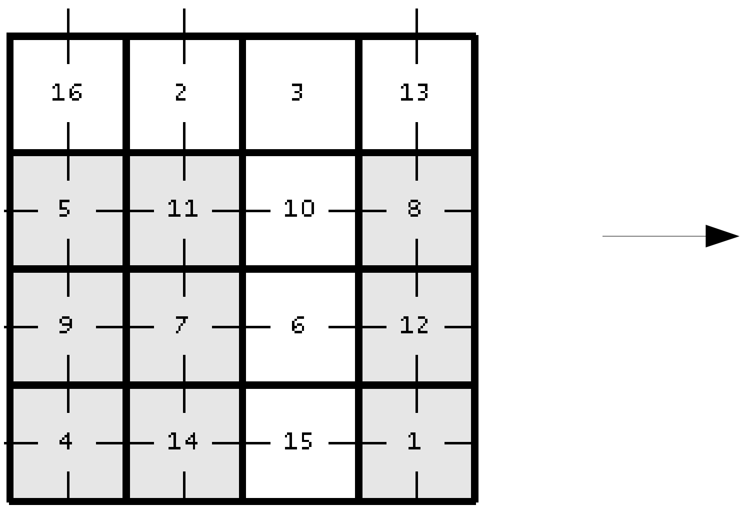
We considered more than 80 images.

Load estimation (2)

- With few samples, we can have a *good* estimation of the runtime difference between regions of the image



So What is the problem ?

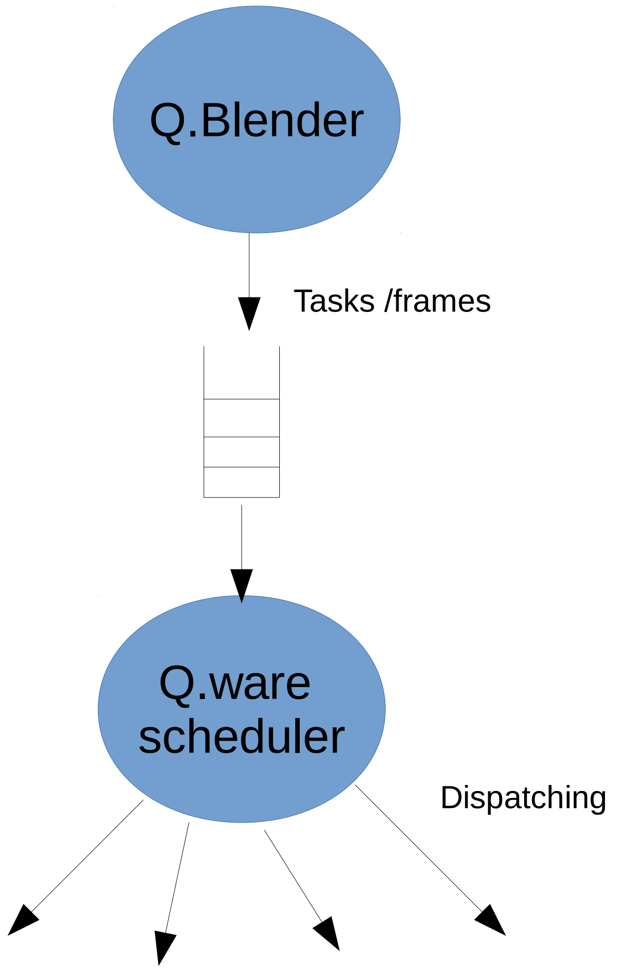


- Assuming a grid with g*g « unit » frames of different load, how to share this load between M processors ?

A proposition

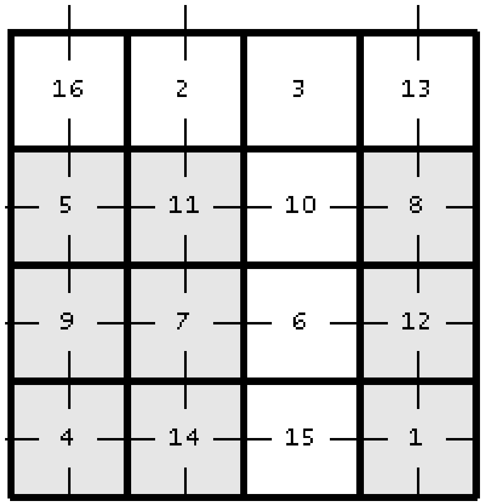
- Assign the « unit » frames such as to reduce imbalance
- **M-partition problem**, NP-hard but there are several good heuristics and semi-exact algorithms

But...



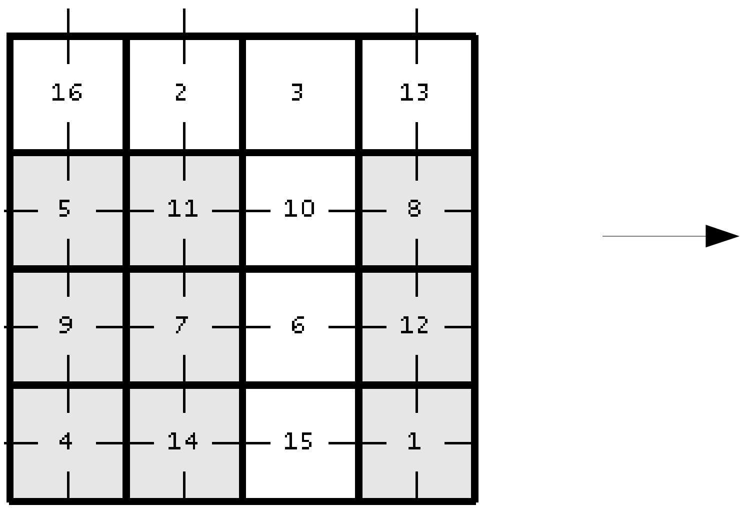
Resource oblivious SaaS logic: Q.Blender interacts with the general cloud resource manager: it cannot explicitely decide on the processor that will compute a given frame

So what can we do ?



Make sure that the imbalance between the frames submitted for processing to the resource manager is not too high

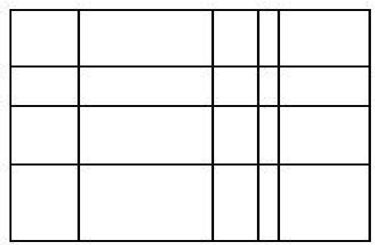
So What is finally the problem ?



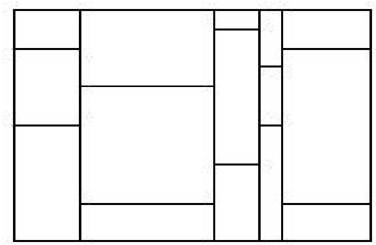
- Create M rectangles such as to minimize Lmax-Lmin. Here, Lmax is the load of the most heavy rectangle. Lmin is the load for the less heavy. M could be set as the number of processors.

Old problem

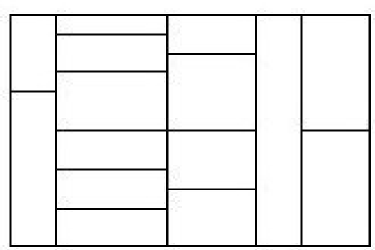
- NP-hard. See https://arxiv.org/abs/1104.2566



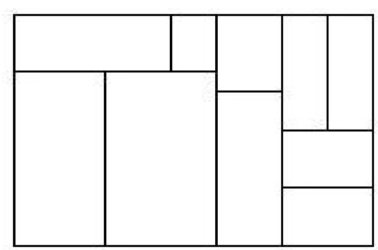
(a) A (5×4) rectilinear partition



(b) A $P \times Q$-way (5×3) jagged partition



(c) A m-way (15) jagged partition

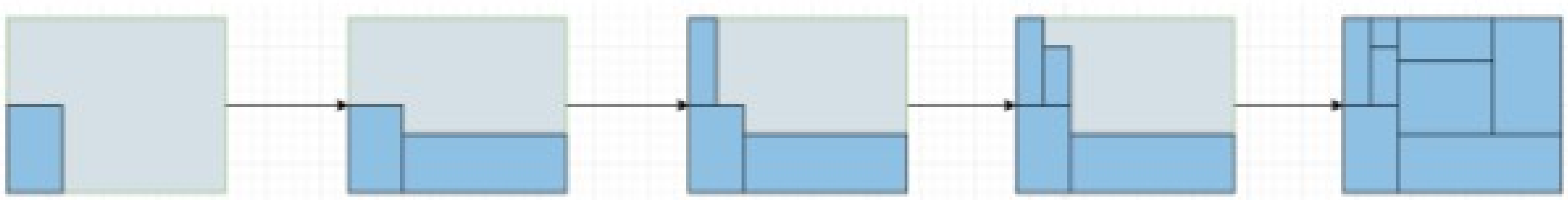


(d) A hierarchical bipartition

- Objective: a more efficient partitioning that only requires the number of rectangles as input.

Proposition

- A greedy «rectangle stacking » algorithm that iteratively chooses a rectangle whose load is close to the average of the area that was not already partitioned.
- Works like brick laying (layer by layer)
- No brick is placed in vaccum



Proposition(2)

Runtime

T1 = 1005s

T2 = 1011s

T3 = 996s

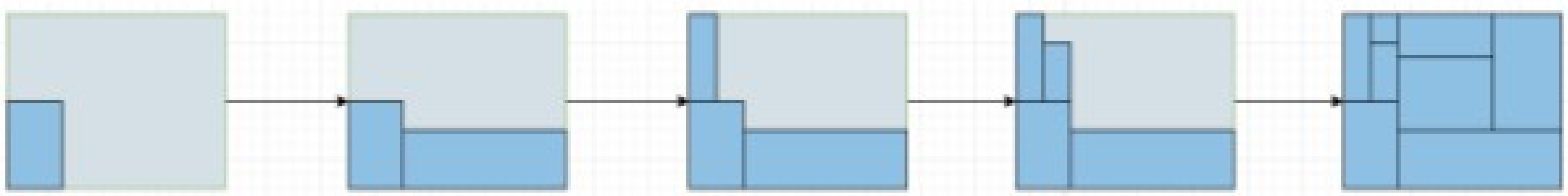
T4 = 952s

T5 = 971s

We gain more than 600 seconds over our traditional tiling solution.

Analysis

- The choice of the first tile is critical. It almost causes a 1D partitioning
- The rectangles become « more vertical » as we increase layers.
- The algorithm is not optimal



Open challenges

- Comparative analysis with related approaches
- Integration in the current «auto» mode
- Improvement of the rectangle stacking algorithm