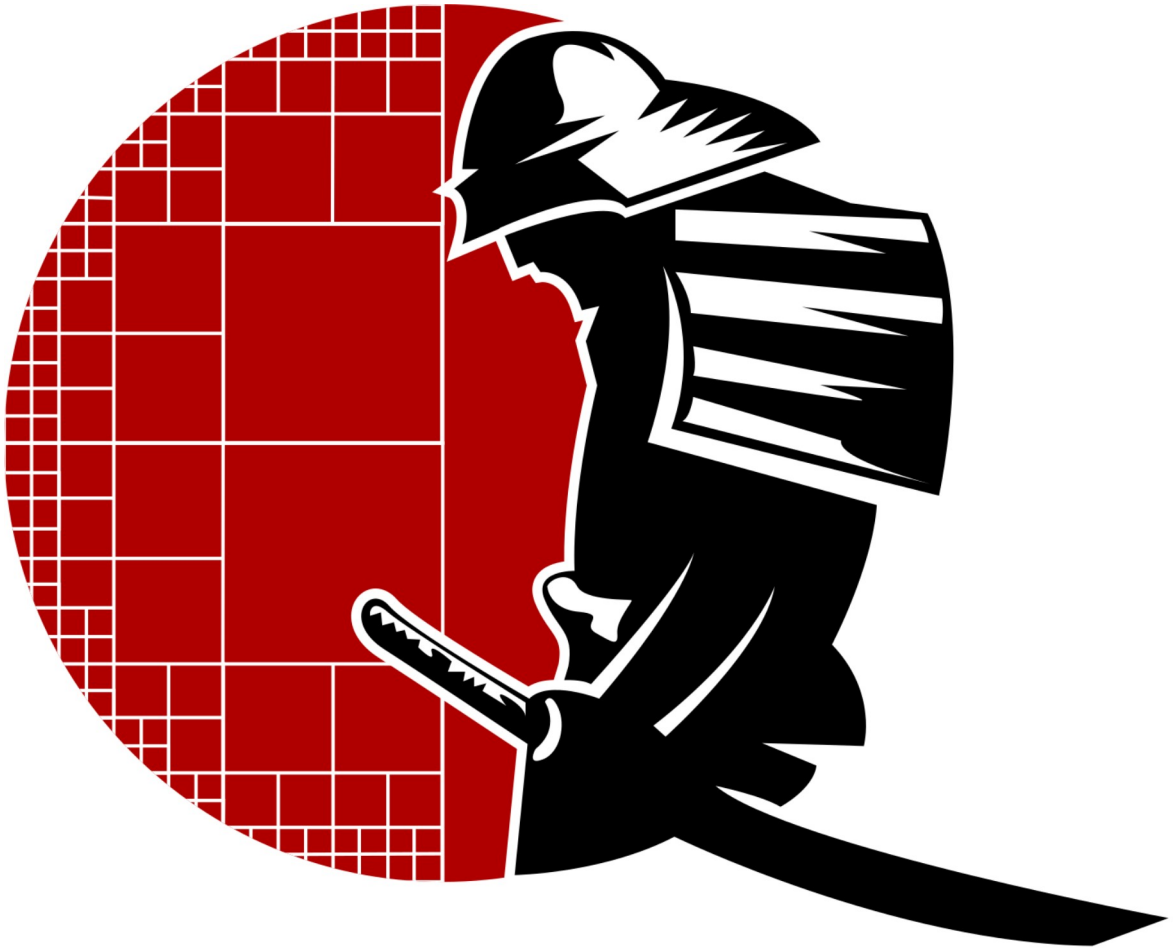
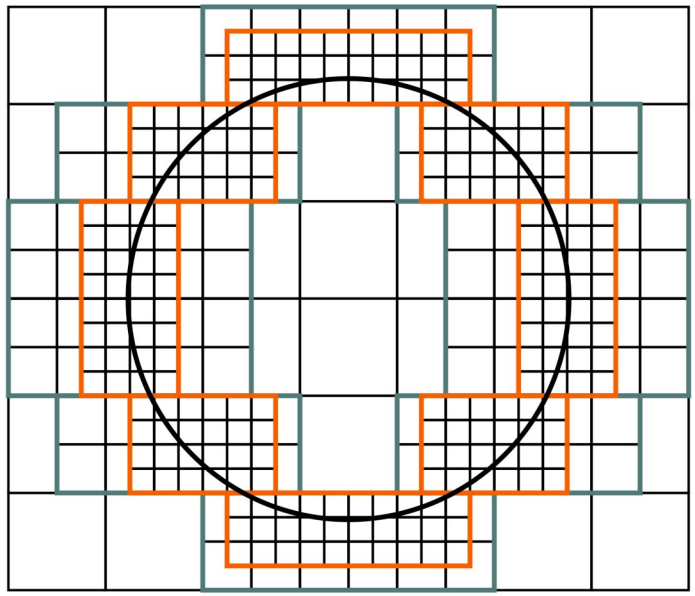


SAMURAI

Block-structured AMR @Exascale

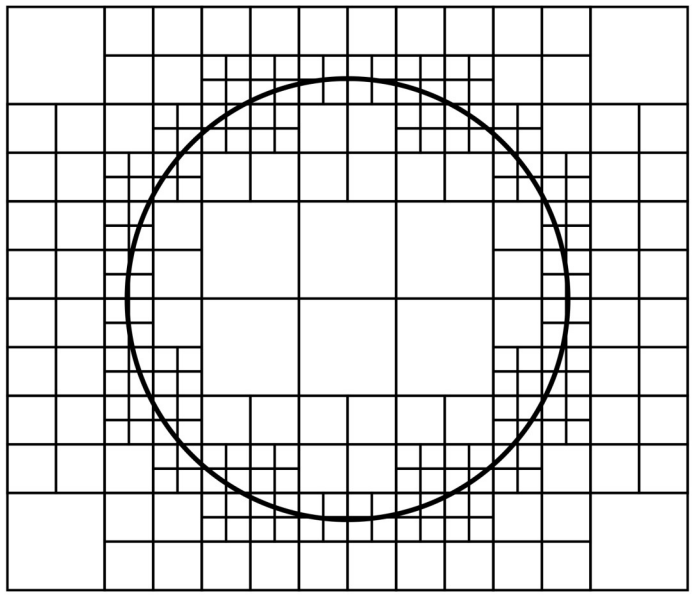
Loïc Gouarin

7 February 2024



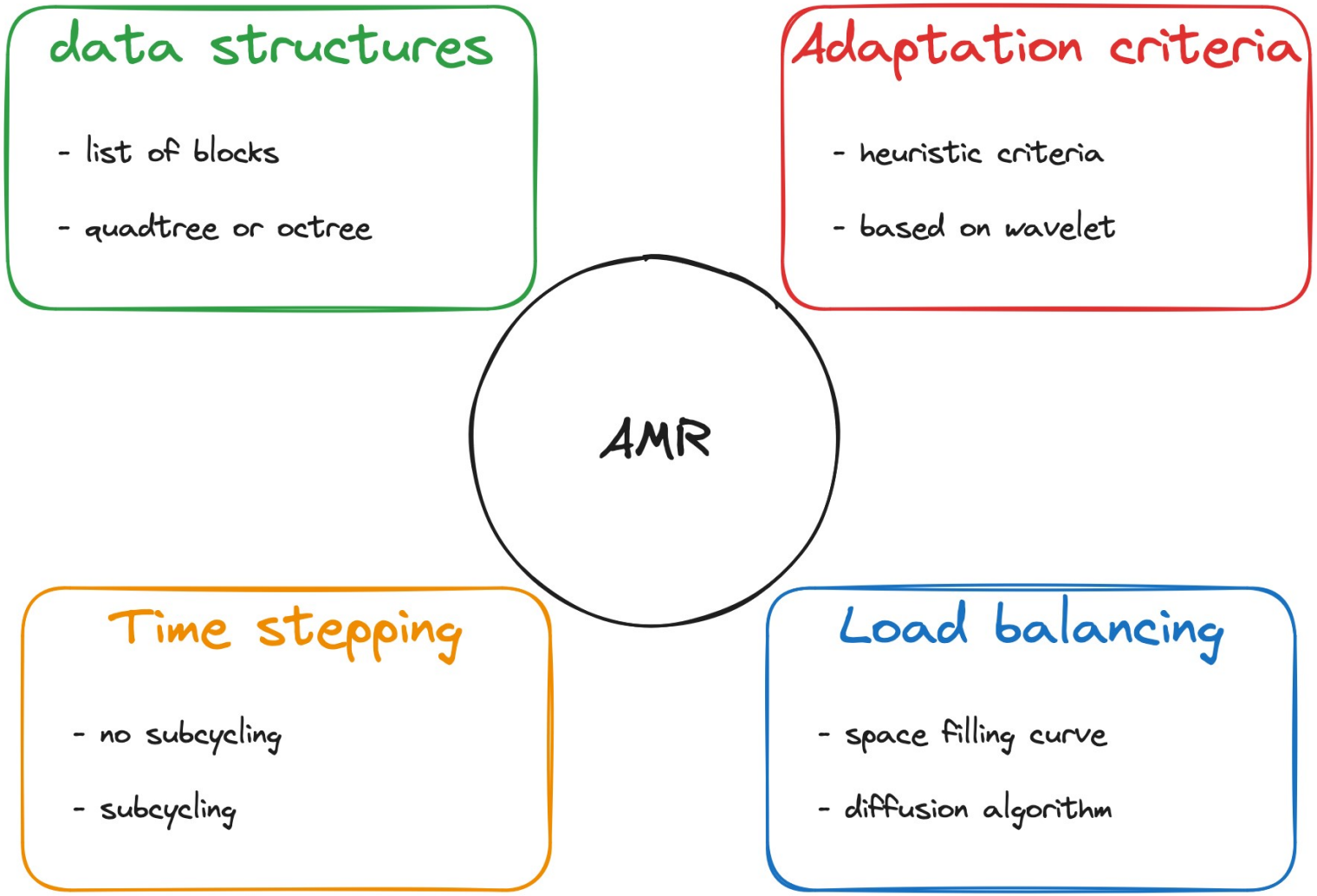
patch-based

AMR



cell-based

AMR and multiresolution

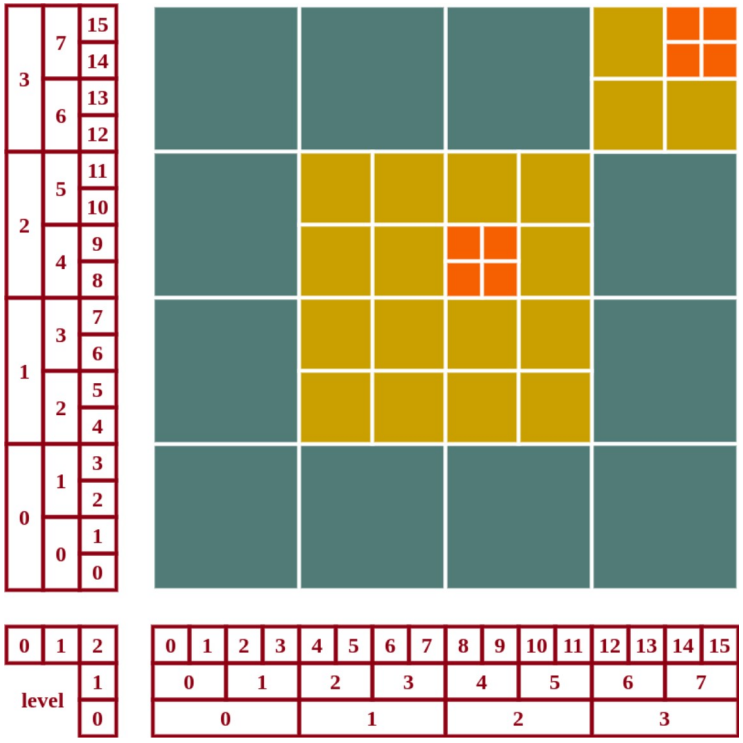

data structures
- list of blocks
- quadtree or octree
Adaptation criteria
- heuristic criteria
- based on wavelet
AMR
Time stepping
- no subcycling
- subcycling
Load balancing
- space filling curve
- diffusion algorithm

Name	Data structure	Adaptation criteria	Time scheme	Load balancing
AMReX	block	heuristic	global/local	SFC
Dendro	tree	wavelet	global	SFC
Dyablo	tree	heuristic	global	SFC
Peano	tree	-	-	SFC
P4est	tree	-	-	SFC
samurai	interval	heuristic/wavelet	RK/splitting/IMEX time-space/code coupling	SFC/ diffusion algorithm

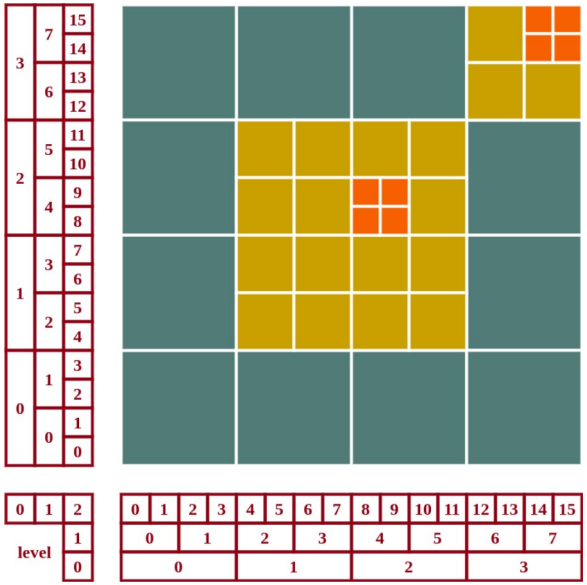
Name	Data structure	Adaptation criteria	Time scheme	Load balancing
AMReX	block	heuristic	global/local	SFC
Dendro	tree	wavelet	global	SFC
Dyablo	tree	heuristic	global	SFC
Peano	tree	-	-	SFC
P4est	tree	-	-	SFC
samurai	interval	heuristic/wavelet	RK/splitting/IMEX time-space/code coupling	SFC/ diffusion algorithm

samurai: create a unified framework for testing a whole range of mesh adaptation methods with the latest generation of numerical schemes.

- Mesh compression according to the level
- Achieve fast look-up for a cell into the structure
- Maximize the memory contiguity of the stored data
- Facilitate inter-level operations
- Give the possibility of writing numerical schemes easily

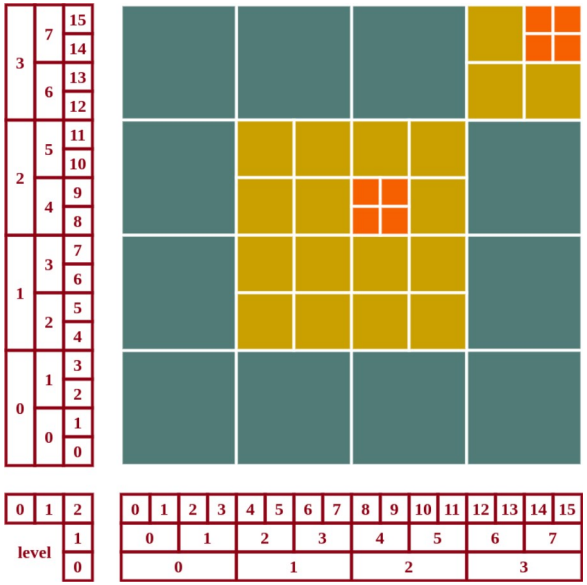


[start , end [@ offset : step



0	1	2
level		1
		0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1		2		3		4		5		6		7	
0				1				2				3			



Level 0

y: 0 → [0, 4[

y: 1 → [0, 1[, [3, 4[

y: 2 → [0, 1[, [3, 4[

y: 3 → [0, 3[

Level 1

y: 2 → [2, 6[

y: 3 → [2, 6[

y: 4 → [2, 4[, [5, 6[

y: 5 → [2, 6[

y: 6 → [6, 8[

y: 7 → [6, 7[

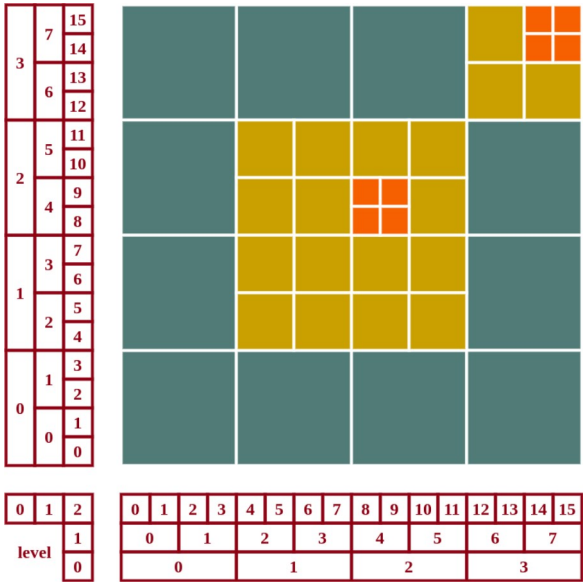
Level 2

y: 8 → [8, 10[

y: 9 → [8, 10[

y: 14 → [14, 16[

y: 15 → [14, 16[



Level 0

y: 0 → [0, 4[

y: 1 → [0, 1[, [3, 4[

y: 2 → [0, 1[, [3, 4[

y: 3 → [0, 3[

Level 1

y: 2 → [2, 6[

y: 3 → [2, 6[

y: 4 → [2, 4[, [5, 6[

y: 5 → [2, 6[

y: 6 → [6, 8[

y: 7 → [6, 7[

Level 2

y: 8 → [8, 10[

y: 9 → [8, 10[

y: 14 → [14, 16[

y: 15 → [14, 16[

Level 0

x: [0, 4[, [0, 1[, [3, 4[, [0, 1[, [3, 4[, [0, 3[

y: [0, 4[

y-offset: [0, 1, 3, 5, 6]

Level 1

x: [2, 6[, [2, 6[, [2, 4[, [5, 6[, [2, 6[, [6, 8[, [6, 7[

y: [2, 8[

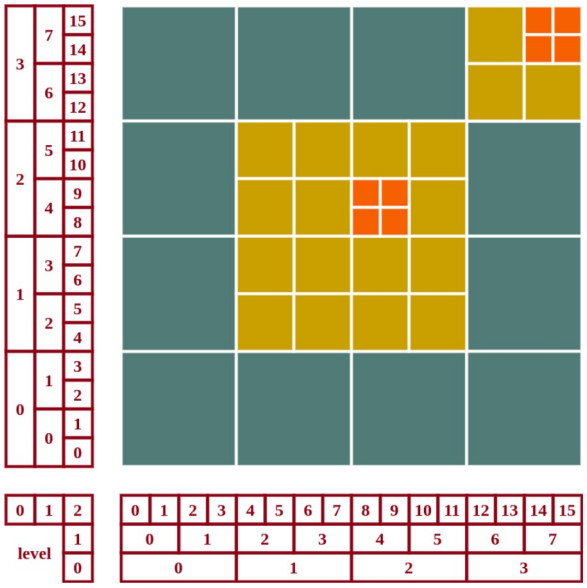
y-offset = [0, 1, 2, 4, 5, 6, 7]

Level 2

x: [8, 10[, [8, 10[, [14, 16[, [14, 16[

y: [8, 10[, [14, 16[

y-offset: [0, 1, 2, 3, 4]



Level 0

y: 0 → [0, 4[

y: 1 → [0, 1[, [3, 4[

y: 2 → [0, 1[, [3, 4[

y: 3 → [0, 3[

Level 1

y: 2 → [2, 6[

y: 3 → [2, 6[

y: 4 → [2, 4[, [5, 6[

y: 5 → [2, 6[

y: 6 → [6, 8[

y: 7 → [6, 7[

Level 2

y: 8 → [8, 10[

y: 9 → [8, 10[

y: 14 → [14, 16[

y: 15 → [14, 16[

Level 0

x: [0, 4[, [0, 1[, [3, 4[, [0, 1[, [3, 4[, [0, 3[

y: [0, 4[@0

y-offset: [0, 1, 3, 5, 6]

Level 1

x: [2, 6[, [2, 6[, [2, 4[, [5, 6[, [2, 6[, [6, 8[, [6, 7[

y: [2, 8[@-2

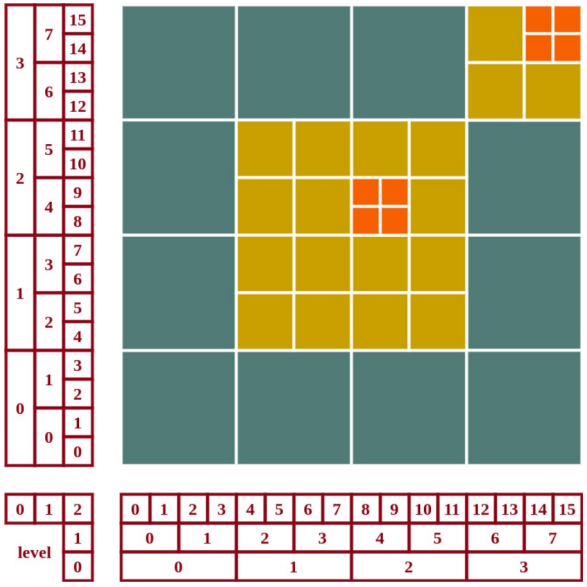
y-offset = [0, 1, 2, 4, 5, 6, 7]

Level 2

x: [8, 10[, [8, 10[, [14, 16[, [14, 16[

y: [8, 10[@-8, [14, 16[@-12

y-offset: [0, 1, 2, 3, 4]



Level 0

y: 0 → [0, 4[

y: 1 → [0, 1[, [3, 4[

y: 2 → [0, 1[, [3, 4[

y: 3 → [0, 3[

Level 1

y: 2 → [2, 6[

y: 3 → [2, 6[

y: 4 → [2, 4[, [5, 6[

y: 5 → [2, 6[

y: 6 → [6, 8[

y: 7 → [6, 7[

Level 2

y: 8 → [8, 10[

y: 9 → [8, 10[

y: 14 → [14, 16[

y: 15 → [14, 16[

cell list

Level 0

x: [0, 4[, [0, 1[, [3, 4[, [0, 1[, [3, 4[, [0, 3[

y: [0, 4[@0

y-offset: [0, 1, 3, 5, 6]

Level 1

x: [2, 6[, [2, 6[, [2, 4[, [5, 6[, [2, 6[, [6, 8[, [6, 7[

y: [2, 8[@-2

y-offset = [0, 1, 2, 4, 5, 6, 7]

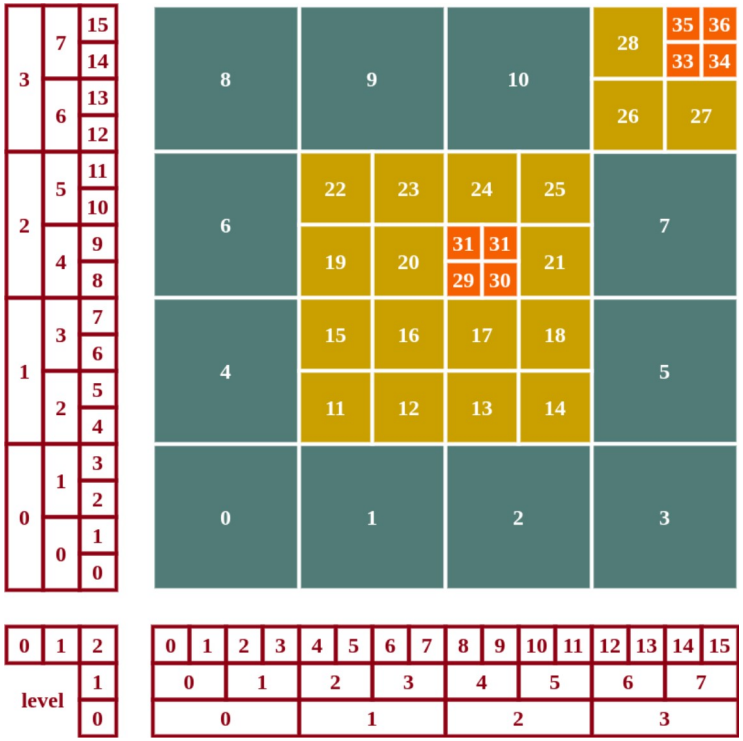
Level 2

x: [8, 10[, [8, 10[, [14, 16[, [14, 16[

y: [8, 10[@-8, [14, 16[@-12

y-offset: [0, 1, 2, 3, 4]

cell array



Level 0

x: [0, 4[@ 0, [0, 1[@ 4, [3, 4[@ 2, [0, 1[@ 6, [3, 4[@ 4, [0, 3[@ 4

y: [0, 4[@ 0

y-offset: [0, 1, 3, 5, 6]

Level 1

x: [2, 6[@ 9, [2, 6[@ 13, [2, 4[@ 17, [5, 6[@ 16, [2, 6[@ 20, [6, 8[@ 20, [6, 7[@ 22

y: [2, 8[@ -2

y-offset : [0, 1, 2, 4, 5, 6, 7]

Level 2

x: [8, 10[@ 21, [8, 10[@ 23, [14, 16[@ 19, [14, 16[@ 21

y: [8, 10[@ -8, [14, 16[@ -12

y-offset: [0, 1, 2, 3, 4]

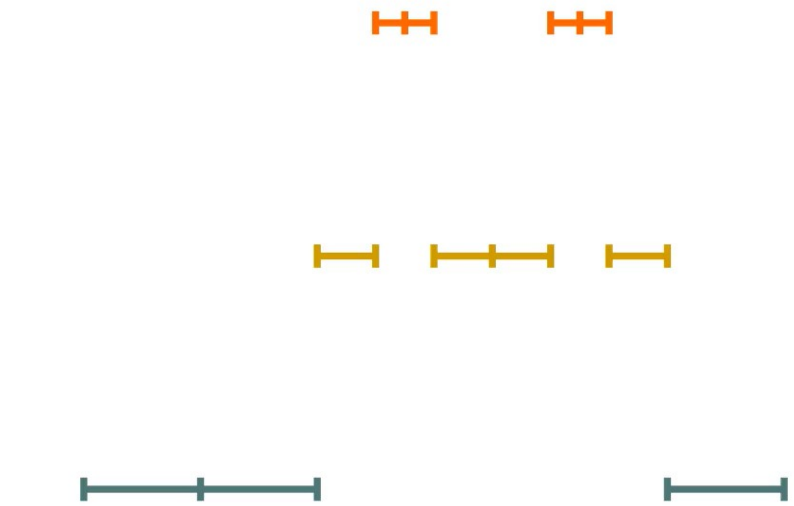
- A refined cell is split into 2 in 1d, 4 in 2d and 8 in 3d equal parts.
- At a given resolution level, the size of the cells is equal.
- The size of the cells is defined by the resolution level.

$$\Delta x = 2^{-level}$$

- A cell is represented by integer coordinates given its location.

$$center = \Delta x(indices + 0.5)$$

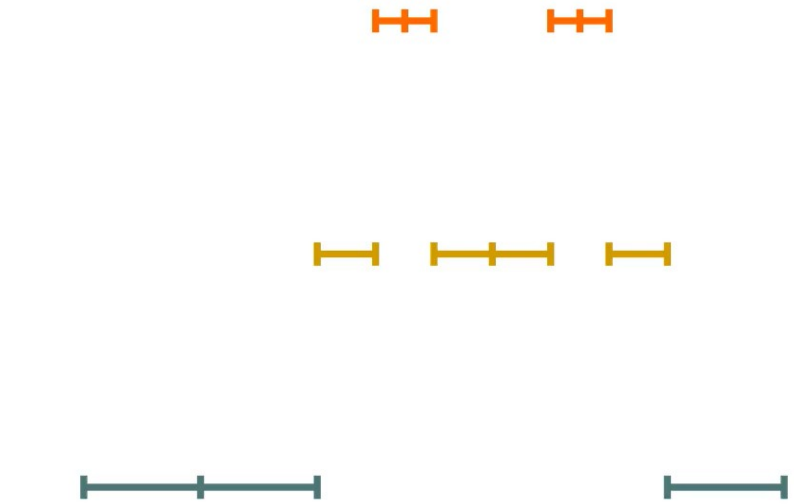
- The adapted mesh is generally graded.



Level 0 → [0, 2[[5, 6[

Level 1 → [4, 5[[6, 8[[9, 10[

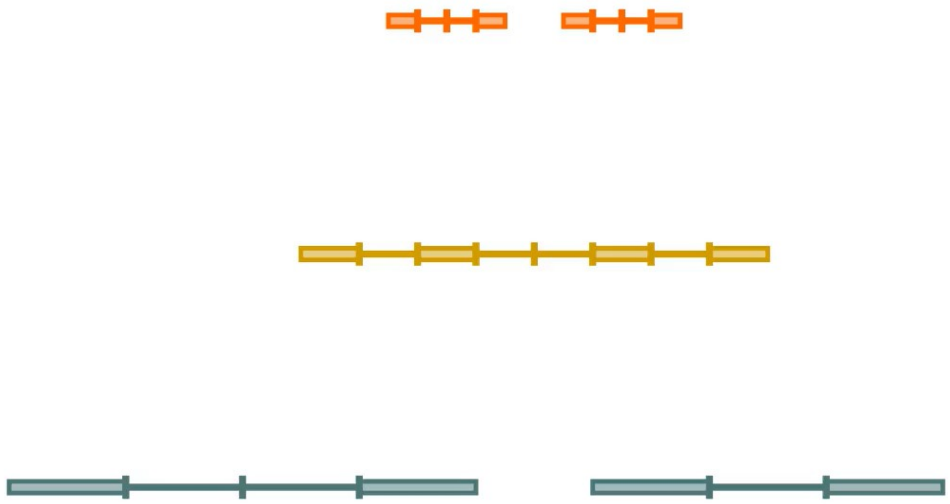
Level 2 → [10, 12[[16, 18[



Level 0 → [0, 2[[5, 6[

Level 1 → [4, 5[[6, 8[[9, 10[

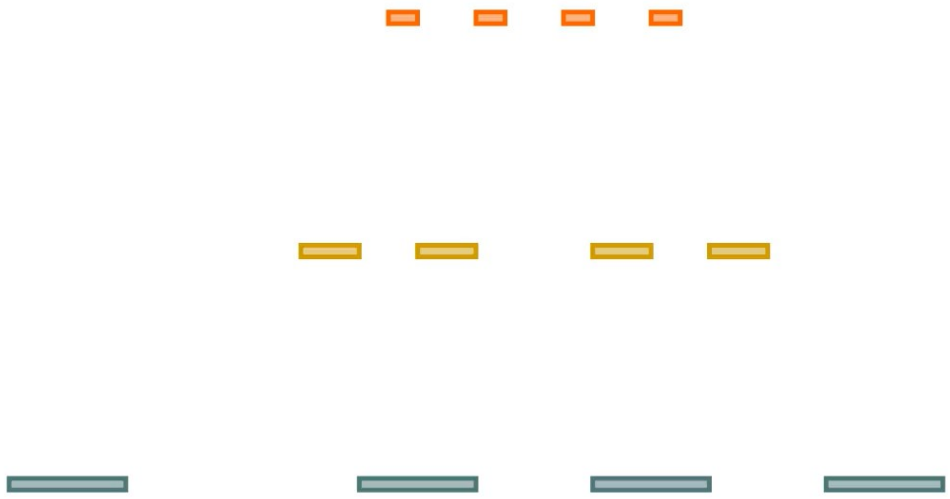
Level 2 → [10, 12[[16, 18[



Level 0 →

Level 1 →

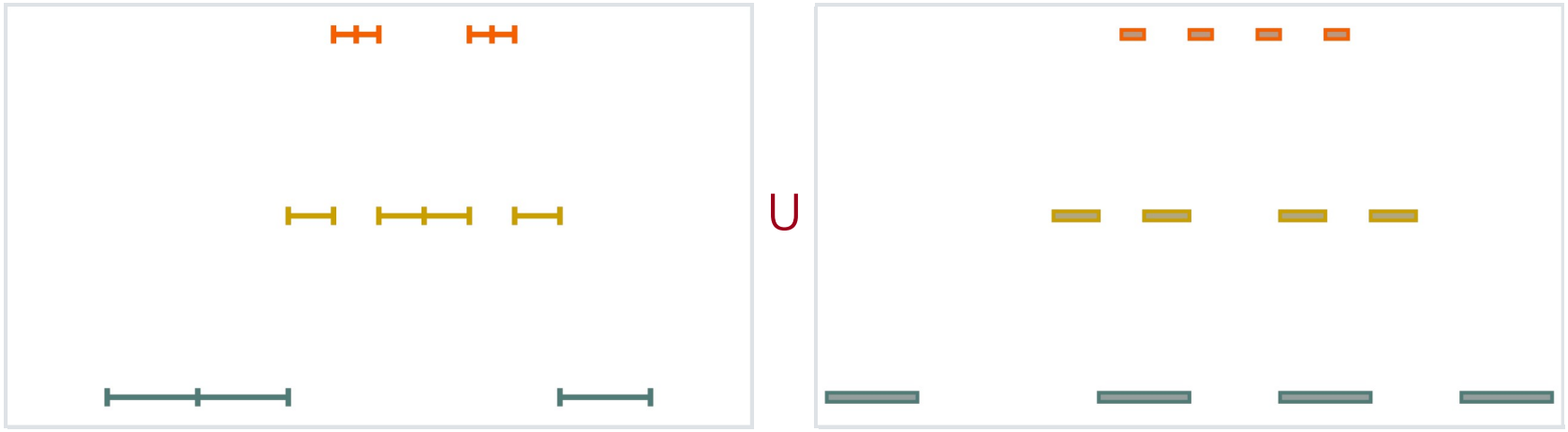
Level 2 →

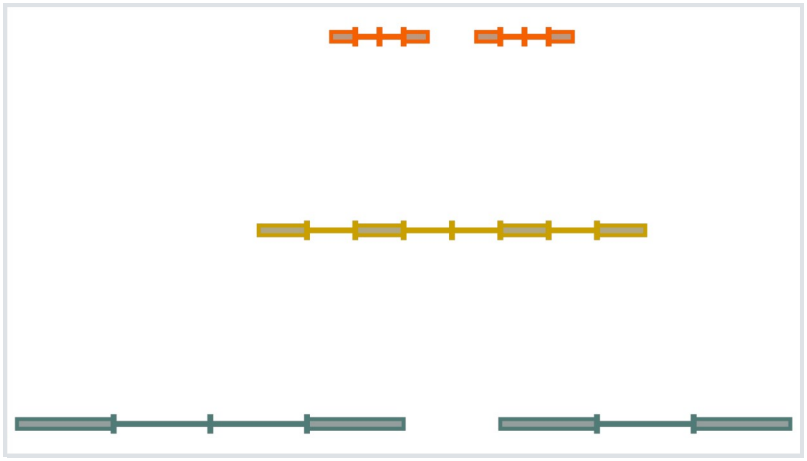


Level 0 → [–1, 0[[2, 3[[4, 5[[6, 7[

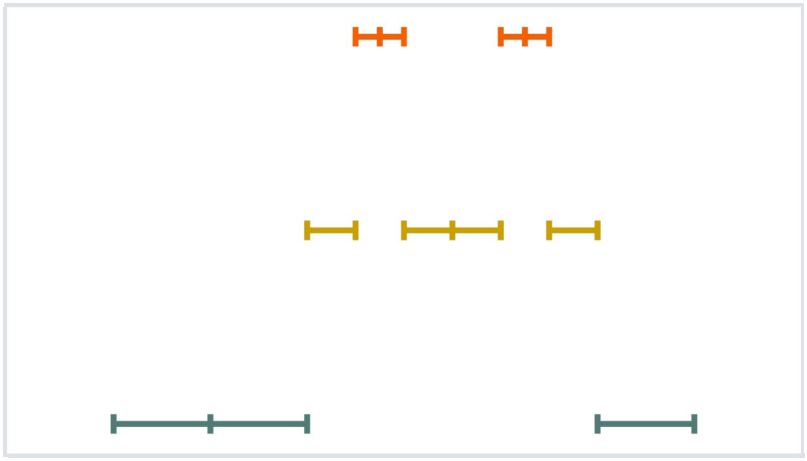
Level 1 → [3, 4[[5, 6[[8, 9[[10, 11[

Level 2 → [9, 10[[12, 13[[15, 16[[18, 19[

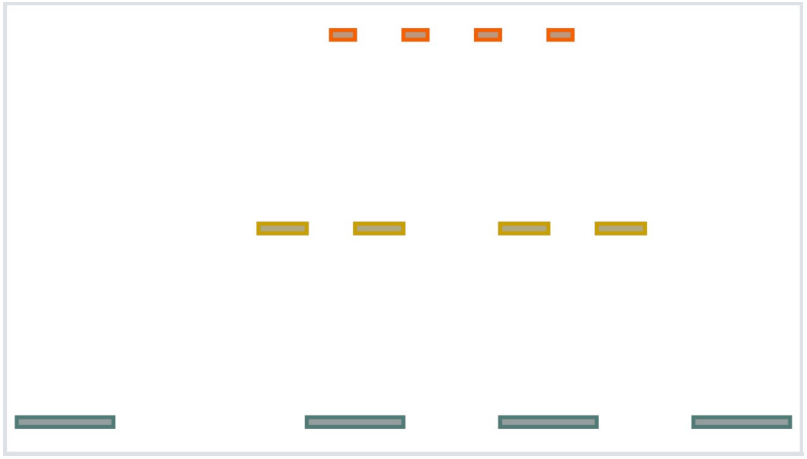

U
=



\



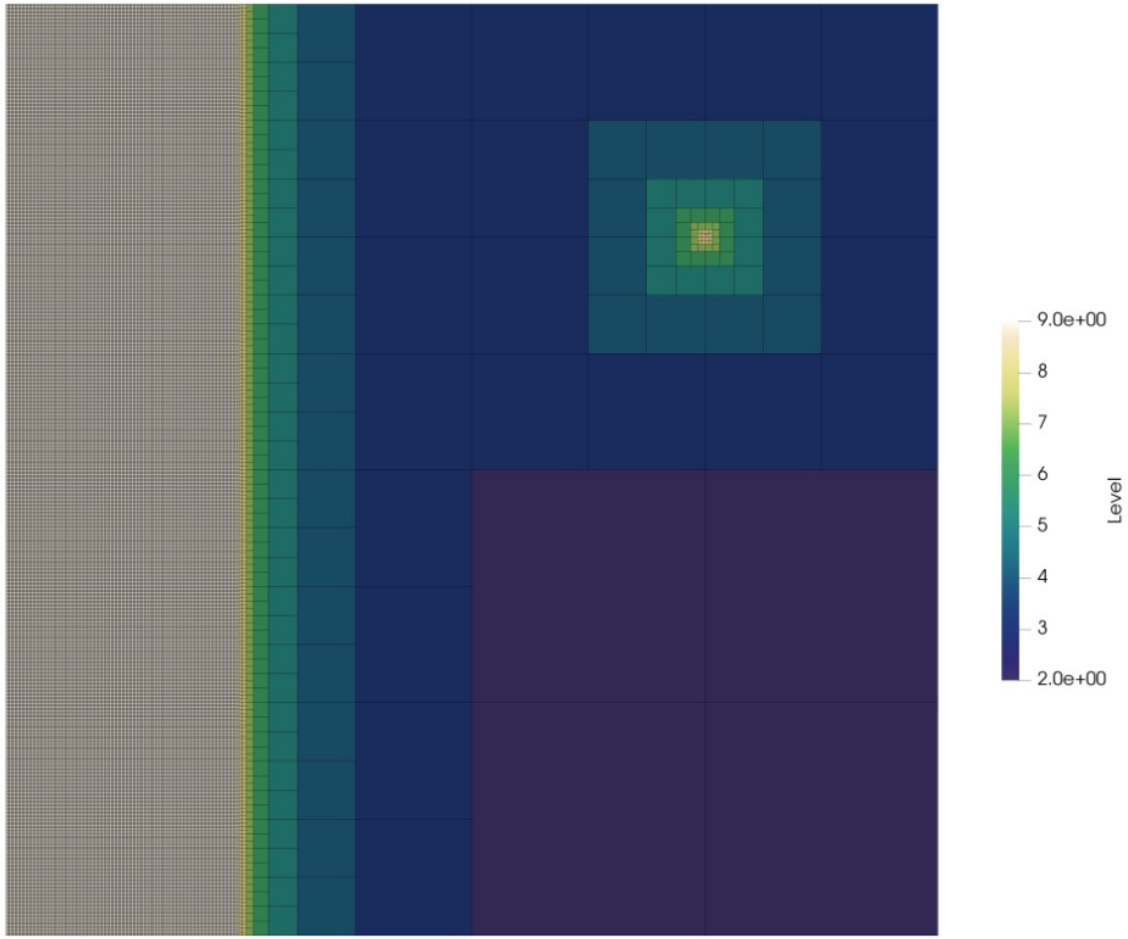
=



The search of an admissible set is recursive. The algorithm starts from the last dimension (y in 2d, z in 3d, ...).

The available operators in samurai are for now

- the intersection of sets,
- the union of sets,
- the difference between two sets,
- the translation of a set,
- the extension of a set.


9.0e+00
8
7
6
5
4
3
2.0e+00
Level

Level	Num. of cells	p4est	samurai (leaves)	samurai (all)	ratio
9	66379	2.57 Mb	33.68 Kb	121 Kb	21.24
10	263767	10.25 Mb	66.64 Kb	236.8 Kb	43.28
11	1051747	40.96 Mb	132.36 Kb	467.24 Kb	87.66
12	4200559	163.75 Mb	263.6 Kb	927 Kb	176.64
13	16789627	654.86 Mb	525.9 Kb	1.85 Mb	353.98
14	67133575	2.61 Gb	1.05 Mb	3.68 Mb	709.24

- Loop algorithms over the levels and the cells
- Simplified access operator
- Helper classes to construct complex meshes
- Helper classes to construct schemes for explicit and implicit usage
- Helper classes to construct N-D operators and expressions using xtensor
- HDF5 support

```cpp
auto mesh = samurai::MRMesh<Config>({{0, 0}, {1, 1}}, 2, 8);
auto u = samurai::make_field<double, 1>("u", mesh);

samurai::for_each_cell(mesh, [&](auto cell)
{
  auto x = cell.center(0);
  auto y = cell.center(1);

  u[cell] = x*x + y*y;
});
```

```cpp
auto u = samurai::make_field<double, 1>("u", mesh);
...
auto set = samurai::intersection(mesh[level], mesh[level + 1])
          .on(level);

set([&](const auto& i, auto)
{
    u(level, i) = 0.5*(u(level + 1, 2*i) + u(level + 1, 2*i + 1));
});
```

```
auto u = samurai::make_field<double, 1>("u", mesh);
...
auto set = samurai::intersection(mesh[level], mesh[level + 1])
           .on(level);

set([&](const auto& i, auto)
{
    u(level, i) = 0.5*(u(level + 1, 2*i) + u(level + 1, 2*i + 1));
});
```

```cpp
auto u = samurai::make_field<double, 1>("u", mesh);
...
auto set = samurai::intersection(mesh[level], mesh[level + 1])
          .on(level);

set([&](const auto& i, auto)
{
    u(level, i) = 0.5*(u(level + 1, 2*i) + u(level + 1, 2*i + 1));
});
```

GRADUATION

before

after

```cpp
constexpr std::size_t dim = 2;
std::size_t start_level = 1;
std::size_t max_refinement_level = 7;

samurai::Box<int, dim> box({0, 0}, {1<<start_level, 1<<start_level});

samurai::CellArray<dim> ca;
ca[start_level] = {start_level, box};
```

```
samurai::CellList<dim> cl;

samurai::for_each_interval(ca, [&](std::size_t level, const auto& interval, const auto& index)
{
    auto choice = xt::random::choice(xt::xtensor_fixed<bool, xt::xshape<2>>{true, false}, interval.size());

    for(int i = interval.start, ic = 0; i<interval.end; ++i, ++ic)
    {
        if (choice[ic])
        {
            cl[level + 1][2*index].add_interval({2*i, 2*i+2});
            cl[level + 1][2*index + 1].add_interval({2*i, 2*i+2});
        }
        else
        {
            cl[level][index].add_point(i);
        }
    }
});

ca = {cl, true};
```

```
std::size_t min_level = ca.min_level(), max_level = ca.max_level();
xt::xtensor_fixed<int, xt::xshape<4, dim>> stencil{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

auto tag = samurai::make_field<bool, 1>("tag", ca);
tag.fill(false);

for(std::size_t level = min_level + 2; level <= max_level; ++level)
{
    for(std::size_t level_below = min_level; level_below < level - 1; ++level_below)
    {
        for(std::size_t i = 0; i < stencil.shape()[0]; ++i)
        {
            auto s = xt::view(stencil, i);
            auto set = samurai::intersection(samurai::translate(ca[level], s), ca[level_below])
                       .on(level_below);
            set([&](const auto& i, const auto& index)
            {
                tag(level_below, i, index[0]) = true;
            });
        }
    }
}
```

```
samurai::CellList<dim> cl;

samurai::for_each_cell(ca, [&](auto cell)
{
    auto i = cell.indices[0];
    auto j = cell.indices[1];

    if (tag[cell])
    {
        cl[cell.level + 1][{2*j}].add_interval({2*i, 2*i+2});
        cl[cell.level + 1][{2*j + 1}].add_interval({2*i, 2*i+2});
    }
    else
    {
        cl[cell.level][{j}].add_point(i);
    }
});

samurai::CellArray<dim> new_ca = {cl, true};
```

We consider the well known heat equation

$$\frac{\partial u}{\partial t} - \Delta u = 0$$

Since we have

$$\int_V \Delta u = \int_{\partial V} \nabla u \cdot \mathbf{n},$$

the flux function to implement is a discrete version of $\nabla u \cdot \mathbf{n}$. Here, we choose the normal gradient of the first order, requiring a stencil of two cells. This is enough to write the static configuration

```
auto u = samurai::make_field<1>("u", mesh); // scalar field

using cfg = samurai::FluxConfig<SchemeType::LinearHomogeneous,
                                1,            // output_field_size
                                2,            // stencil_size
                                decltype(u)>; // input_field_type
```

Now, denoting by V_L (left) and V_R (right) the stencil cells and F their interface, the discrete flux from V_L to V_R writes

$$\mathcal{F}_h(u_h)_{|F} := \frac{u_R - u_L}{h},$$

where u_L and u_R are the finite volume approximations of u in the respective cells, and h is the cell length. The flux function then writes

```
samurai::FluxDefinition<cfg> gradient([](double h)
{
    samurai::FluxStencilCoeffs<cfg> c;
    c[0] = -1/h; // left
    c[1] =  1/h; // right
    return c;
});

auto diff = samurai::make_flux_based_scheme(gradient);
```

```
...
auto diff = samurai::make_diffusion<decltype(u)>();
auto id   = samurai::make_identity<decltype(u)>();
for(std::size_t ite=0; ite<200; ++ite)
{
    samurai::update_ghost_mr(u);

    if (explicit_scheme)
    {
        unp1 = u - dt * diff(u);
    }
    else
    {
        auto back_euler = id + dt * diff;
        // solves the linear equation   [Id + dt*Diff](unp1) = u
        samurai::petsc::solve(back_euler, unp1, u);
    }

    std::swap(u.array(), u_np1.array());

    samurai::save(fmt::format("mesh_{}", ite + 1), mesh, u);
}
```

```cpp
auto MRadaptation = samurai::make_MRAdapt(u);
double eps = 1e-4, regularity = 2;

for(std::size_t ite=0; ite<200; ++ite)
{
    MRadaptation(eps, regularity);
    unp1.resize();
    samurai::update_ghost_mr(u);

    if (explicit_scheme)
    {
        unp1 = u - dt * diff(u);
    }
    else
    {
      auto back_euler = id + dt * diff;
      // solves the linear equation   [Id + dt*Diff](unp1) = u
      samurai::petsc::solve(back_euler, unp1, u);
    }

    std::swap(u.array(), u_np1.array());
    samurai::save(fmt::format("mesh_{}", ite + 1), mesh, u);
}
```

Implement your finite volume scheme on a uniform cartesian grid

and

Implement your finite volume scheme on a uniform cartesian grid

and

you have explicit and implicit at your disposal

Implement your finite volume scheme on a uniform cartesian grid

and

you have explicit and implicit at your disposal

you have a large renage of time schemes available

Implement your finite volume scheme on a uniform cartesian grid

and

you have explicit and implicit at your disposal

you have a large renage of time schemes available

you have the adaptation (MRA / AMR) without further effort

Implement your finite volume scheme on a uniform cartesian grid

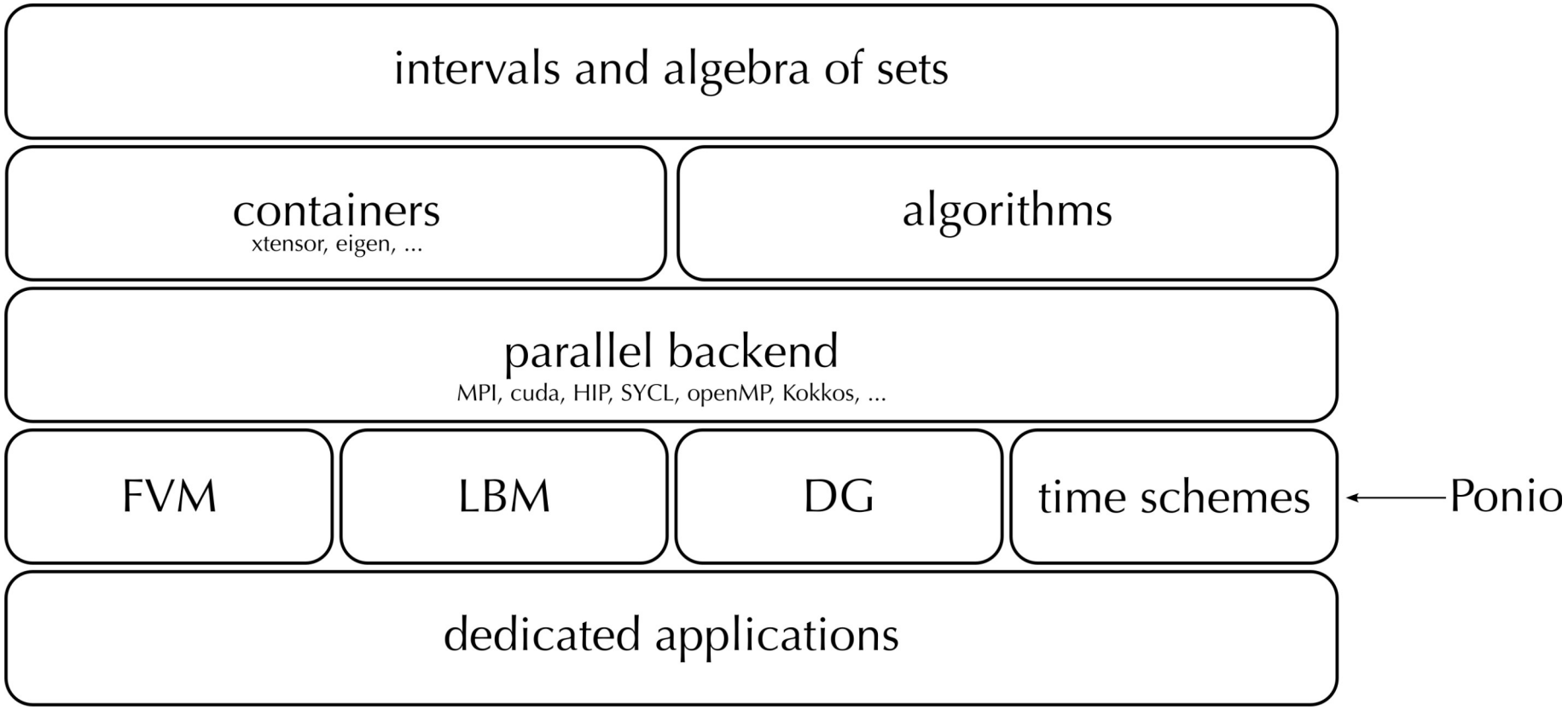
and

you have explicit and implicit at your disposal

you have a large renage of time schemes available

you have the adaptation (MRA / AMR) without further effort

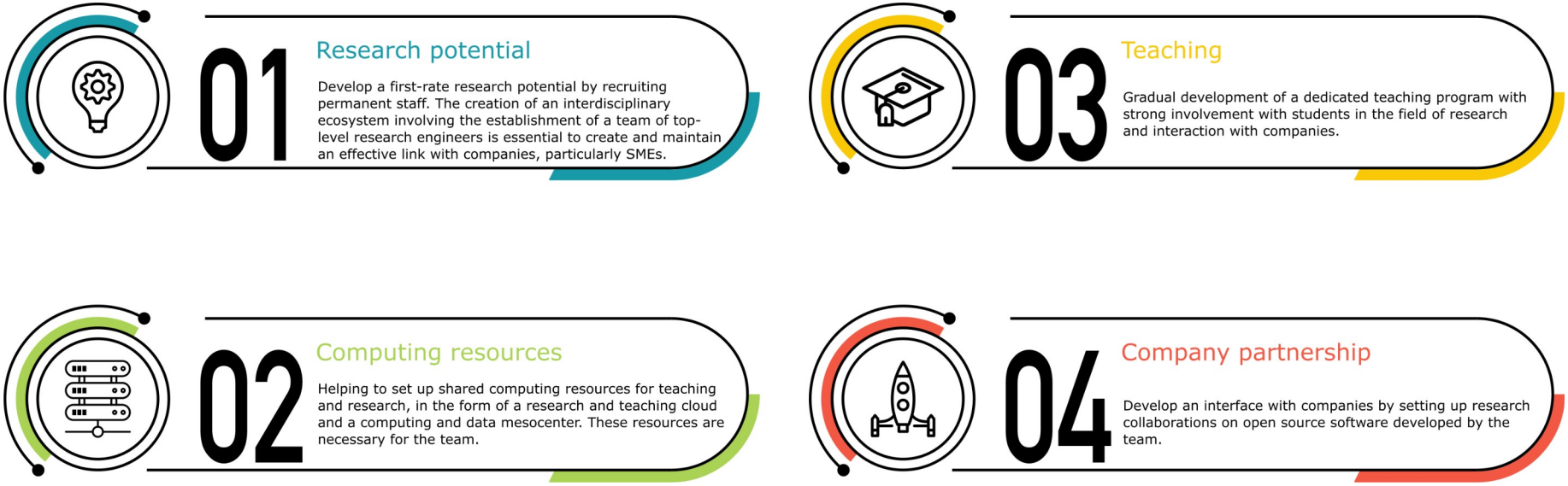
you have parallelism


intervals and algebra of sets
containers
xtensor, eigen, ...
algorithms
parallel backend
MPI, cuda, HIP, SYCL, openMP, Kokkos, ...
FVM
LBM
DG
time schemes
Ponio
dedicated applications

- Optimization
- Flexibility in term of container choice
- Parallel backends
- Load balancing using diffusion algorithm or space filling curves
- GPU support
- IO optimization and compression

samurai is part of the projects developed in the HPC@Maths team (5 permanent researchers, 5 research engineers, 1 project manager, 11 research associates, 6 PhD students, 1 Post-doc).

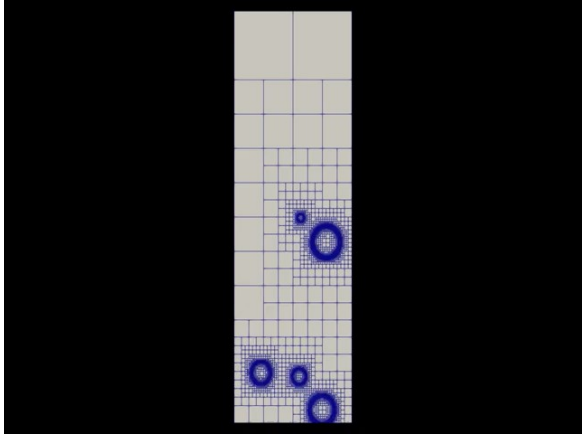
The main goals of this team are

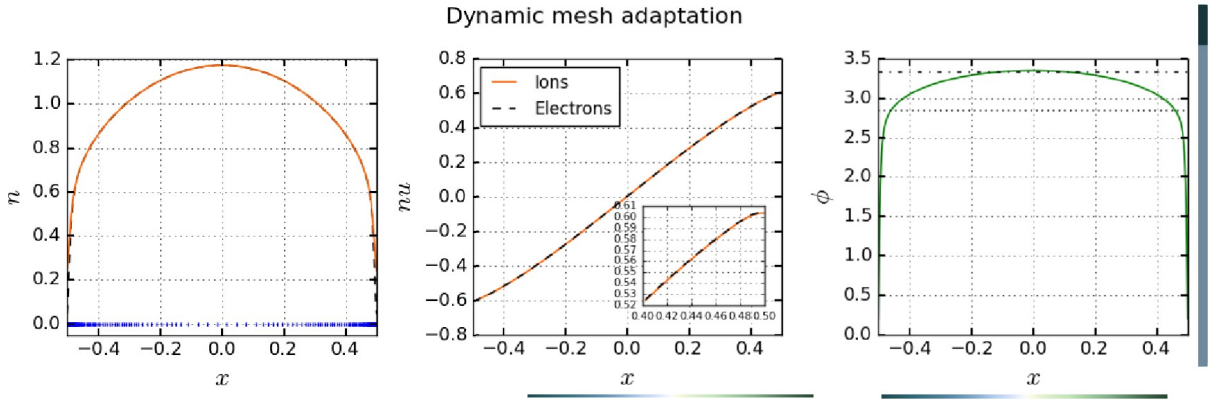


PI: L.G and M. Massot - HPC@Maths website

- Lattice Boltzmann methods and multiresolution - **Thomas Bellotti** (IRMA/Université Strasbourg) and **Benjamin Graille** (LMO/Université Paris-Saclay)
- Plasma discharges and electric propulsion - **Alejandro Alvarez** (LPP/Ecole polytechnique) and **Louis Reboul** (ONERA)
- Direct numerical simulation of lithium-ion batteries based on high-resolution 3D images of porous electrode microstructures - **Ali Asad** (TotalEnergies) and **Laurent François** (ONERA)
- Sharp interface method for low Mach two-phase flows - **Nicolas Grenier** (LISN/Université Paris-Saclay) and **Christian Tenaud** (EM2C/Université Paris-Saclay)
- Low-Mach reactive flows - **Christian Tenaud** (EM2C/Université Paris-Saclay)
- Interfacial flow simulation - **Giuseppe Orlando** (CMAP/Ecole polytechnique) and **Marica Pelanti** (ENSTA/IP Paris)
- Mathematical modeling and simulation of non-equilibrium plasmas for the prediction of electric propulsion - **Zoubaïr Tazakkati** (CMAP/Ecole polytechnique)
- Simulation analysis on the Hydrogen risk - **Luc Lecointre, Pierre-Alexandre Masset** (CEA) and **Christian Tenaud** (EM2C/Université Paris-Saclay)

- 2 PhD with CEA and INRIA (ExaMA WP1)
- Engineer (ExaMA WP7)
- Post-doc with CEA (ExaDost)




Dynamic mesh adaptation
n
x
nu
Ions
Electrons
x
ϕ
x

SAMURAI

https://github.com/hpc-maths/samurai