# GPU & Parallelization

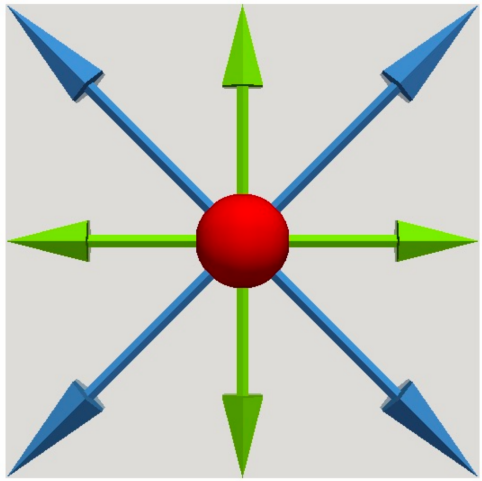
AC290r - Simone Melchionna

National Research Council Italy & Lexma Technology LLC

April 10, 2019

# Recap

$$
\begin{aligned}
f_p(x + c_p, t + 1) &= f_p^*(x, t) \\
&= f_p(x, t) + \\
&\quad \omega \left[ f_p^{eq}(\rho, \mathbf{u}) - f_p \right](x, t) + \\
&\quad w_p \frac{\mathbf{c}_p \cdot \mathbf{g}}{c_s^2}
\end{aligned}
$$



Mesh is Cartesian

Struggling with debugging?

$$
\begin{aligned}
f_p^{eq}(\rho, \mathbf{u}) &= w_p \rho \left[ 1 + \frac{\mathbf{u} \cdot \mathbf{c}_p}{c_S^2} + \frac{(\mathbf{c}_p \cdot \mathbf{u})^2 - c_S^2 u^2}{2 c_S^4} \right] \\
\rho(\mathbf{x}, t) &= \sum_p f_p(\mathbf{x}, t) \\
\mathbf{u}(\mathbf{x}, t) &= \frac{1}{\rho(\mathbf{x}, t)} \sum_p \mathbf{c}_p f_p(\mathbf{x}, t)
\end{aligned}
$$

$$
\begin{aligned}
\sum_p f_p(\mathbf{x}, t) &= \sum_p f_p^*(\mathbf{x}, t) \\
N = \sum_x \sum_p f_p(\mathbf{x}, t) &= const
\end{aligned}
$$

$(h = 1)$

# How to organize a LBM code

Simplicity: no finite differences or integrals (fluxes)

1. Read input parameters
2. Set populations to equilibrium
3. Set timestep = 0
4. While (timestep < TotalSteps):
5. Boundary Conditions at inlet/outlet
6. Collide populations
7. Stream populations
8. Bounce-back BC
9. Every n steps, perform analysis
10. timestep += 1 and Goto 4.

Initial conditions

$$\rho(x, t = 0) = \rho_0(x)$$

$$u(x, t = 0) = u_0(x)$$

$$f_p(x, 0) = f_p^{eq}(\rho_0(x), u_0(x))$$

Boundary Conditions (Bounce Back)

$$f_p(x + c_p, t + 1) = f_p^*(x, t)$$

# Thinking of a large-scale project

Organizing a large-scale code requires:

- low-level compiled language for speed
- high-level scripting language for management

Object Orienting (OO) and data layout are key in sw engineering:

1. data encapsulation: data private to classes and bind to methods
2. inheritance: class hierarchy should be as linear as possible
3. polymorphism: calling code is agnostic about operating on objects

# Shared libraries & scripting

Strategy:

- A shared library
- Mirroring scripting with C/Fortran classes

Python is great for high-level usage and organization.
Highly expressive, elegant and benefits from a huge community
Integrate python + C/Fortran via ctypes or cython (efficient due to typing)

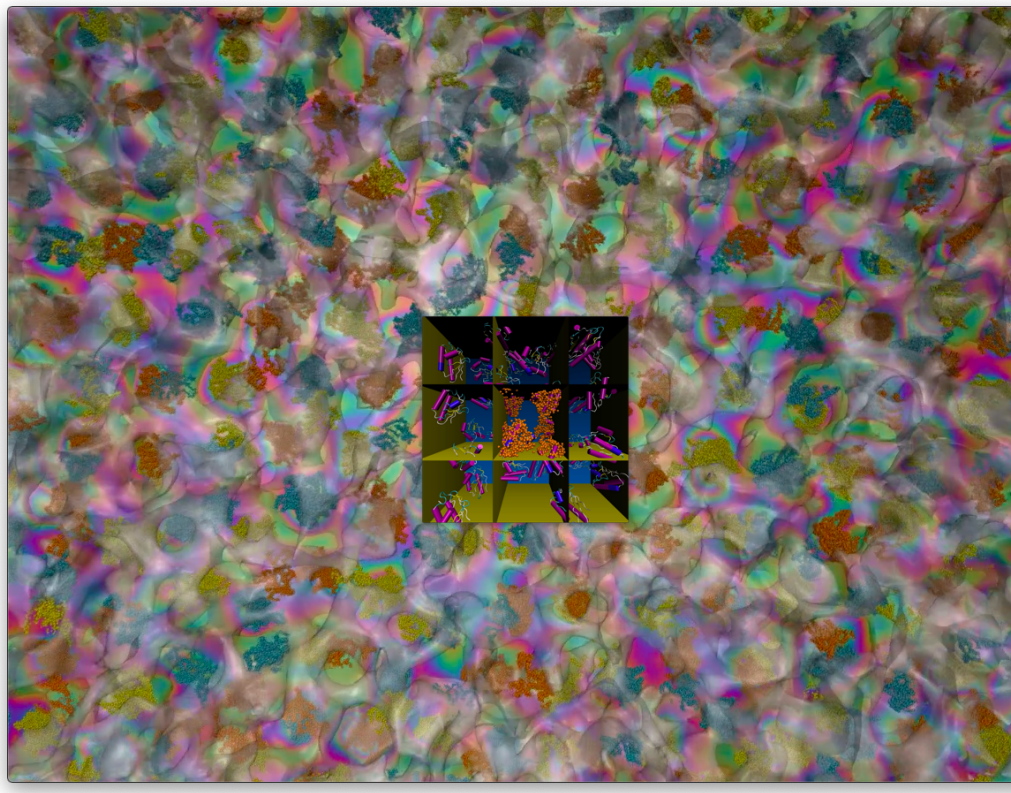
```
gcc -shared -fPIC foo.c -o libfoo.so
....
import ctypes
libtest = ctypes.cdll.LoadLibrary('./libfoo.so')
print libtest.doit()
```

Almost anything (data analytics, image analysis, bioinformatics, visualization) has a python API to access high to mid-level data without spoiling performances.
Maintenability, efficiency and stability are all equally important.

# MUPHY (& MOEBIUS)

Today simulation is facing new challenges:

- complex and deformable physiological geometries (medical imaging)
- Cover disparate space/time scales with multi-resolution

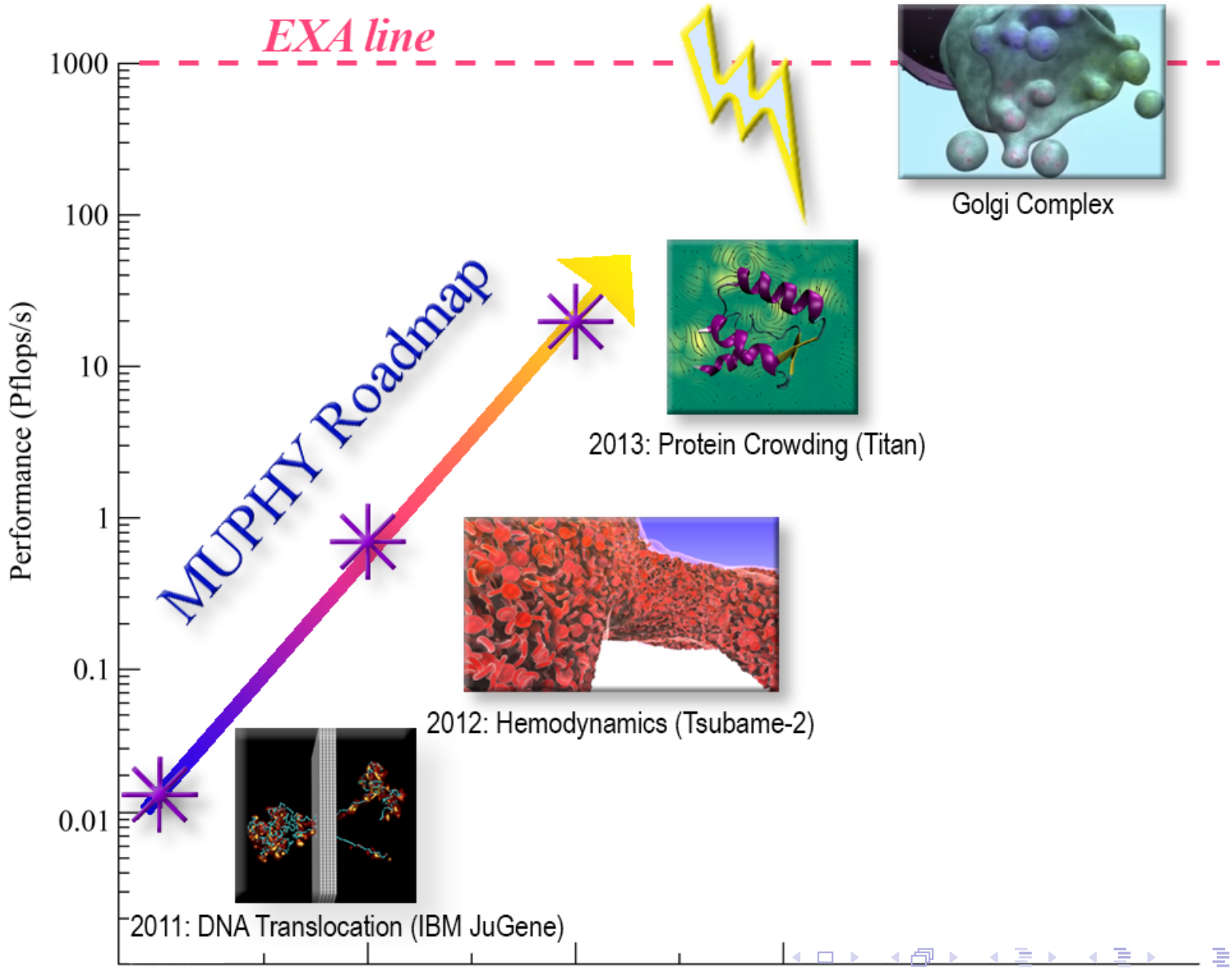


These can be tackled by an organic platform:

- MUPHY (10 years old): Multi-Physics
- MOEBIUS (recent, by Lexma Tech): Multi-Scale

Ultimate common goal: bridging from engineering to life sciences

# Exascale coming of age



# MUPHY Usage

A few initial operations:
Download everything from folder:
https://drive.google.com/open?id=192JNqK8gixy4oA6hVqQ3yExwp9s3xhRh

module load cuda/10.0.130-fasrc01

module load metis/5.1.0-fasrc01
export MODULEPATH=$MODULEPATH:/n/sw/eb/modulefiles/centos7/Core
module load GCC/7.3.0-2.30
module load OpenMPI/3.1.1 module load VTK/8.1.1-Python-3.6.6

We will work with a precompiled library and an ecosystem of scripts. Thus:

export MOEBIUS_ROOT= .../MAGIC
export
PYTHONPATH=$MOEBIUS_ROOT/BACKEND/SHOP:$MOEBIUS_ROOT/BACKEND/SCRIPTS

./run2.py is the script that, after loading the module MagicUniverse, runs the simulation

MagicUniverse.py has several functions to set/get and streer the simulation.
help(MagicUniverse) is scarcely informative.

```
from MagicUniverse import *
MagicBegins()

# define universe & actors
u = Universe()
s = Scale()
m = Mesh()
f = Fluid()
t = Tracker()

u.addItems([s,m,f,t])
u.setTitle('Periodic channel')
u.setNumberOfSteps(100)
u.setStateRestart(False)
u.setStateDumpFrequency(-1)
u.create()

s.set(name='MonoScale',
mesh=m, actors=[f,t])

# set various parameter
m.setRegularMesh(True)
m.setPeriodicity('100')

t.setDiagnosticFrequency(10)
t.setDataShow(density=True,velocity=True)
t.setMapDirections('zx')
t.setVtkDump(True, start=0,
frequency=10)

f.setName('Fluid')
f.setViscosity(1./6.)
f.setInletOutletMethod('closed')
f.setHomogeneousForce(1.e-4,0.,0.)

u.decorate()

# time loop
for itime in u.cycle():
    u.animate()

MagicEnds()
```

# Files

- Input

run2.py : runner
bgkflag.hdr : mesh header file
bgkflag.dat : mesh nodes
OR
bgkflag_0....N-1.dat and nodeownr.inp for user-defined partitioning on N tasks
bgfklag.ios : inlet/outlet boundary conditions

- Output

standard output to track simulation
DIRDATA_X : the folder with 1D and 2D data for density and velocity
DIRDATA_X/VTK/* : folder with files for visualization by frames (paraview)

More sophisticated usage of MUPHY on request.

# Indirect addressing

- Each discrete velocity is labelled $p$
- the connectivity matrix $Conn[p, i, j, k]$ points to the neighbor node $i', j', k'$
- the connectivity matrix also allows accessing the neighbor nodes in a compact way

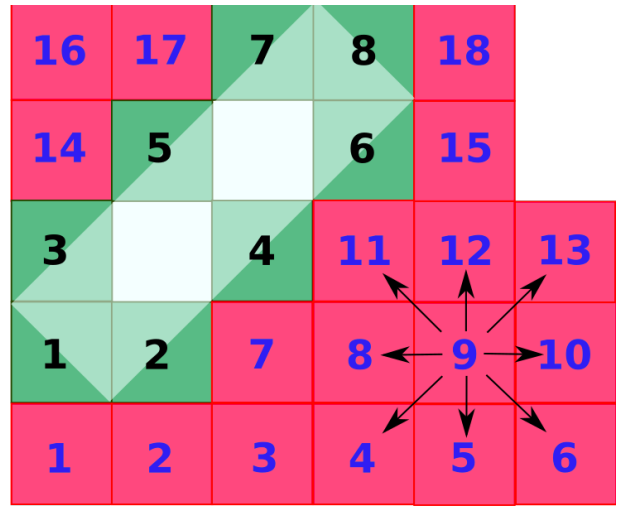



Figure: Red for fluid nodes, green for wall nodes for a rectangular obstacle

- +: Only fluid, wall, inlet, outlet nodes need to be stored
- +: Homogeneous nodes can be stored contiguously
- -: Target locations of streaming not directly/easily computed
- -: Storage for connectivity matrix is required.

# Optimizing LBM

Let's discuss the code snippet
    streamCollideCompute.c
and understand how to optimize LBM.

# Cache optimization

Data locality delivers the best performances. Suppose to store pops with linear indices $NX * (NY * ip + j) + i$
Read and writing to memory should improve locality but this is limited to a given direction.

For D2Q9, pops are separated by 8*NX*NY bytes. When it's a multiple of the cache block size (e.g. number of nodes = high power of two) all writes use the same line within a block: inefficient. Non-power-of-two domain sizes are better.
Padding, i.e. adding memory to prevent alignment of memory addresses, alleviates the problem.
E.g. allocate 131 x128 domain but only use 128 x 128 nodes.

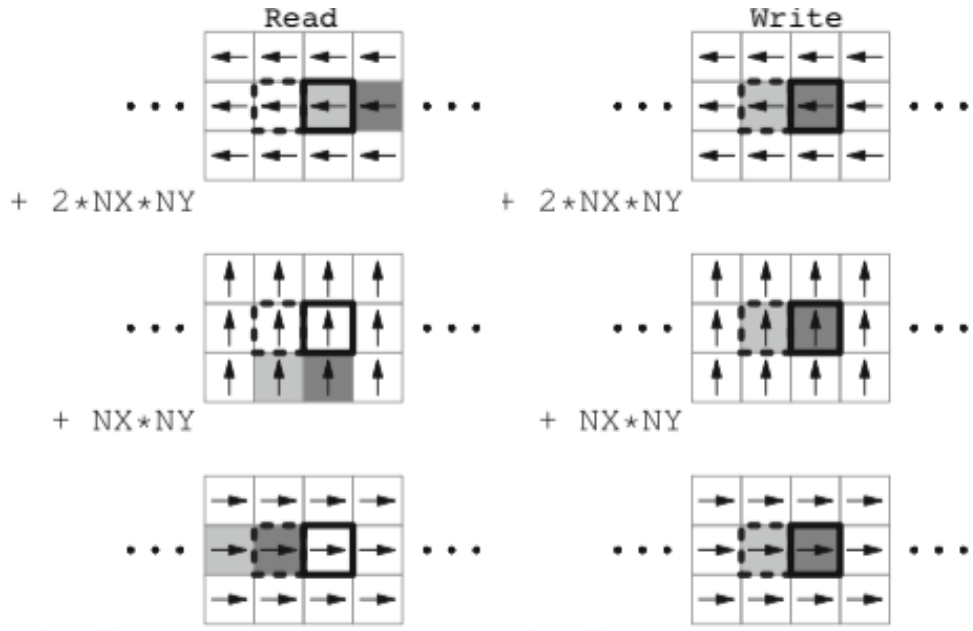


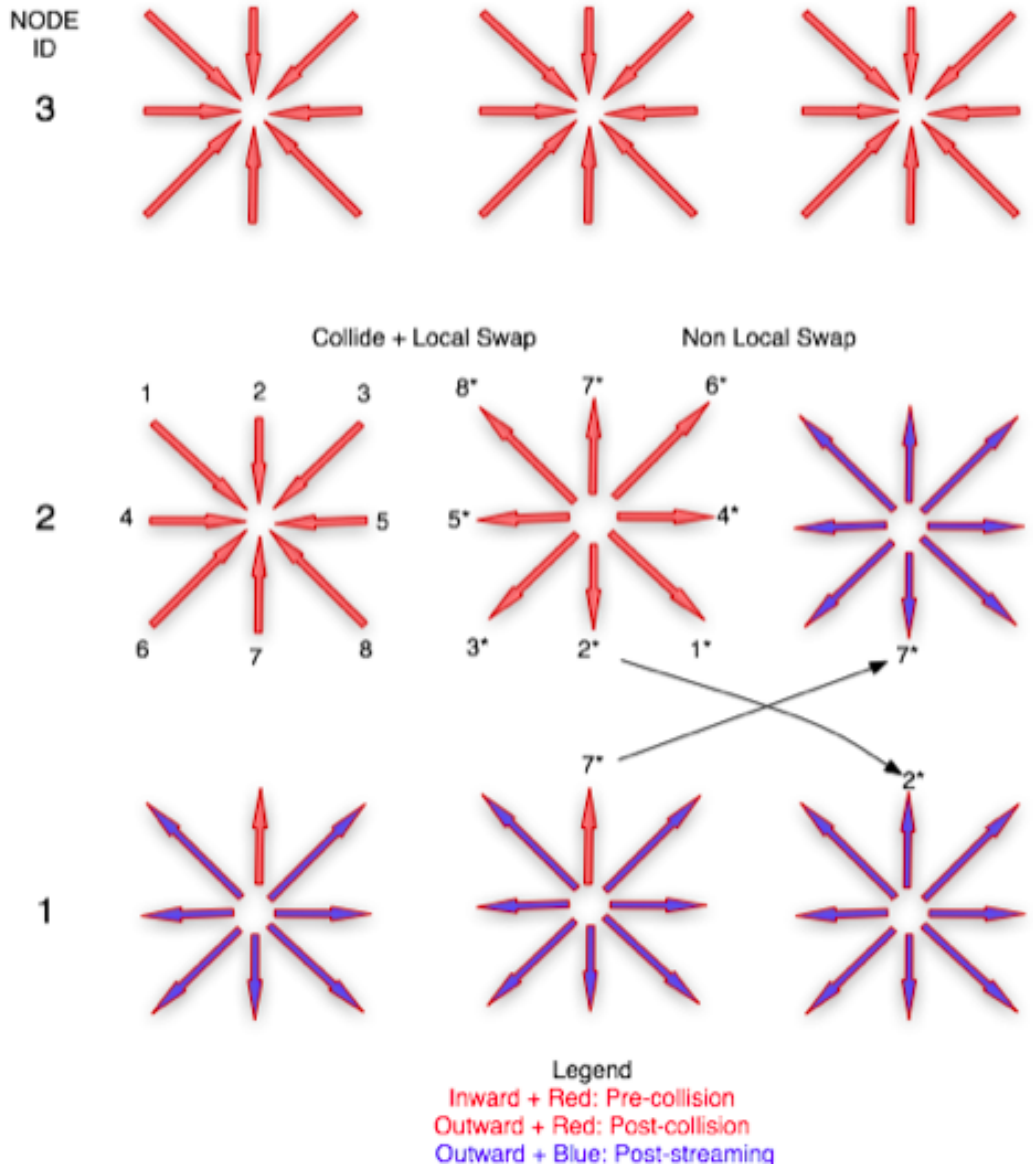

Figure: Populations read/write for consecutive nodes (grey cells).

# Single vs Double Buffer

We don't really need two arrays, we can read post-collisionaal and write the post-streaming populations on the same array

A swap trick allows to save a factor 2 in memory

It's more intricate to program and bears some limitations (say on threads)



# LBM on CPU/GPU: code management

How to write code that runs on CPUs and GPUs ?

Memory layout is different and thread model is different. Thus: write and mantain two codes

But CPU code can be re-used as much as possible.

Good practice is:

- Use classes and functions with a direct mirror on the GPU side
- Choose CPU code as the preferred development system

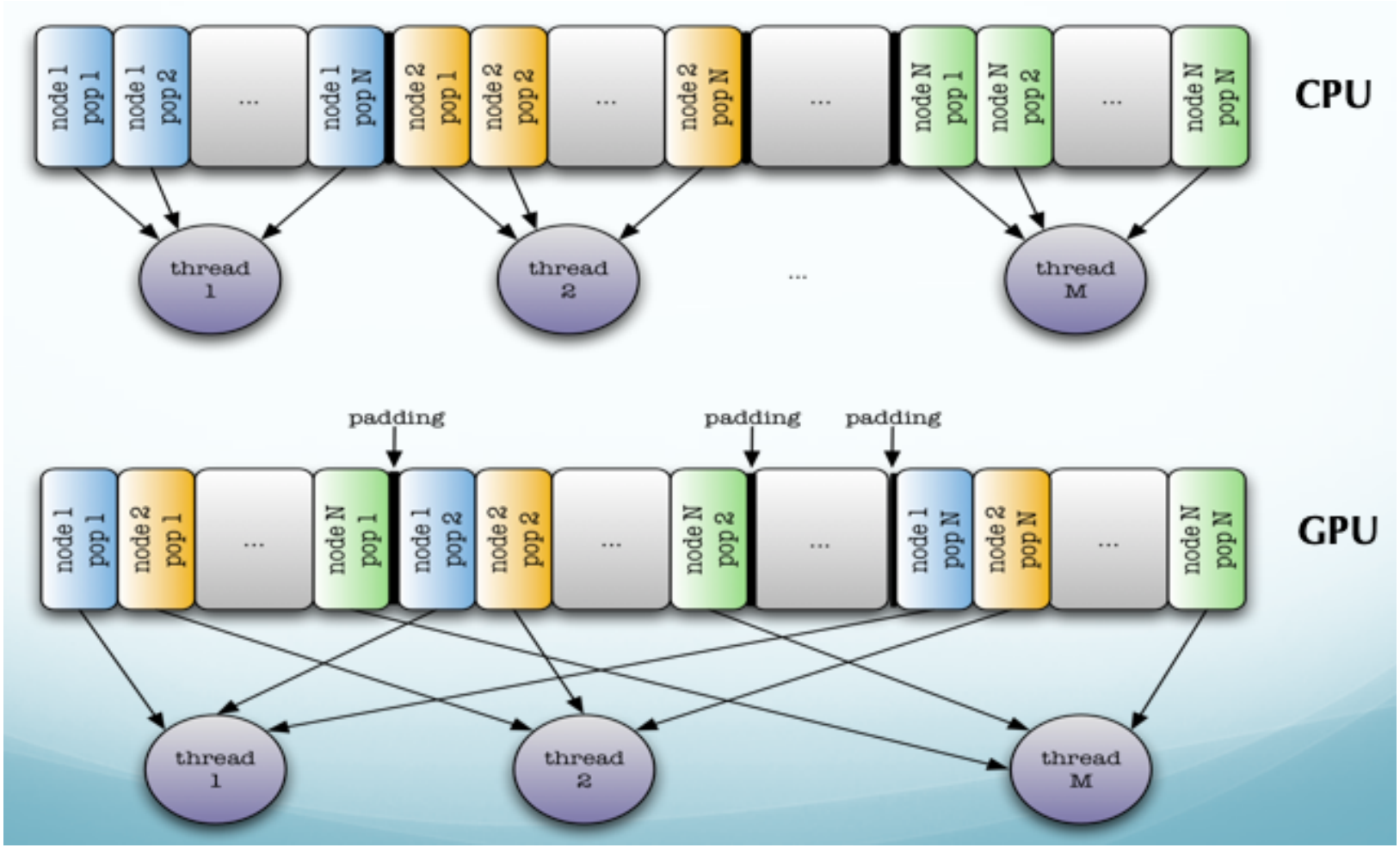
# LBM on CPU/GPU(2)

CPU computing is still mainstream in industrial sw for engineering:

- Legacy software is everywhere
- Developing and mantaining a double code is unwieldly
- Costs and turnarountime still ok (but not for medicine)
- ...

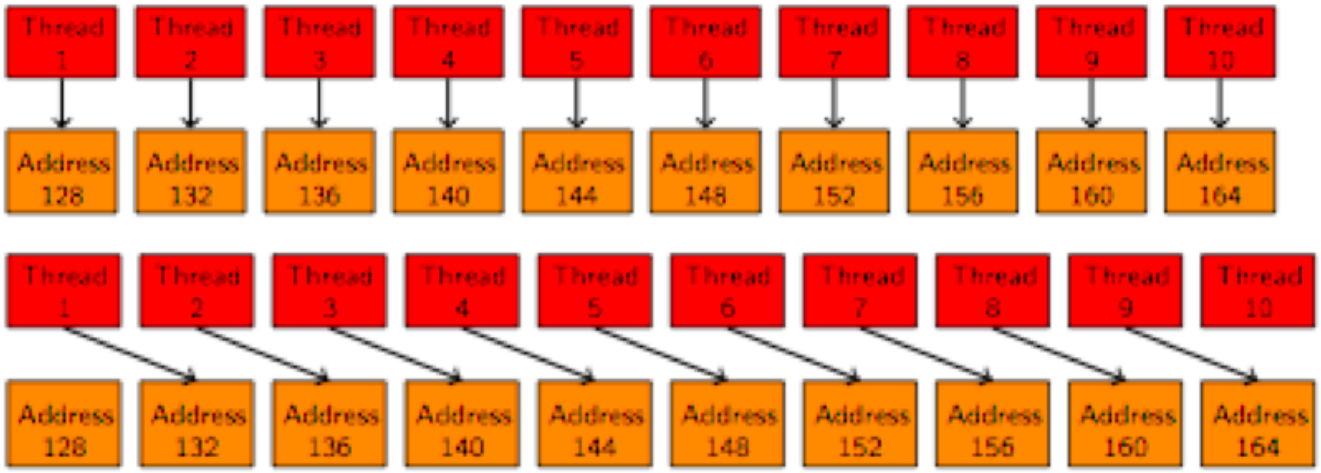
But demand for GPU is rapidly increasing and things change rapidly !

# Data layout & strategy




CPU vs GPU layout reflects the basic need for coalesced memory access on shared memory.

# Coalesced vs uncoalesced




Misaligned starting address


Non sequential access


Wrong size (12 bytes)

# Step 1: Load populations and connectivity data

```
// loop on all fluid nodes. Num_threads run in parallel
/* poffset1, poffset2, ..., poffset18 are the offsets of populations
laid out as 1D array */

for(ifl=istart+tid; ifl<=nfluidppl; ifl+=num_threads) {

    //load current values of the fluid populations
    pop0 = currpop[ifl];
    pop1 = currpop[poffset1+ifl];
    ......
    pop18 = currpop[poffset18+ifl];

    //load entries of the connectivity matrix for the node
    off1 = conn[poffset1+ifl];
    off2 = conn[poffset2+ifl]; ...
    off18 = conn[poffset18+ifl];
```

# Step 2: evaluate hydrodynamics variables

tl1_5 = pop1+pop2+pop3+pop4+pop5;
tl6_10 = pop6+pop7+pop8+pop9+pop10;
tl15_17 = pop15-pop17;
tl16_18 = pop16-pop18;

rho = pop0+pop11+pop12+pop13+pop14+
pop15+pop16+pop17+pop18+ tl1_5+tl6_10;

ju = pop2-pop3-pop7+pop8+pop11-pop12+ tl15_17-tl16_18;
jv = pop4-pop5-pop9+pop10+pop13-pop14+ tl15_17+tl16_18;
jw = tl1_5-tl6_10;

All variables are local and live on registers.

# Step 3: update populations (collision phase)

```
rhoi = 1./rho;

pop0 = oneom*pop0 +
          w[0]*(rho + rhoi*(gu2*qxx[0] + gv2*qyy[0] + gw2*qzz[0]));
pop1 = oneom*pop1 +
          w[1]* (rho + locsjw*cs2i + rhoi* (gu2*qxx[1] + gv2*qyy[1] +
gw2*qzz[1]));
......
pop18 = oneom*pop18 +
          w[18]*(rho + (ju - jv)*cs2i + rhoi* (gu2*qxx[18] + gv2*qyy[18]
+ gw2*qzz[18] + 2.*(gu*gv*qxy[18])));
```

omega, oneom=1-omega, qxx, qyy, qzz are stored in constant memory

# Step 4: copy back populations to target locations (streaming)

```
    newconf[ifl] = pop0;
    newconf[poffset1+off1] = pop1;
    newconf[poffset2+off2] = pop2;
    ...
    newconf[poffset18+off18] = pop18;
} // end of loop on fluid nodes

// swap currpop and newpop. Easy using pointers.
// start new iteration
```

Offsets off1, off2, . . . , off18 make most memory accesses to global memory uncoalesced.

# LBM on GPU: conclusions

- LBM carries out ~150 Million Lattice UPdates/Sec (MLUPS)
- the penalty due to uncoalesced memory accesses is very large
- there is a factor 6 of difference between the first and the second kernel alleviated by the substantial floating point operations of LBM (the difference between the third and the fourth kernel is only a factor 3)
- Regardless the number of uncoalesced memory accesses, indirect addressing requires a number of load operations that is 2x implementations using the full matrix representation.

# Sample GPU code

A sample LBM GPU code is available for you. Feel free to experiment once your LBM assignment is completed.

You can run rather large simulations:

- 19 populations x 2 buffers + hydrodynamic variables = 50 entries per LBM node.
- 16 GB / (50 x 8B) = 40M mesh nodes

# Single-GPU simulation

It's time to try MUPHY on a single GPU.

107_HEMO_BIFURCA.PARTITION.test contains a geometry for a bifurcated artery with 1 inlet and 2 outlets.
The slurm script jobgpu_serial.slurm is available.

All files have been uploaded, you can run the code and see what happens with the aid of paraview

# CPU vs GPU: MPI vs threads

A hybrid multi-GPU scheme does not have any major difficulty and MPI-based LBM is rather straightforward.

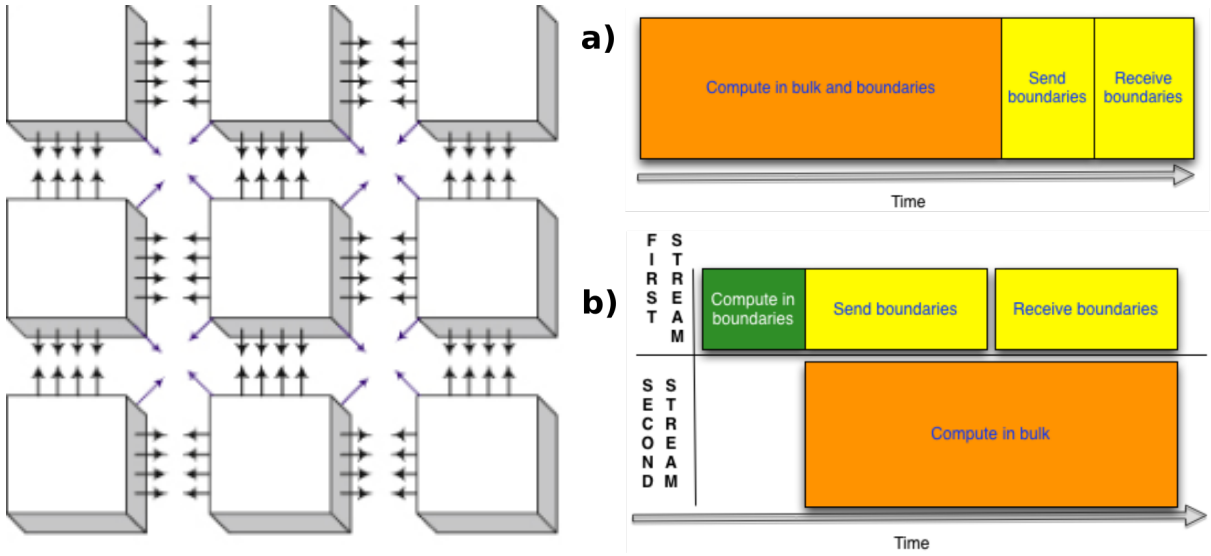



Figure: Regular Domain Decomposition

The halo regions will pass information only relative to DxQy structure, a rather small connectivity.
Communications are very efficient thanks to the cartesian regularity.

# Domain Decomposition for irregular geometries

Irregular domains pose a major challenge: how to guarantee load balancing?

Strong scalalability depends on the details of domain-domain interfaces. Hard to control and
optimal partitioning is an NP-complete problem.

# Domain Decomposition for irregular geometries(2)

Optimal partitioning can be performed by recognizing that:

- the LBM mesh is a graph in iself with quasi-regular connectivity
- graph bisection methods based on heuristics work very well (eg METIS)
- performances improve by further refining with flooding-based partitioning

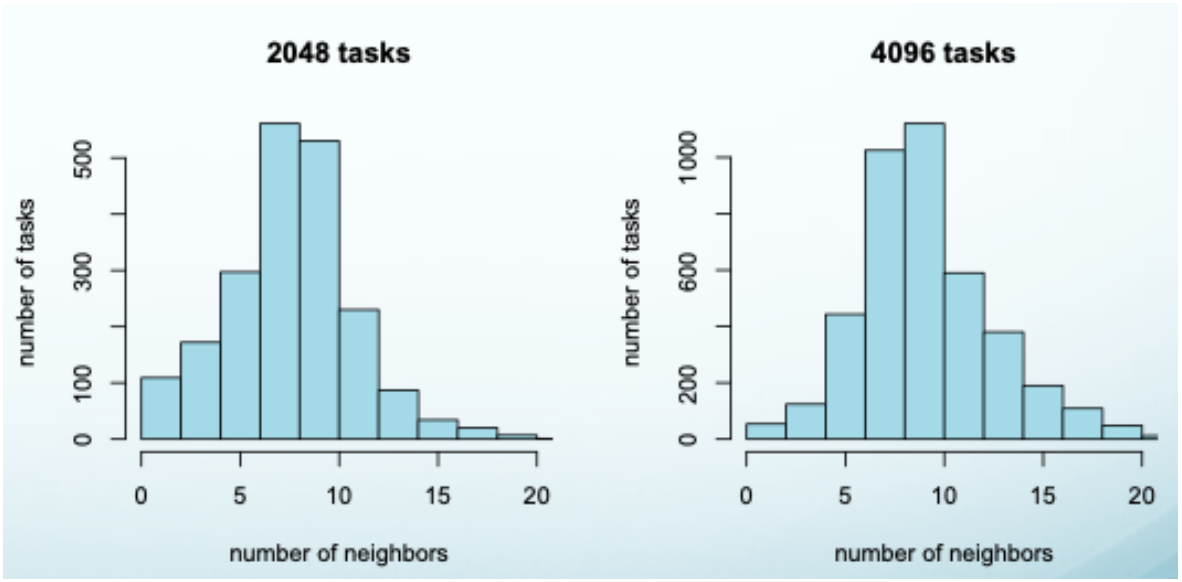



Figure: On doubling the number of partitions, interdomain connectivity remains constant

# Multi-GPU simulation

Multi GPU can be run via the script
    jobgpu_parallel.slurm
The input files are prepared for 4 irregular domains.

Regular domain decomposition can be used to check the differences in performances, by setting in run2.py:
    m.setDomainDecomposition(7)
which corresponds to decomposition in equal parallelepipeds.