# ForestClaw : Ghost filling and parallel communication

*Donna Calhoun (Boise State University)*

*Carsten Burstedde, Univ. of Bonn, Germany*

*p4est Summer School*

*July  20 - 25, 2020*

*Bonn, Germany (Virtual)*

# p4est interface for ForestClaw

Files that provide the interface between p4est and ForestClaw

- **src/forestclaw2d.h** - definitions of *patch*, *block,* and *domain* structs.
- **src/forestclaw2d.c** - nearest neighbors searches, transformations for multi-block boundaries, iterators, tagging
- **src/fclaw_base.c** - option handling utilities
- **src/fclaw2d_convenience.c** - multi-block domain definitions, routines for adapting and partitioning the domain
- Additional header files, and a few more files that provide mapping utilities for cubed sphere, torus, and so on.

ForestClaw is built on top of the routines in these files.

# Face neighbor searches

```
fclaw2d_patch_relation_t
    fclaw2d_patch_face_neighbors(fclaw2d_domain_t * domain,
                                 int blockno, int patchno, int faceno,
                                 int rproc[P4EST_HALF], int *rblockno,
                                 int rpatchno[P4EST_HALF], int *rfaceno)
{
    /* Returns neighbor type (BOUNDARY, HALFSIZE, SAMESIZE, DOUBLESIZE) */

    /* Additional output : MPI rank, patch number and block number for
       remote patch neighbors. */
}
```

- This is one of two essential routines needed to build ghost-filling infrastructure for ForestClaw.

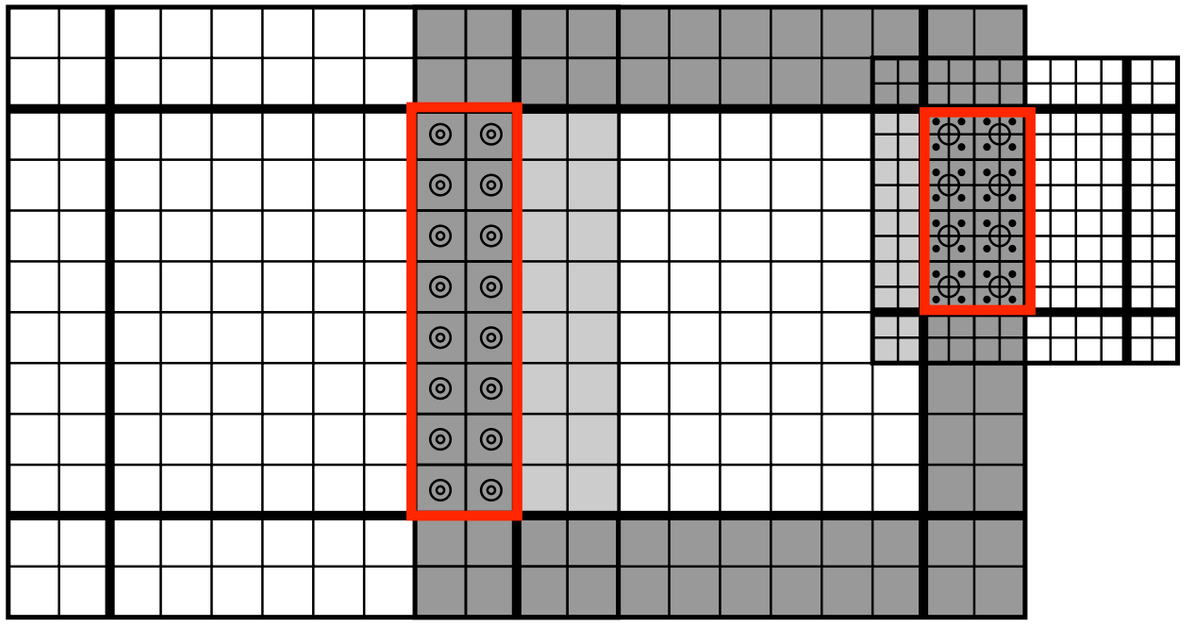
# Corner neighbor searches

```
int
fclaw2d_patch_corner_neighbors (fclaw2d_domain_t * domain,
                                int blockno, int patchno, int cornerno,
                                int *rproc, int *rblockno, int *rpatchno,
                                int *rcorner,
                                fclaw2d_patch_relation_t * neighbor_size)
{
    /* Returns 0,1 to indicate whether patch has a corner neighbor. */
}
```

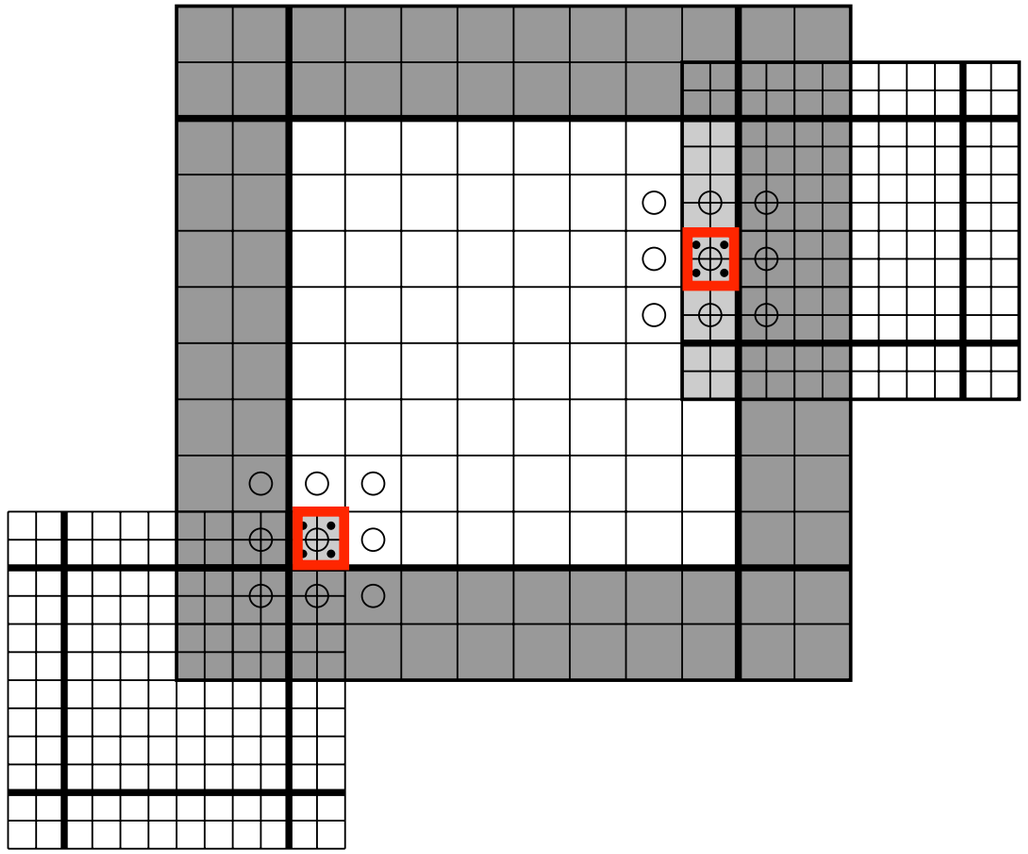
- Corner information needed for unsplit finite volume schemes.
- Corners exchange introduced some new challenges for parallel communication in p4est.

# Filling ghost cells

Assume valid data in the interior of each patch



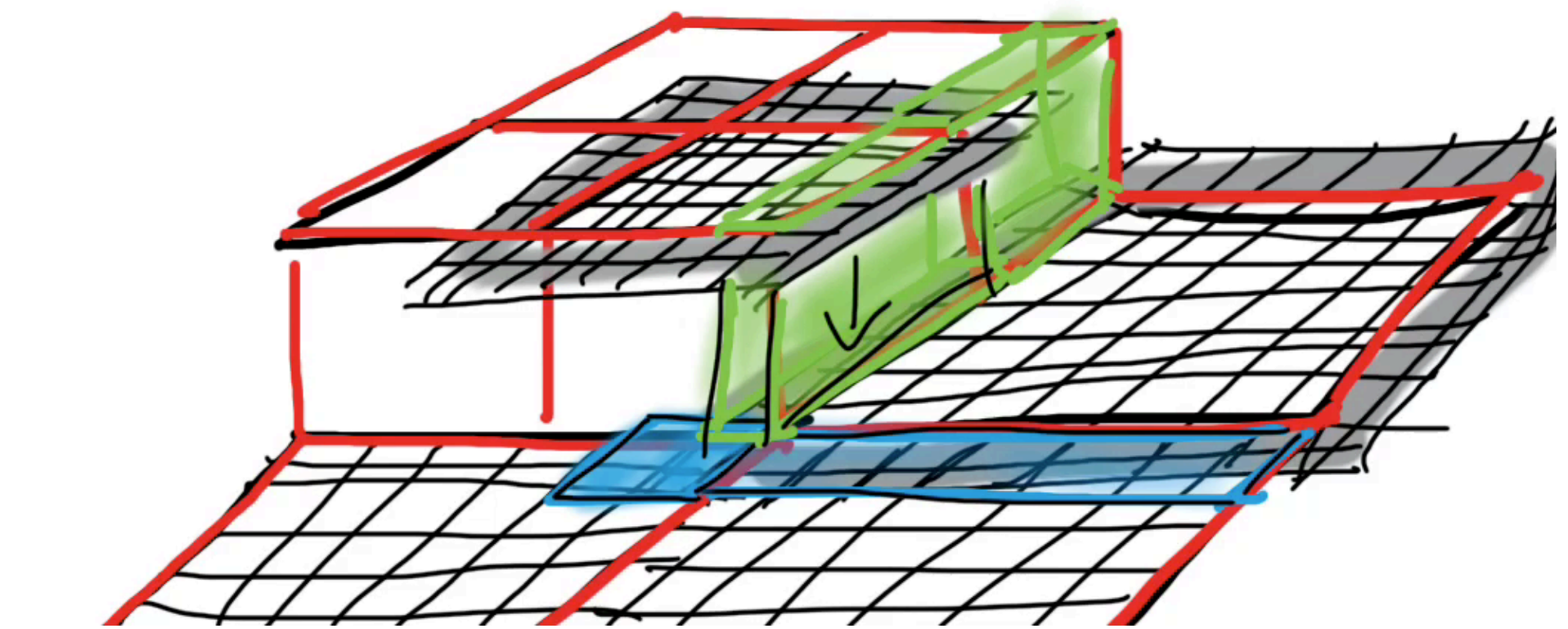
Step 1 : Fill "coarse grid" cells. Copy between same size neighbors; Average from fine grid to coarse grid.



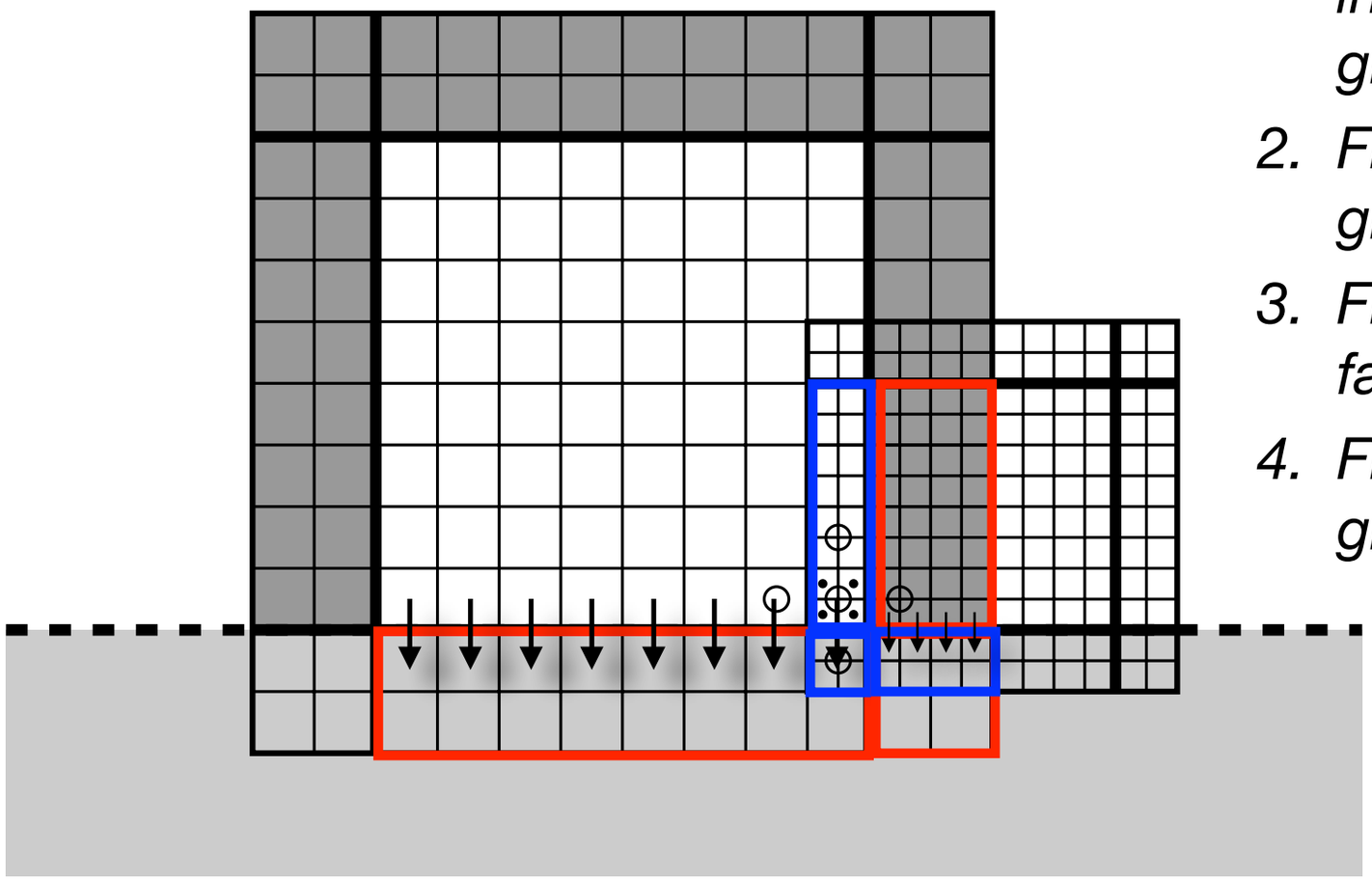
Step 2 : Interpolate from coarse grid to  to fine ghost regions, using coarse grid ghost regions

*Unsplit version of finite volume wave-propagation algorithm requires corner exchanges. Two layers of ghost cells needed for limiting waves to avoid unphysical oscillations.*

# How are ghost cells filled?



# Physical boundary conditions



1. *Fill exterior and interior coarse face ghost regions*
2. *Fill exterior coarse grid corner region*
3. *Fill fine grid interior face region*
4. *Fill exterior corner ghost regions*

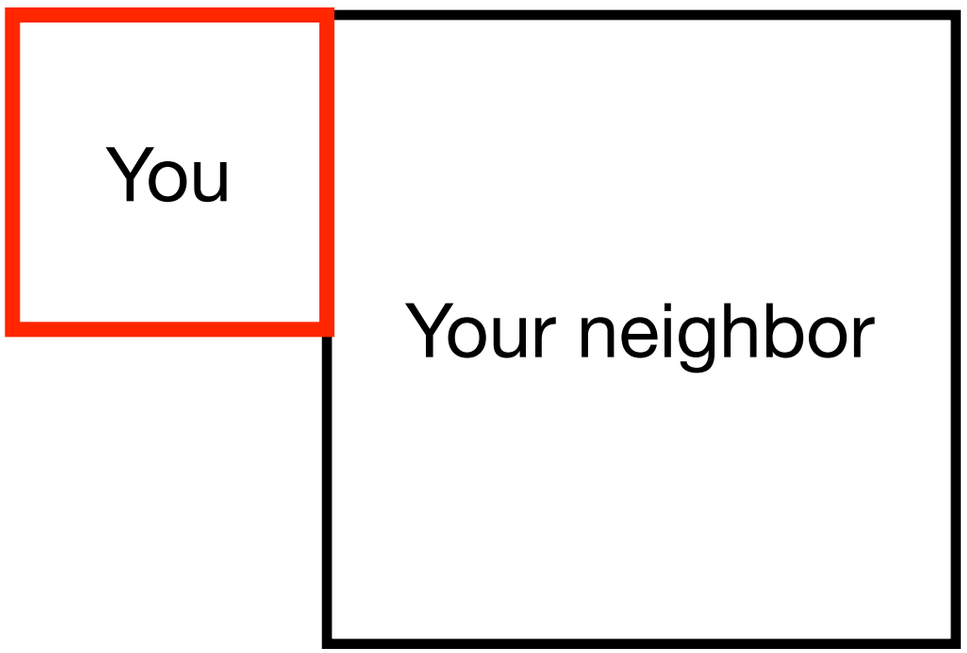
*Outside physical domain (exterior region)*

- Two passes of physical boundary conditions are required to fill corners in the exterior region.

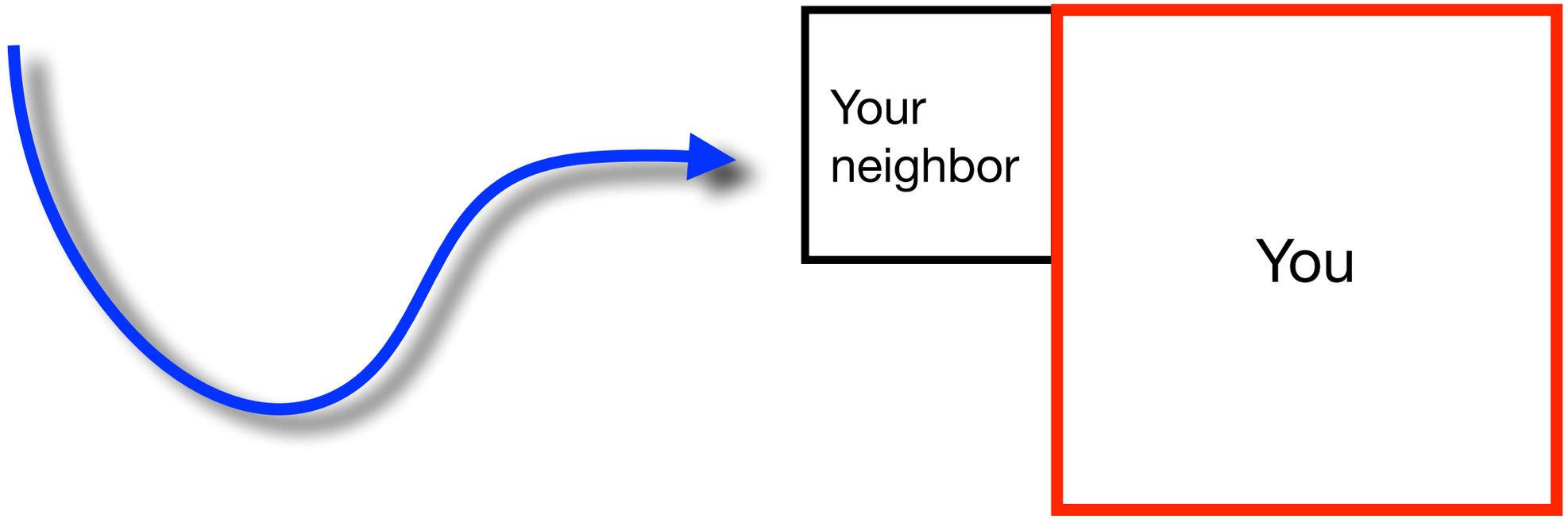
# Ghost filling

- Iterators used to iterate over patches
  - Sequencing very important, so multiple iterations over ghost cells are required.
- For each patch, nearest neighbors are queried.
  - Depending on stage in sequence, face or corner ghost regions may or may not be filled.
- 20 possible arrangements of a grid and neighbors (not including potential rotations at multi block boundaries) reduced to 12.  Trick : A grid with a double size neighbor is swapped with its neighbor.
- Routines for ghost-filling at faces are parameterized by direction (0,1), face (0,1,2,3), and neighbor type, so that only three routines are needed - one for copying, one for averaging, and one for interpolation.
- Routines for ghost-filling at corners are parameterized by corner number and neighbor type.  Three routines for copying, averaging and interpolation

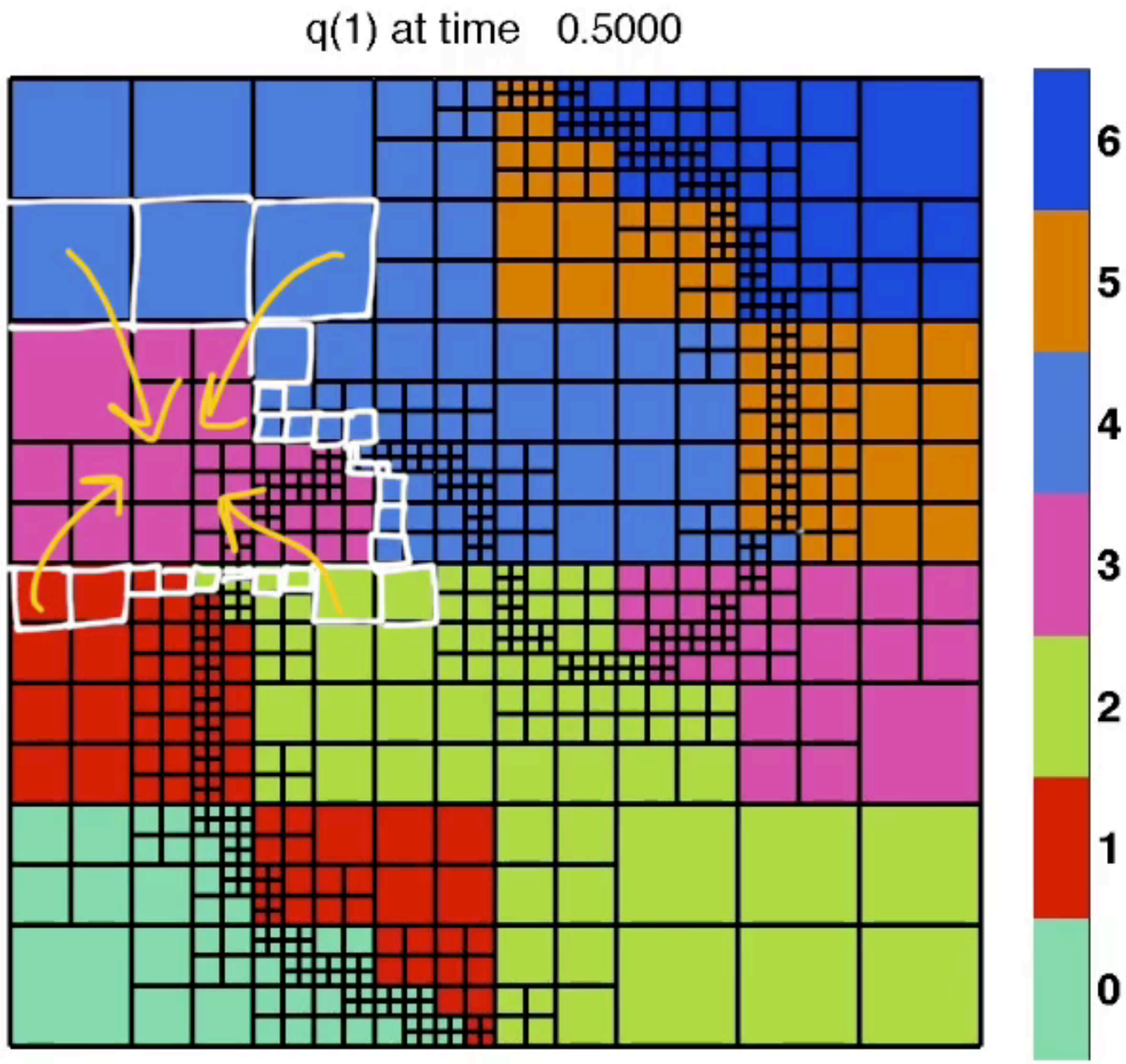
# “Context switching”




- “Context switching” allows us to reduce possible combinations of grid pairings.
- Uses a “swap” routine supplied by p4est so that face numbers are relative to “You” and not your neighbor.
- Works seamlessly with multi-block boundaries.

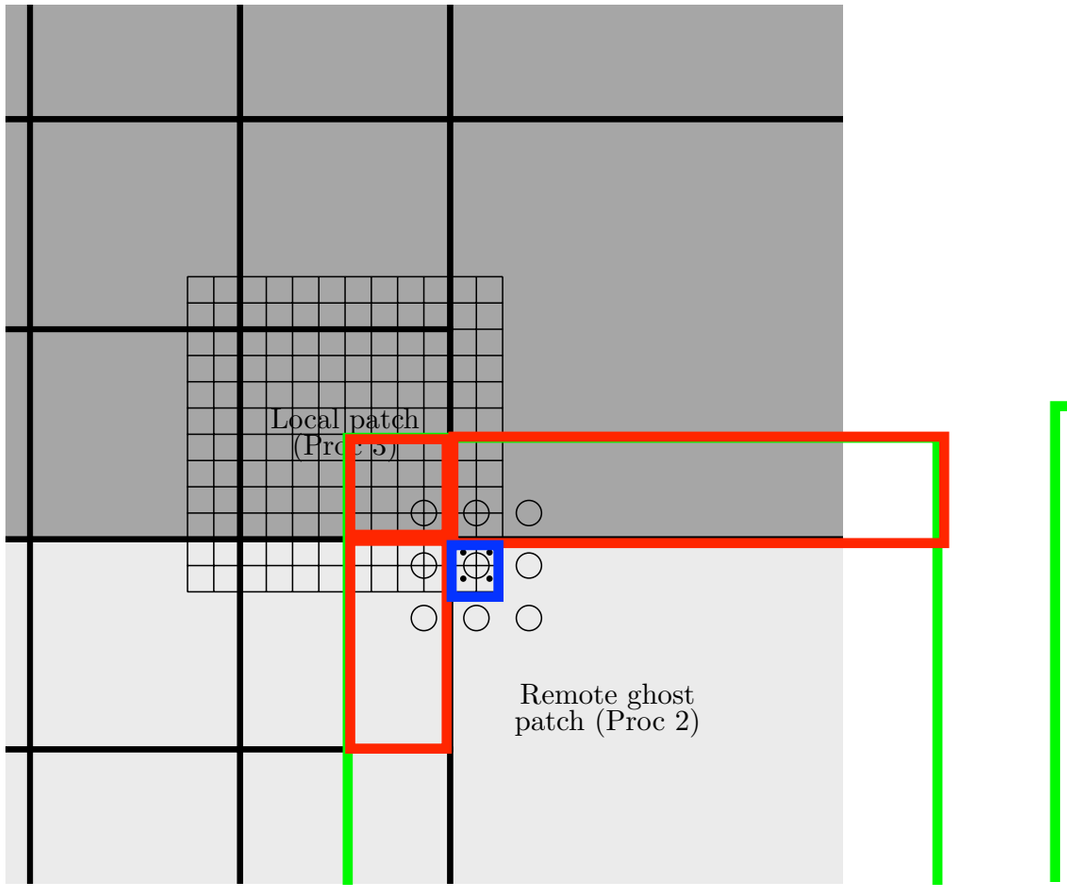


# Parallel ghost filling

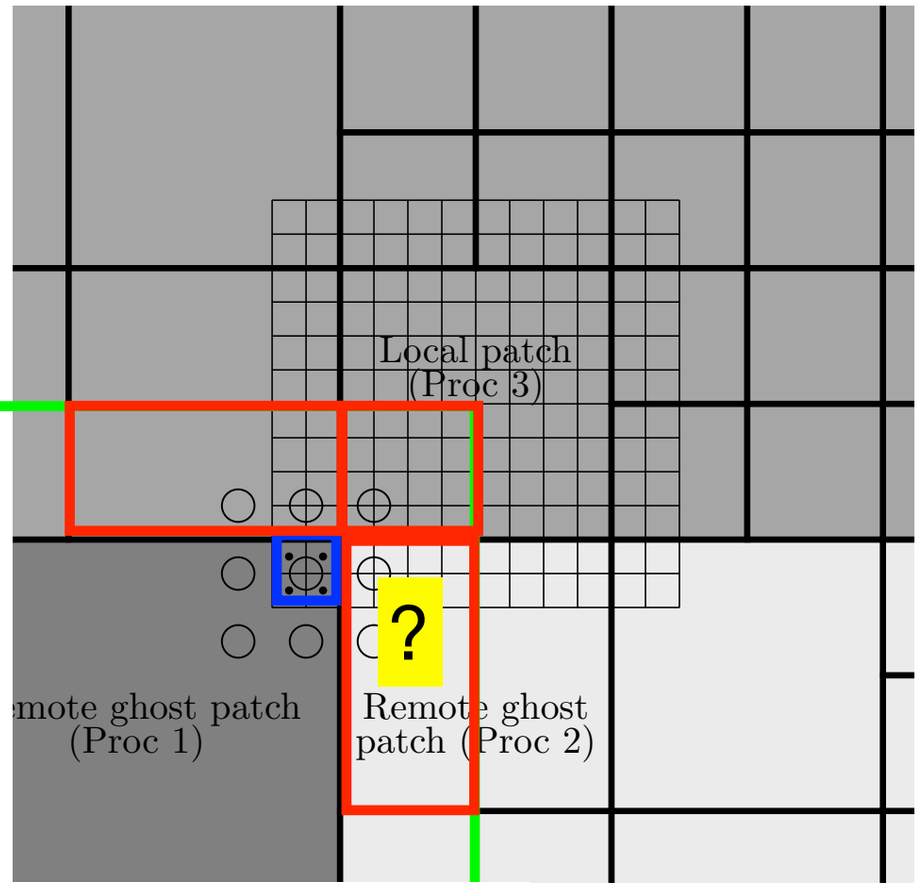



- Remote patches are created by p4est, and are stored in separate data structure
- Patch routines in ForestClaw are used to re-build essential information in ghost patches

# Parallel ghost filling algorithm



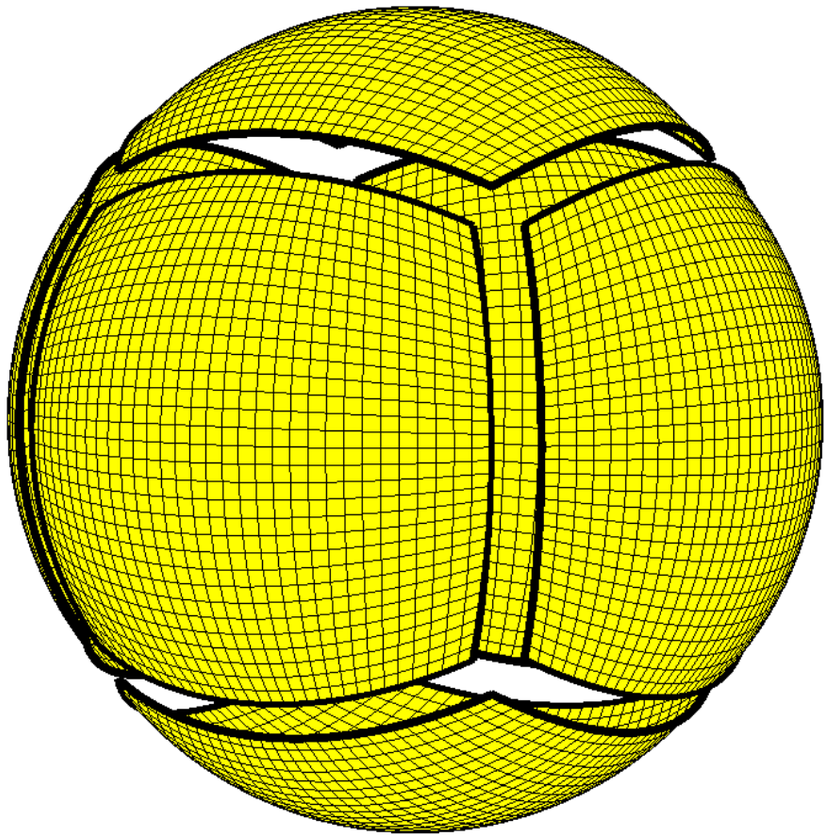

*Remote patches on processor boundary must exchange ghost cells before being sent to local processor*



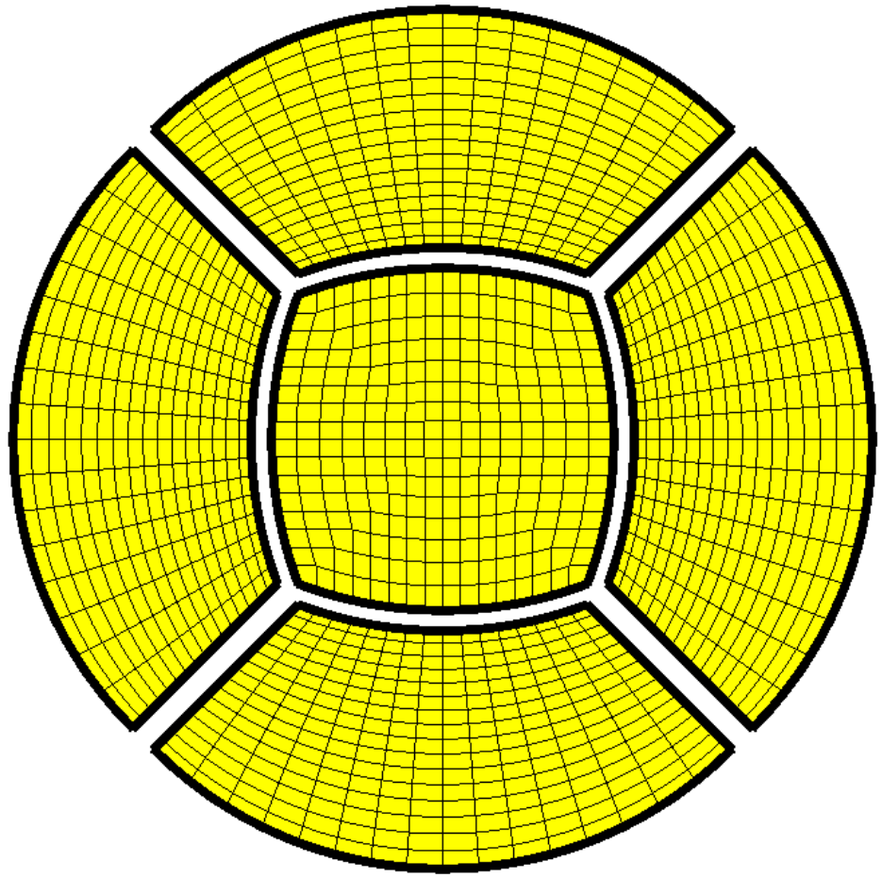

An lightweight indirect exchange is required between remote proc 2 and 3

- Remote patches must have valid coarse grid ghost data so that corners on local patches can be filled in.
- Requires one communication pass per ghost cell update

# Multiblock boundaries



- Ghost filling at multi-block boundaries is transparent to the user
- Requires index transformations supplied by p4est interface
- Straightforward to modify coarse/fine averaging and interpolation stencils, even at multi-block boundaries.



Questions?

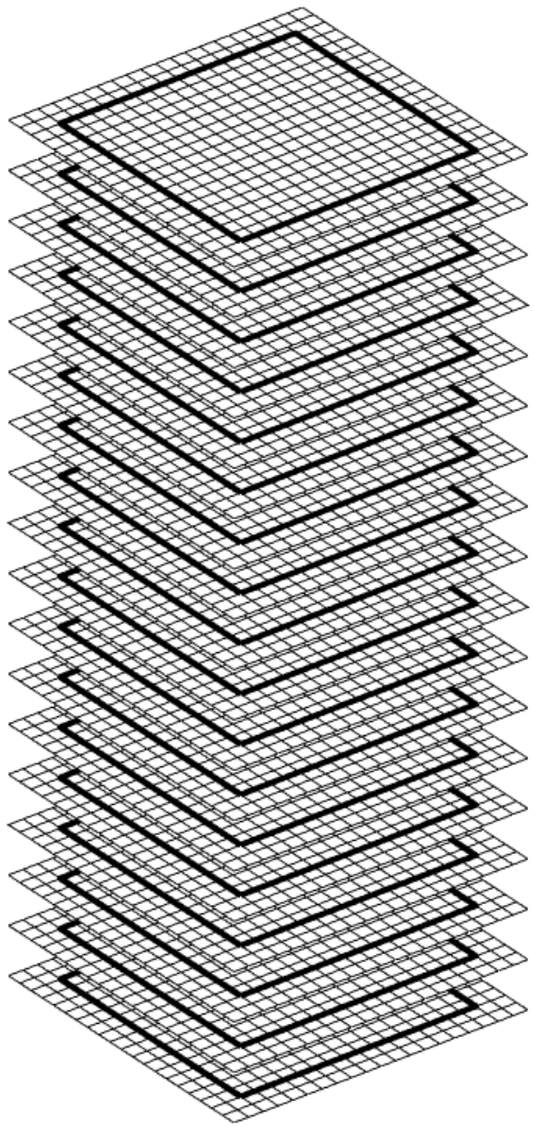
# ForestClaw on GPUs

Ported fully unsplit wave propagation algorithm for hyperbolic conservation laws (implemented in Clawpack) to CUDA.

- Copy time level solution on all patches to single contiguous block of CPU memory
- Copy contiguous block of CPU memory to the GPU.
- Configure the GPU to assign one 1d thread block to each single ForestClaw patch
- Divide shared memory equally among thread blocks=patches
- All solution data resides in global memory;  shared memory is only used for temporary data
- CUDA function pointers used to provide custom Riemann solvers.
- Best to use the 4.x (SOA)  data layout
- All core ForestClaw routines, and p4est remain on the CPU.  Only the patch update is ported to the GPU.

# ForestClaw on GPUs

```
dim3 grid(1,1,batch_size);
```



*One ForestClaw patch per CUDA block*

```
block_size = 128;  batch_size = 4000;
mwork = 9*meqn + 9*maux + mwaves + meqn*mwaves;
bytes_per_thread = sizeof(double)*mwork;
bytes = bytes_per_thread*block_size;

dim3 block(block_size,1,1);
dim3 grid(1,1,batch_size);

claw_flux2<<<grid,block,bytes>>>(mx,my,meqn,..)
```

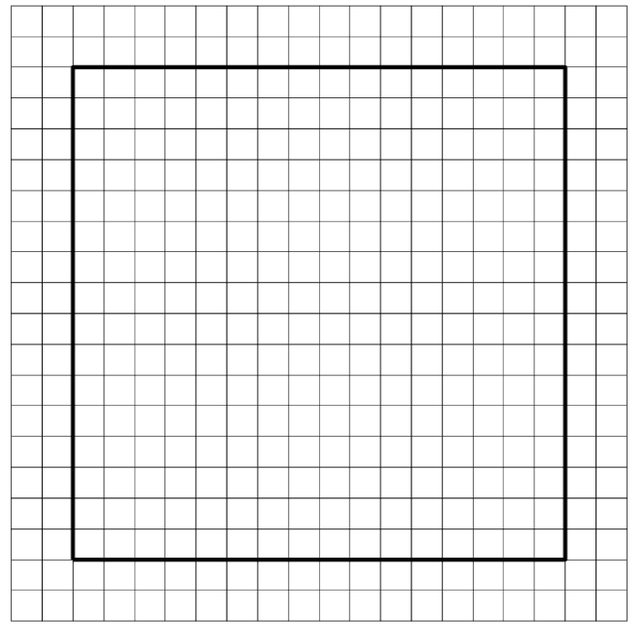
*1d thread blocks*

*3d grid*

*~4000 patches in a batch*

*~128 threads per block*

*Patch layout with valid ghost cell data*



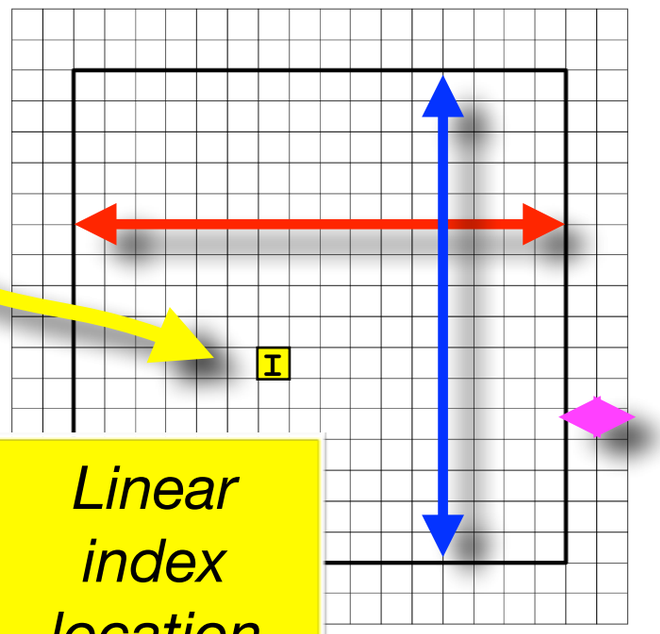
# Thread block - loop over faces

```
ys = (2*mbc + mx);    /* Stride */
ifaces_x = mx + 2*mbc-1;
ifaces_y = my + 2*mbc-1;
num_cells = ifaces_x*ifaces_y;

for(ti = threadIdx.x; ti < num_ifaces; ti += blockDim.x)
{
  ix = ti % ifaces_x;
  iy = ti/ifaces_x;

  I = (iy + 1)*ys + (ix + 1);
    ....
```

*Solve a normal Riemann problem at each face; include 1 ghost cell in each direction*



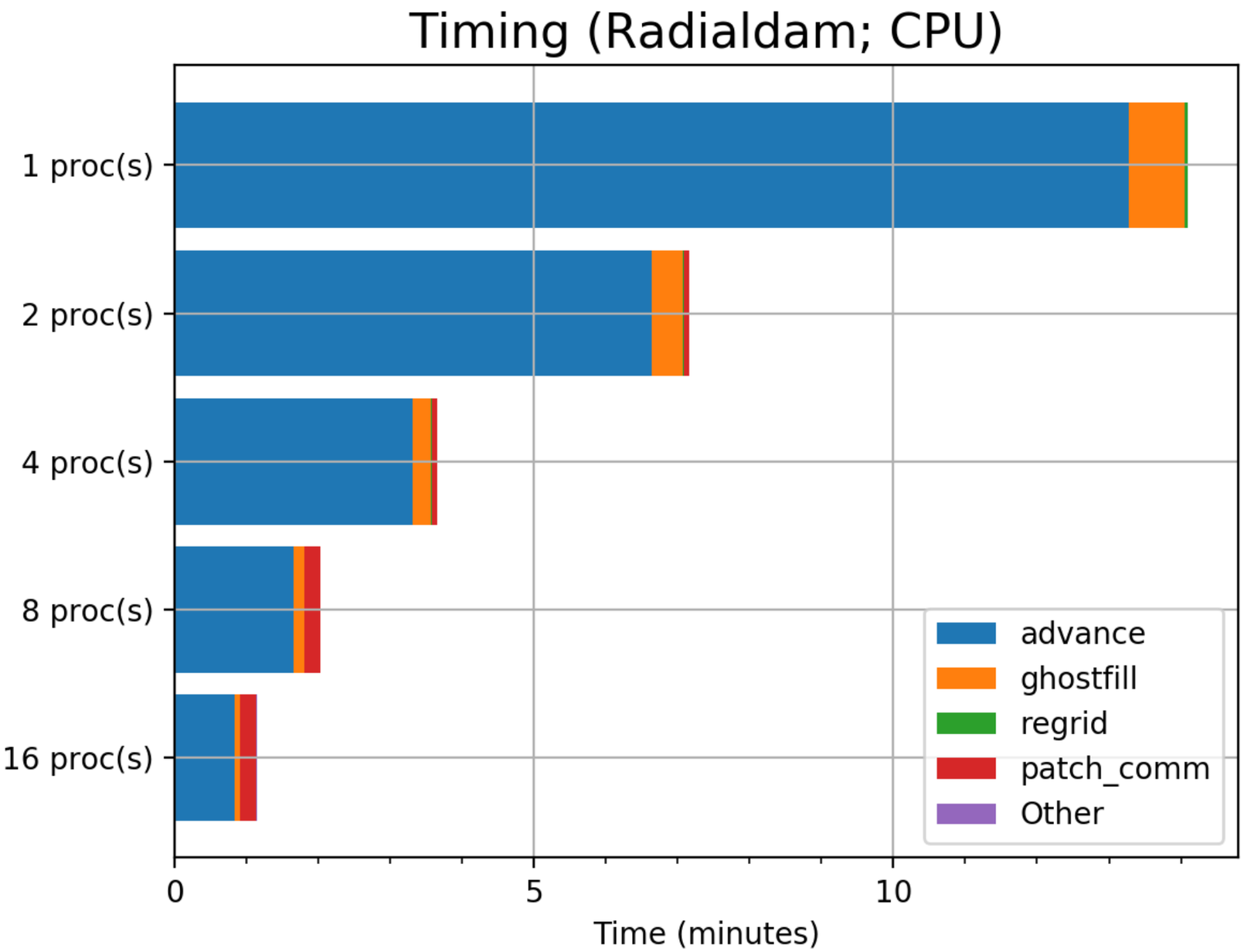

*Linear index location*

*mx : Number of interior grid cells in x*

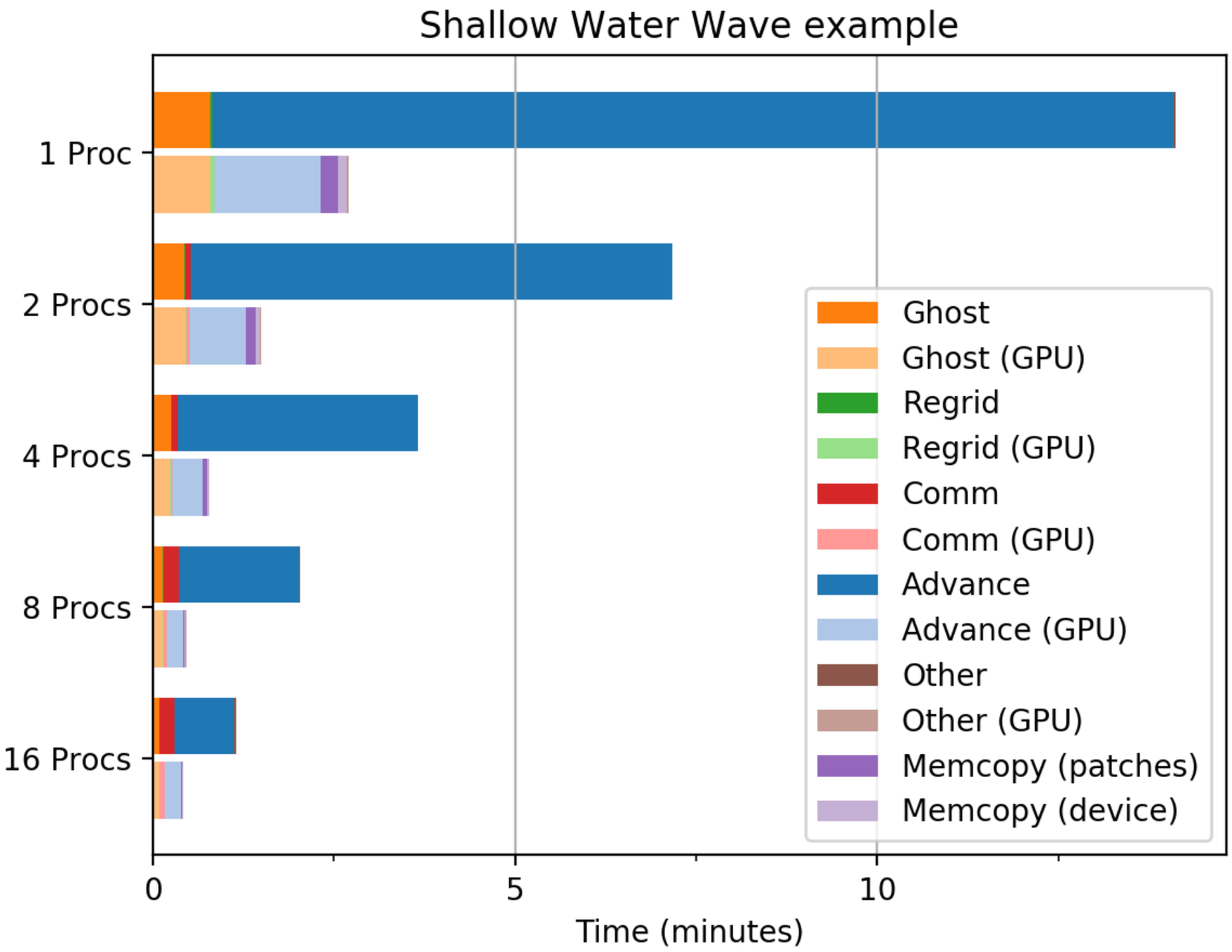
*my : Number of interior grid cells in y*

*mbc : Number of ghost cells*

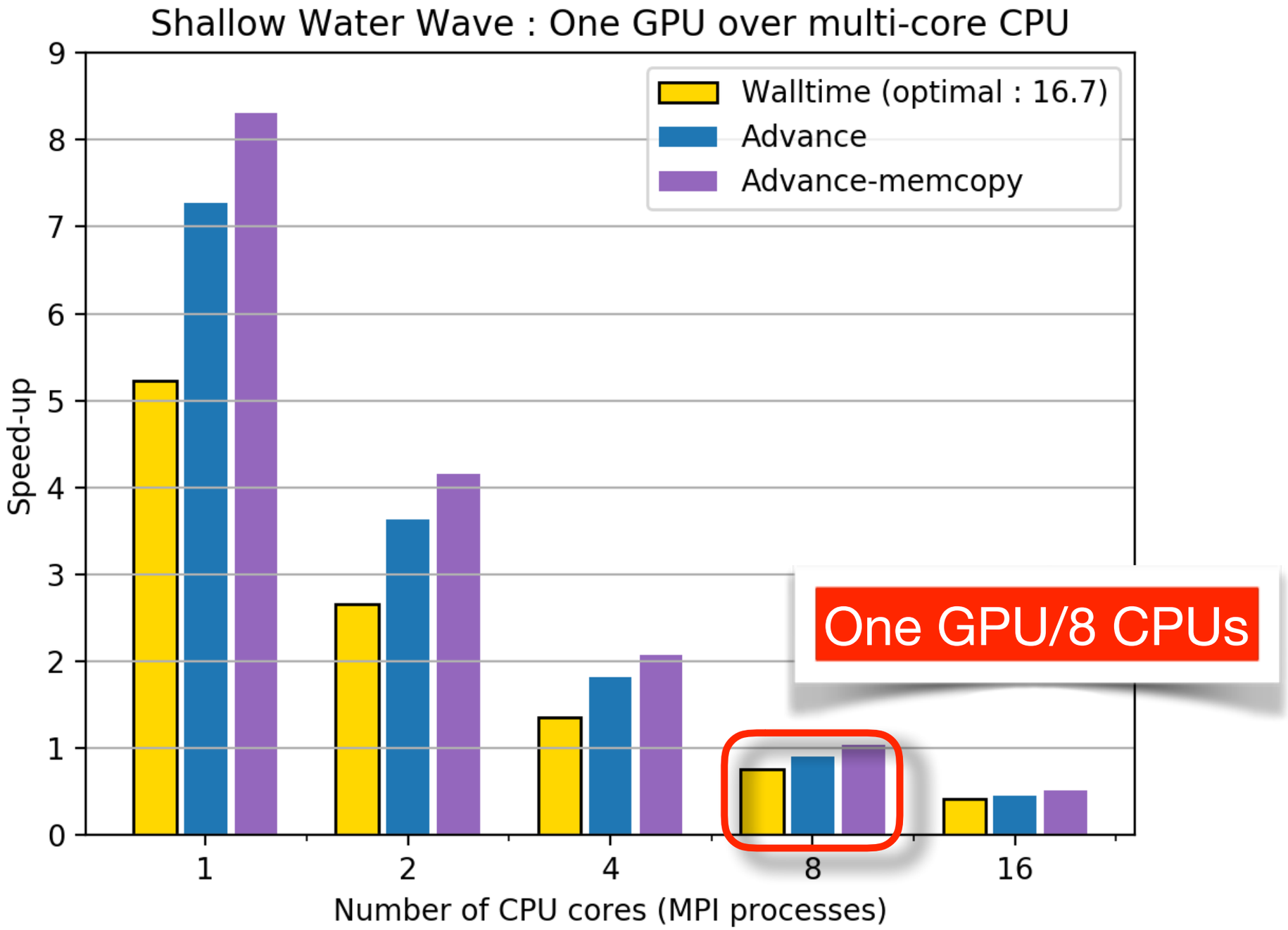
# Shallow water



# Shallow water

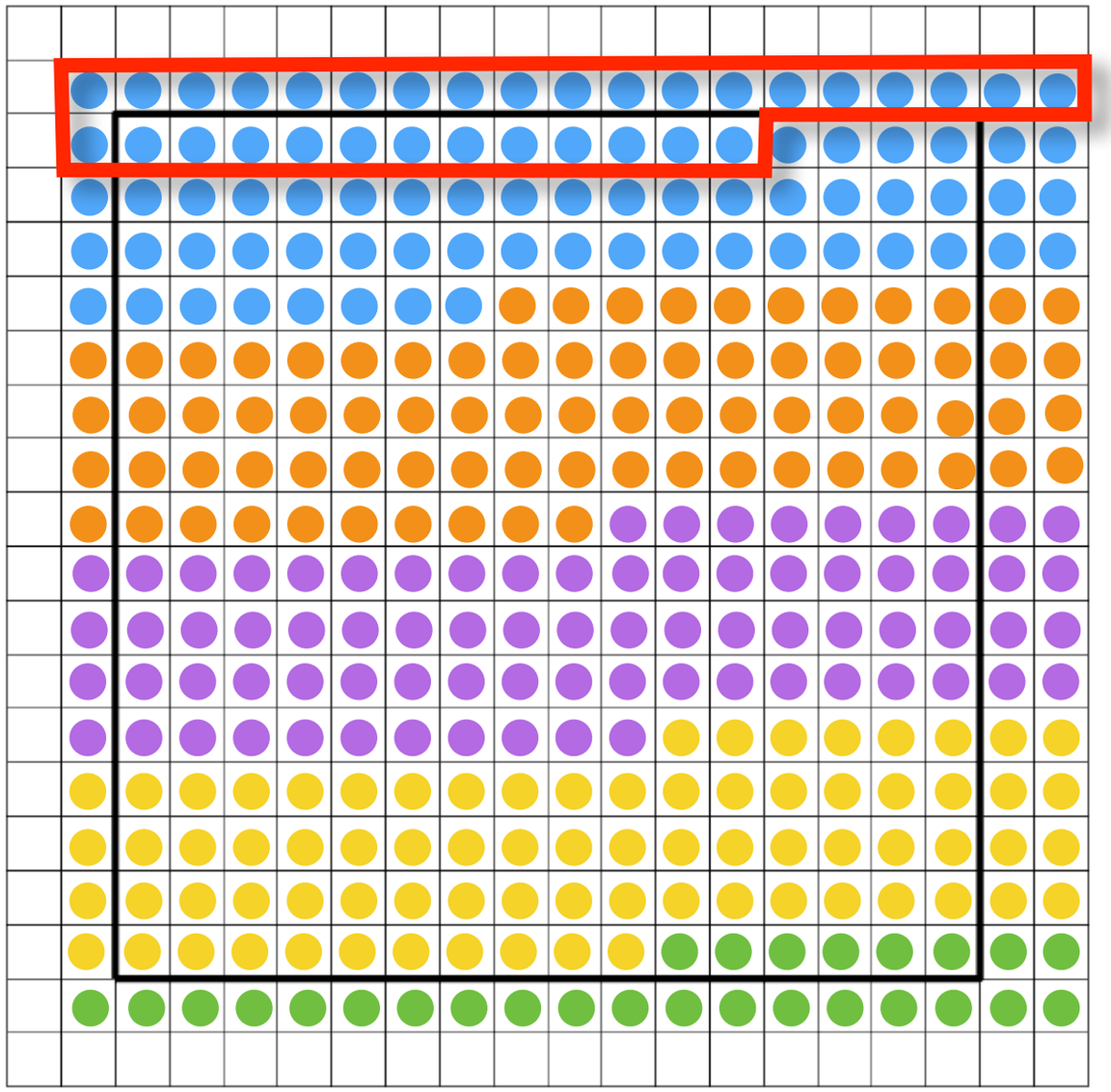


# Shallow water



# One dimensional thread block

- No block synchronization required
- Typical patch sizes are 32x32
- Number of threads per patch : ~128, depending on shared memory requirements



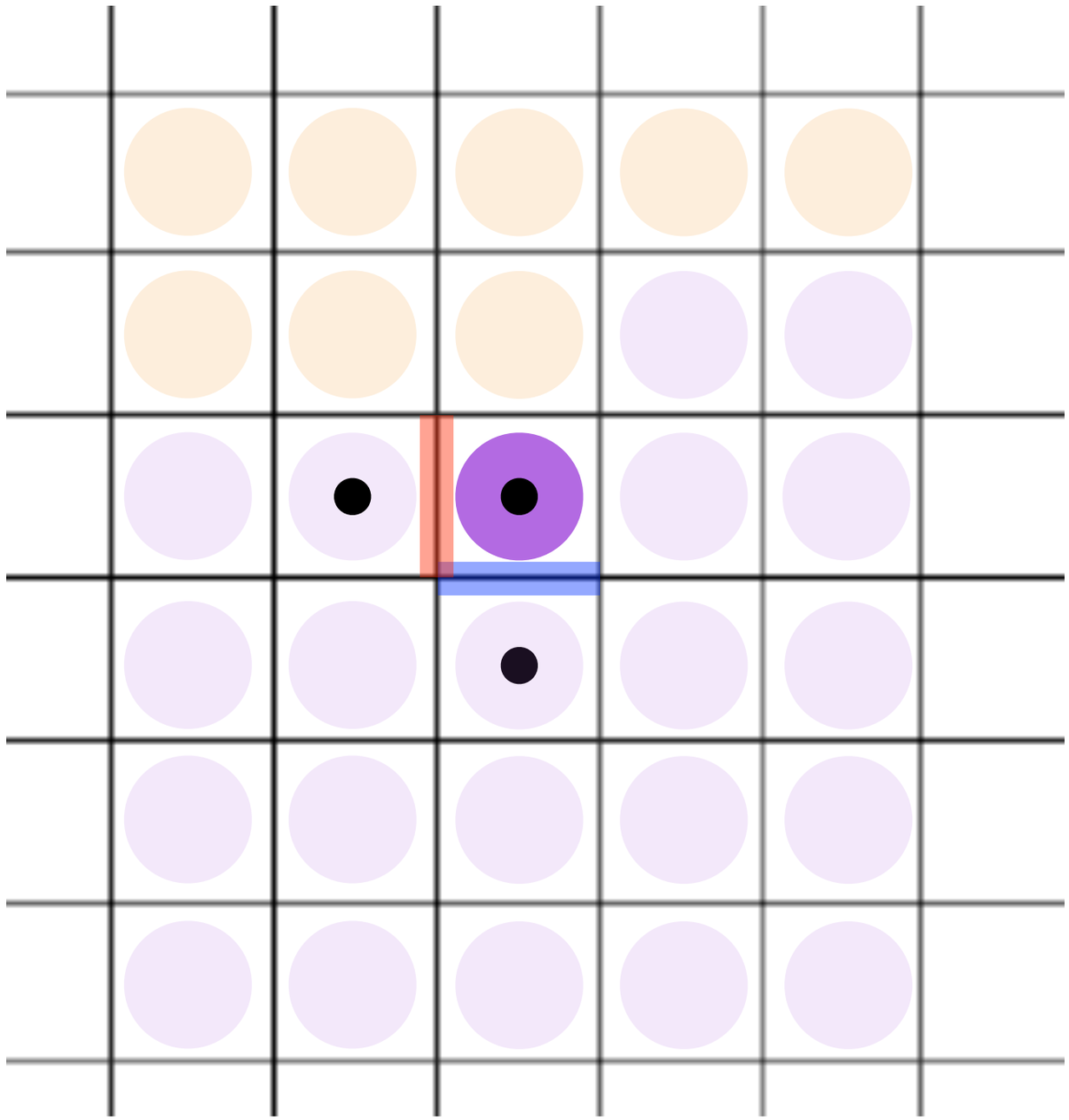
- *First pass*
- *Second pass*
- *Third pass*
- *Fourth pass*
- *Fifth pass*

Warp = 32 threads

Solve Riemann problems at x and y faces

# Normal Riemann problems

- Each thread makes a local copy of global data and stores it in shared memory.
- Fluxes computed Riemann problems stored in global array



Thread

Global solution data accessed by thread

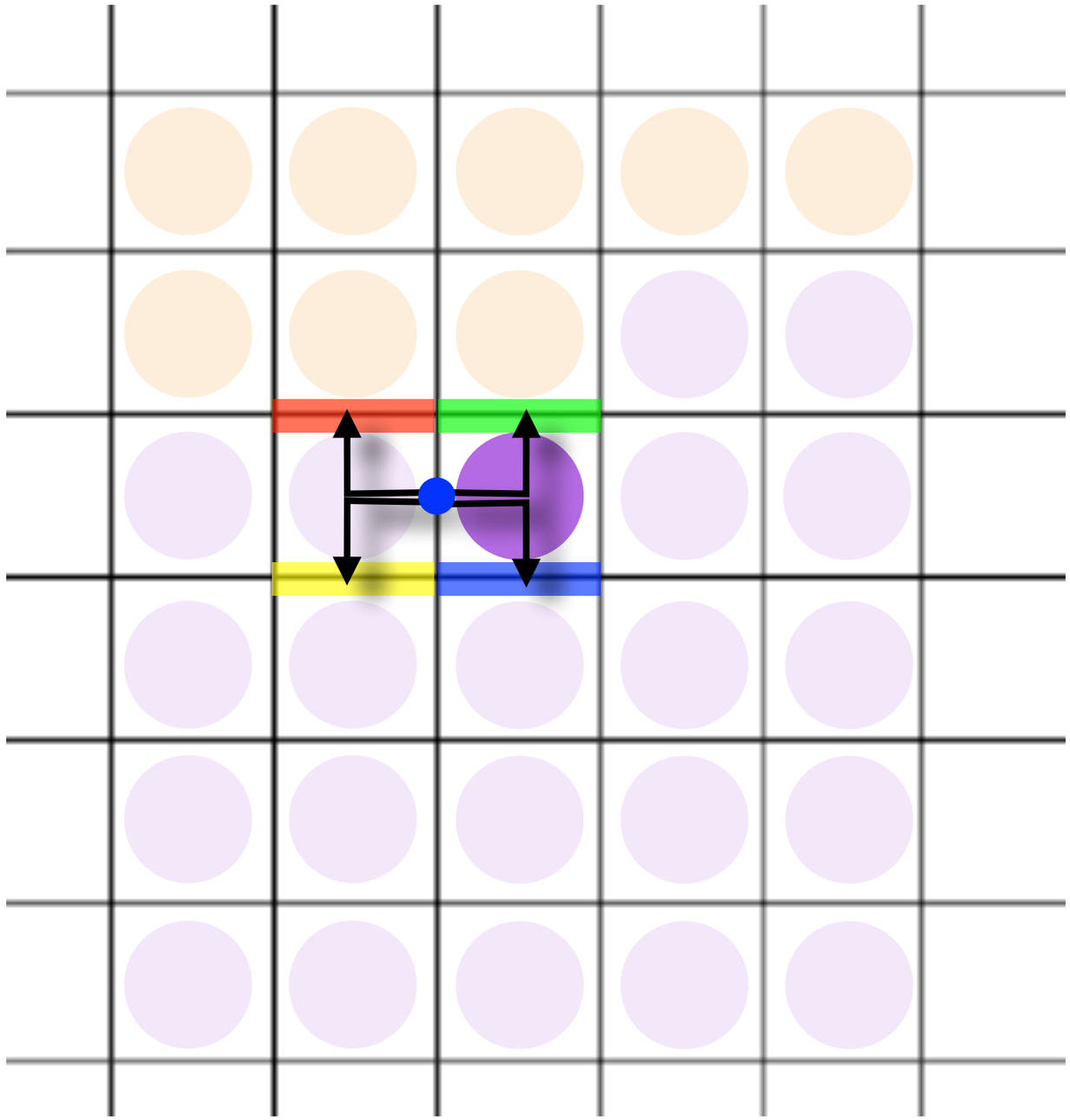
Fluxes computed at an x interface

Fluxes computed at a y interface

Fluxes are computed by solving Riemann problems

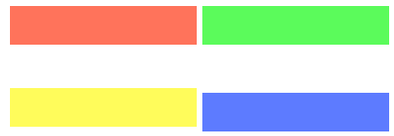
# Unsplit algorithm

- Each transverse solve stores data in the same global memory space
- To avoid data collisions with other threads writing to the same global memory, four passes over all the global data are required, one for each “color”
- Sync threads between each pass



Thread

Global data stored at cell interfaces

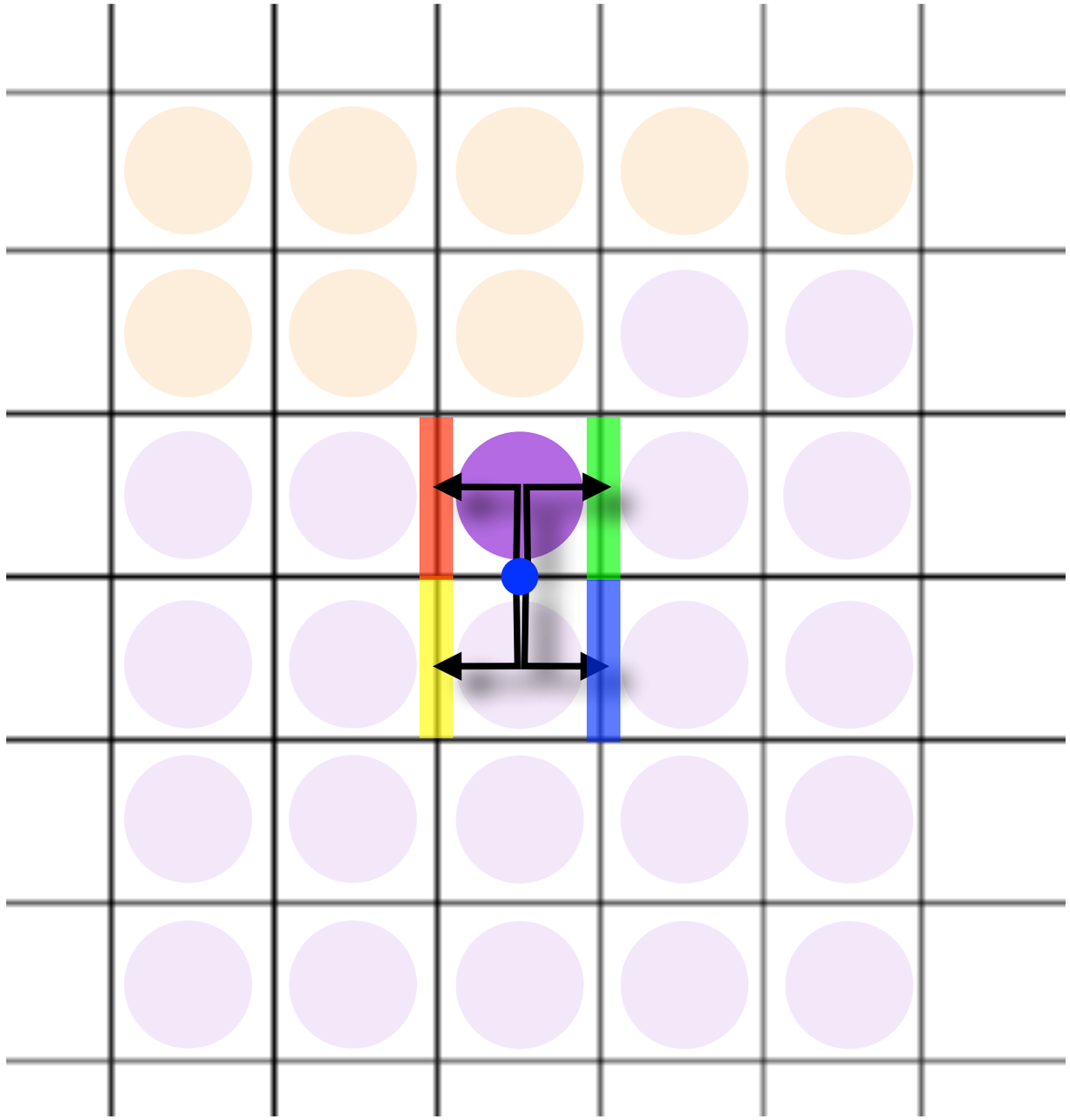


Results from a horizontal normal Riemann problems are propagated in the vertical direction

Fluxes are computed by solving Riemann problems

# Transverse Riemann problems

- Each transverse solve stores data in the same global memory space
- Four more passes over all the global data are required



Thread

Global data accessed by thread

Results from the vertical Riemann problems are then propagated in a horizontal direction

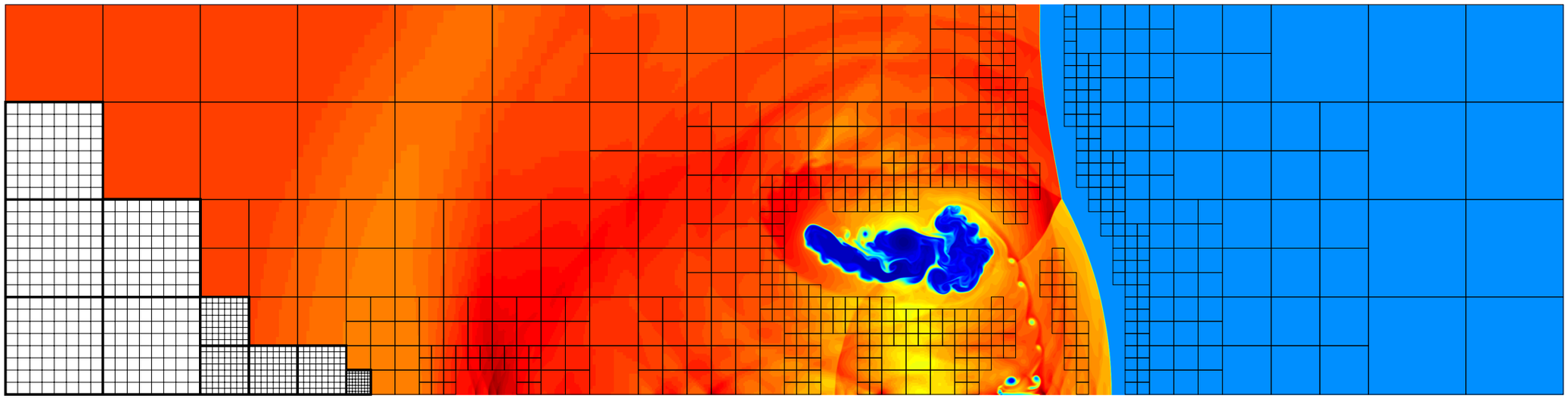
Fluxes are computed by solving Riemann problems

# Unsplit wave propagation

Fully unsplit wave propagation algorithm is implemented in a single CUDA kernel.

- While more expensive than the dimensionally split version, the unsplit algorithm may be more suited to AMR.
- The cost in CUDA is that parts of the code that can be done together are are now split to avoid race conditions. *Maybe we can improve on this by using more global memory?*
- Our GPU configuration does not require any synchronization between thread blocks
- Since all patches are the same size, they can be processed in large batches (O(1000) per batch)
- All AMR tasks including filling ghost cells is done on the CPU
- Conservation requires extra memory copy from device.

# Shock-bubble problem



- Euler equations : Four field variables per finite volume cell
- Riemann solvers written in CUDA and passed in as CUDA pointers
- 32x32 patch sizes seem optimal
- Ran on 4 node (4 cores per node) cluster with 2 GeForce Titan X (2015) per node
- CPU and GPU results agree to machine precision

Related work :

H. G. Ohannessian, G. Turkiyyah, A. J. Ahmadia, and D. I. Ketcheson, CUDACLAW: A high-performance programmable GPU framework for the solution of hyperbolic PDEs, arXiv, 1805.08846 (2018).

X. Qin, R. J. LeVeque, M. Motley, “Accelerating wave-propagation algorithms on adaptive mesh with the graphics processing unit (GPU)”, 2018.

# EuroHack 2018 - Lugano, Switzerland


*Scott Aiton (BSU), Andreas Jocksch (CSCS), Xinsheng Qin (Univ. of Washington), D. Calhoun (BSU), Melody Shih (NYU)*

Sponsored by : NVIDIA + Swiss National Computing Center