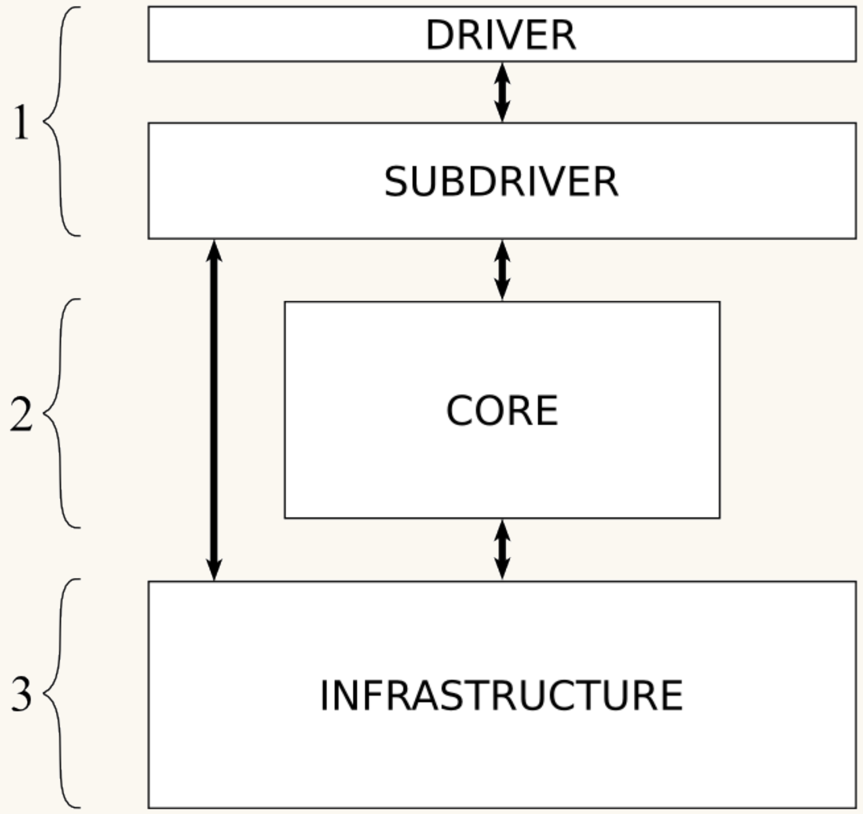
MPAS IMPLEMENTATION OVERVIEW

MPAS SOFTWARE ARCHITECTURE



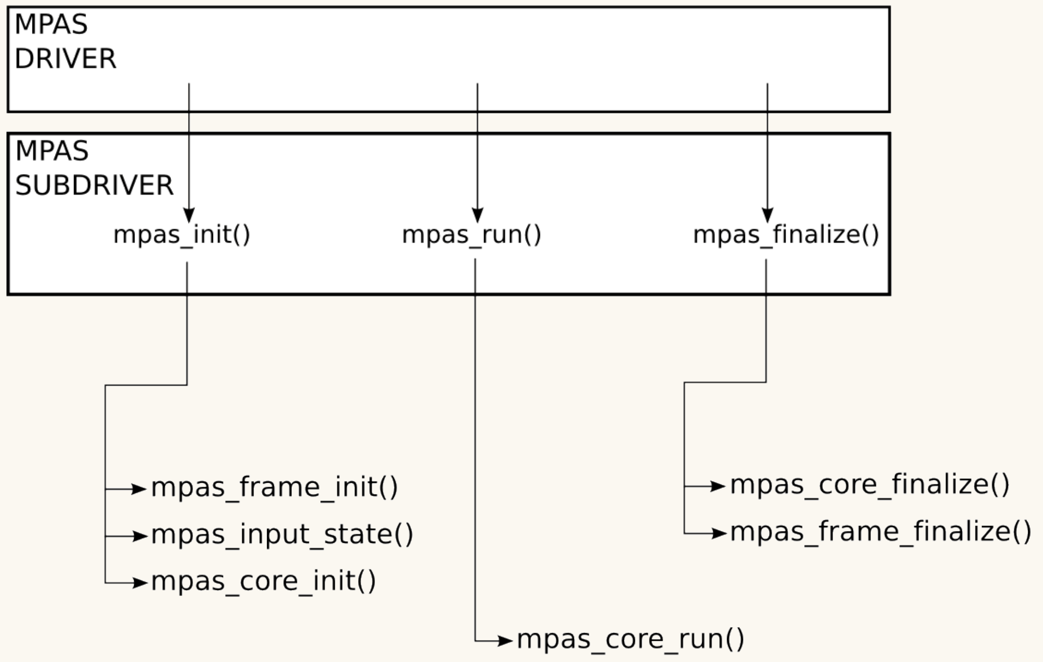
***Arrows** indicate interaction between components of the MPAS architecture*

1. **Driver layer** – The high-level DRIVER and SUBDRIVER interact with the core through a generic interface.

2. **MPAS core** – The MPAS CORE performs the computational work of MPAS, employing data structures and functionality of the INFRASTRUCTURE.

3. **Infrastructure** – The infrastructure provides derived types that the I/O, parallelism, and operators implemented in the infrastructure work with.

THE DRIVER LAYERS



The SUBDRIVER interacts with both the INFRASTRUCTURE and CORE, and takes care of “boilerplate” code that would otherwise be duplicated in all MPAS cores.

The choice to split the superstructure into two layers was motivated by

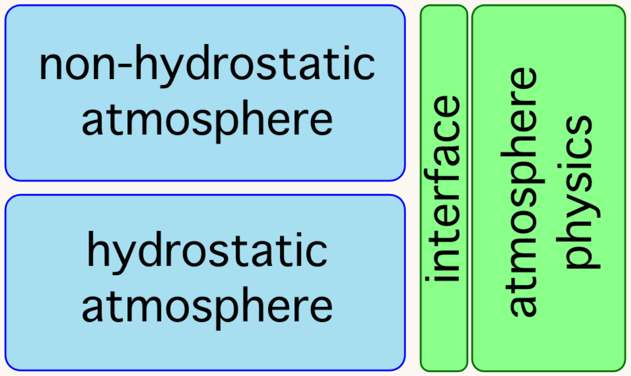
- The convenience of placing core-agnostic, driver-like code into one common place
- The potential need to replace the highest-level driver code with, e.g., an interface layer when driving MPAS from within other earth system models

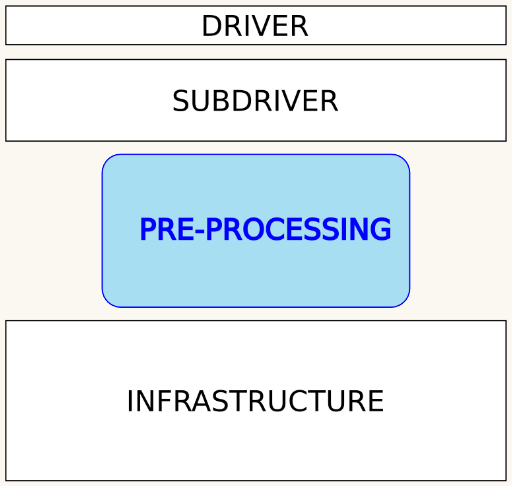
MPAS MODEL CORE

The particular MPAS core to be used is a compile-time decision

For MPAS models, the core is a combination of dynamics and physics

- The core's *run()* routine contains time loop, since time integration is core-specific
- Both atmospheric dynamical cores share the same physics routines
- **The core is passed higher-level DDTs, but use arrays directly from *field* DDTs provided by infrastructure**



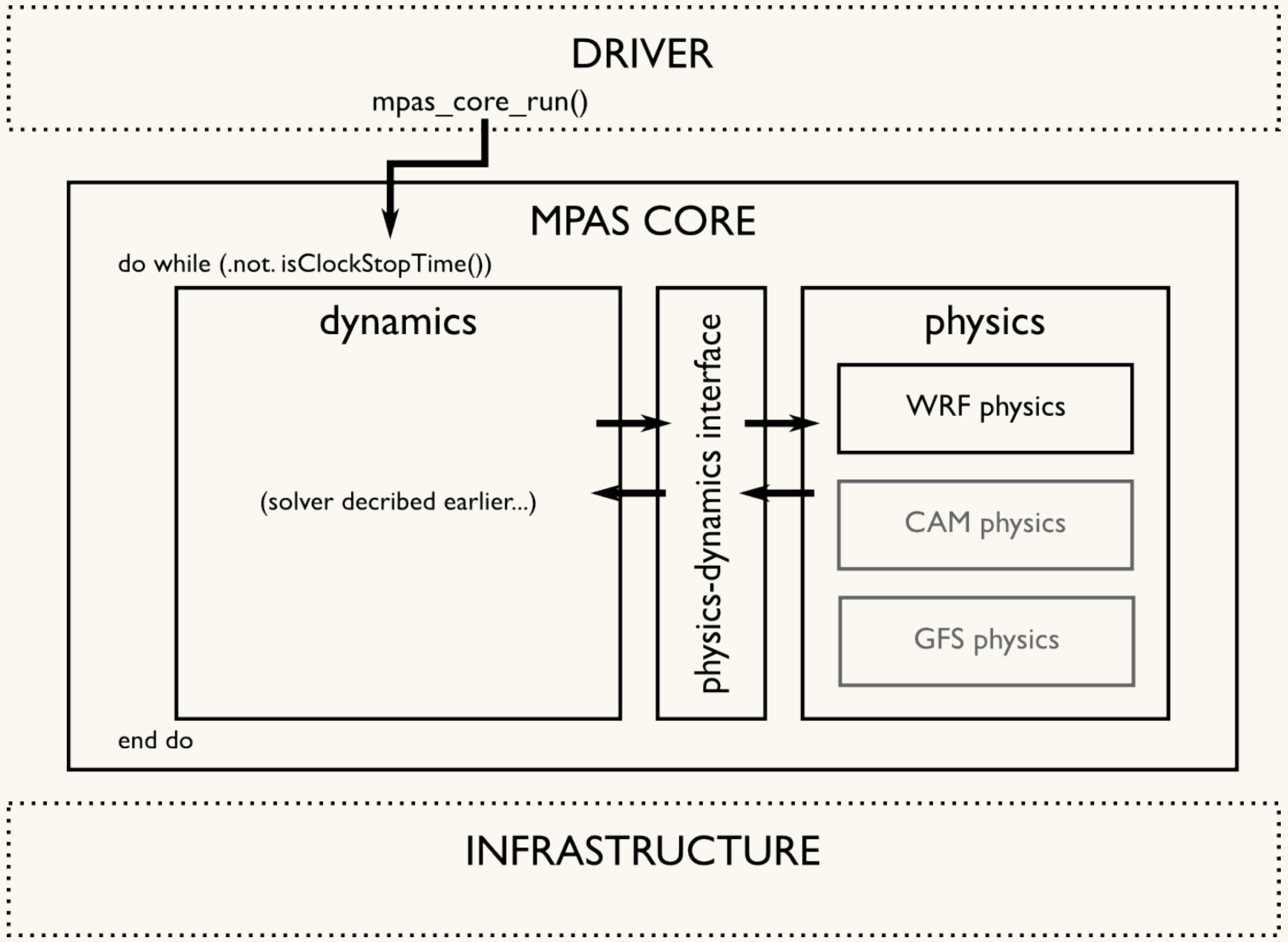


Applying a broader interpretation of *core*, we envision building pre- and post-processing, or MPAS analysis software, within the MPAS framework as well

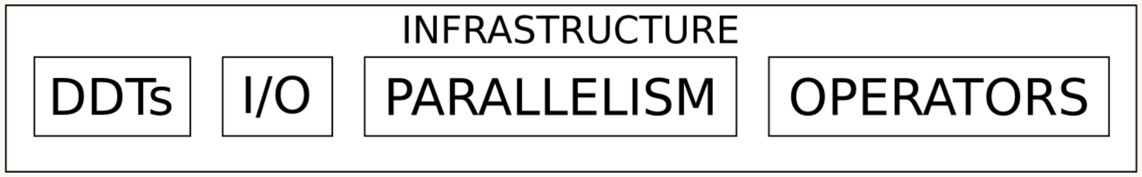
- Actually, a separate real-data initialization core for the non-hydrostatic model is currently under construction!

MPAS MODEL CORE

The non-hydrostatic atmosphere core is as described in Bill's slides:

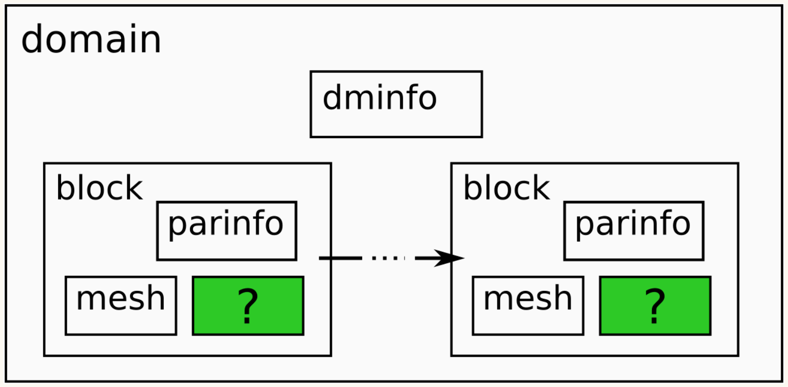


MODEL INFRASTRUCTURE



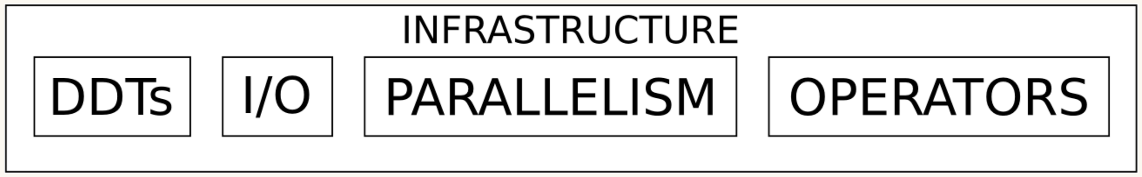
DDTs: We make heavy use of Fortran derived types in MPAS

- *domain* encapsulates complete state of computational domain for a process
- *block* contains model fields, mesh description, and parallel information for a single piece of the computational domain
- *field* stores single field's data and metadata on a block
  - *fields* are packaged in container types (green box) for convenience within a core
  - MPAS model core ultimately uses field array component directly from *field* types
- **Packaging of fields means that infrastructure DDTs are customized for a core**



- *dminfo* contains MPI communicator and other information used by PARALLELISM
- *parinfo* contains information about which cells/edges/vertices in a mesh are in a the halo region, etc.

MODEL INFRASTRUCTURE (2)



I/O: Currently, we're just using serial netCDF for our file I/O

PARALLELISM: Implements operations on *field* types needed for parallelism

- e.g., redistribute cells/edges/vertices among tasks, halo updates, scatter/gather
- callable from either serial or parallel code (no-op for serial code)
- *ideally, for multiple blocks per process, differences between shared-memory and MPI are hidden*

OPERATORS: Provides implementations of general operations on CVT meshes

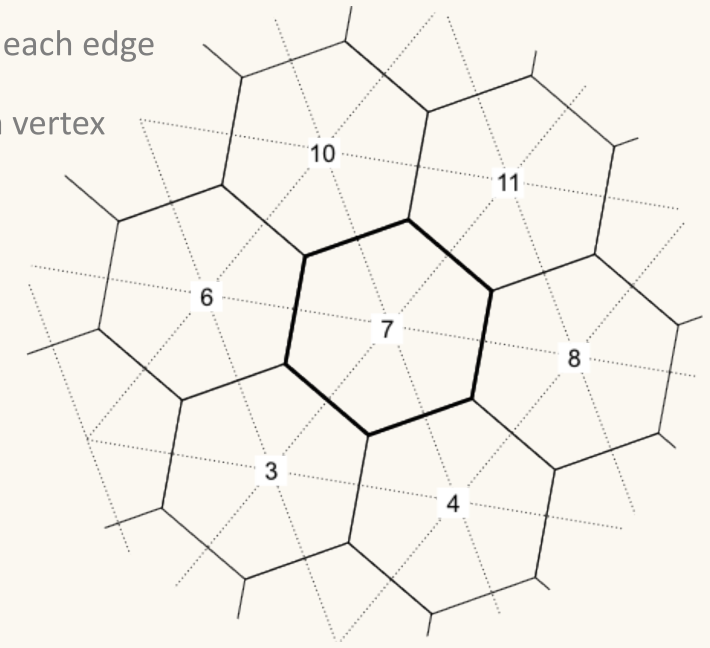
- E.g., div and curl operators, interpolation, advection, etc.
- these are "building blocks" of MPAS models

MESH REPRESENTATION

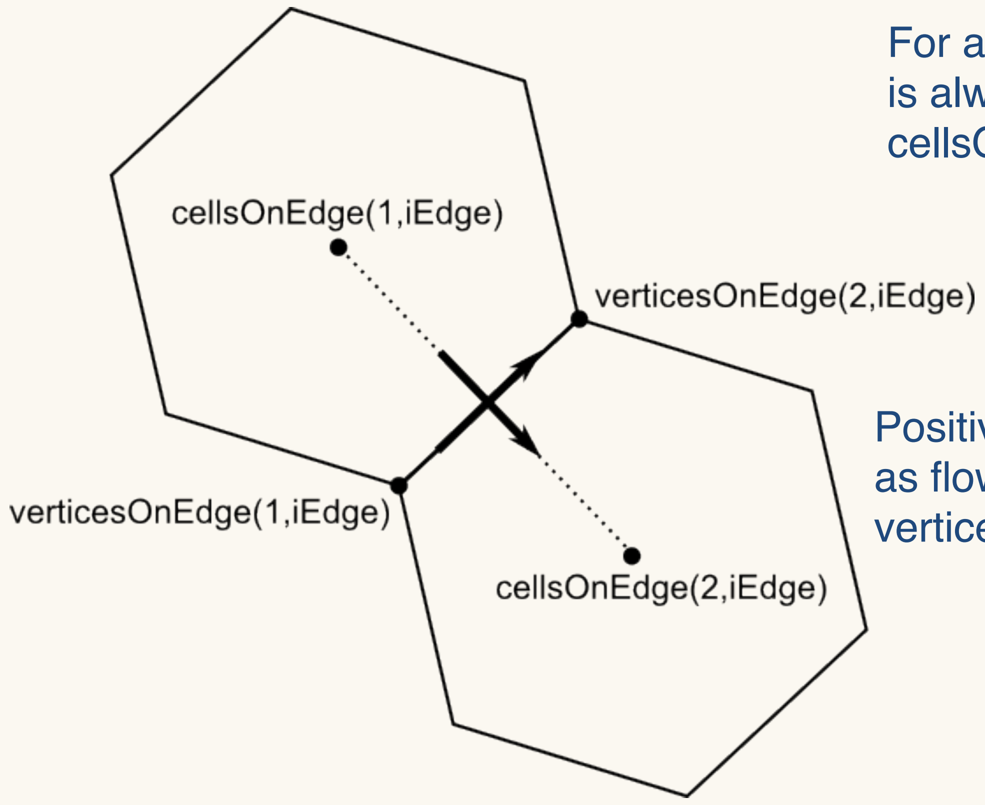
Meshes are explicitly represented in MPAS by a set of connectivity and geometry arrays:

- nEdgesOnCell(nCells) – the number of neighbors for each cell
- cellsOnCell(maxEdges, nCells) – the indices of neighboring cells for each cell
- edgesOnCell(maxEdges, nCells) – the indices of bounding edges for each cell
- verticesOnCell(maxEdges, nCells) – the indices of corner vertices for each cell
- edgesOnVertex(3,nVertices) – the indices of edges incident with each vertex
- verticesOnEdge(2,nEdges) – the indices of endpoint vertices for each edge
- cellsOnVertex(3,nVertices) – the indices of cells meeting at each vertex
- cellsOnEdge(2,nEdges) – the indices of cells split by each edge

```
nEdgesOnCell(7)=6   cellsOnCell(1,7)=8
                    cellsOnCell(2,7)=11
                    cellsOnCell(3,7)=10
                    cellsOnCell(4,7)=6
                    cellsOnCell(5,7)=3
                    cellsOnCell(6,7)=4
```



MESH REPRESENTATION(2)



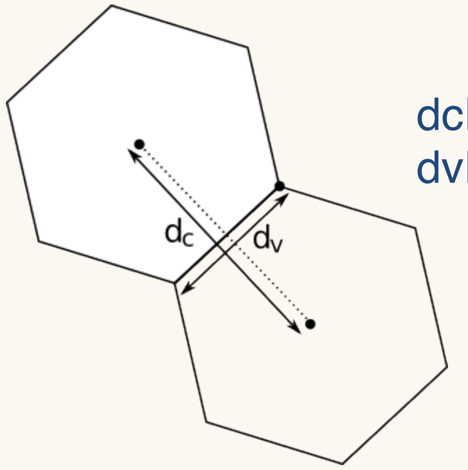
For any edge iEdge, positive u (normal) velocity is always defined as flow from cellsOnEdge(1,iEdge) to cellsOnEdge(2,iEdge)

Positive v (tangential) velocity is always defined as flow from verticesOnEdge(1,iEdge) to verticesOnEdge(2,iEdge)

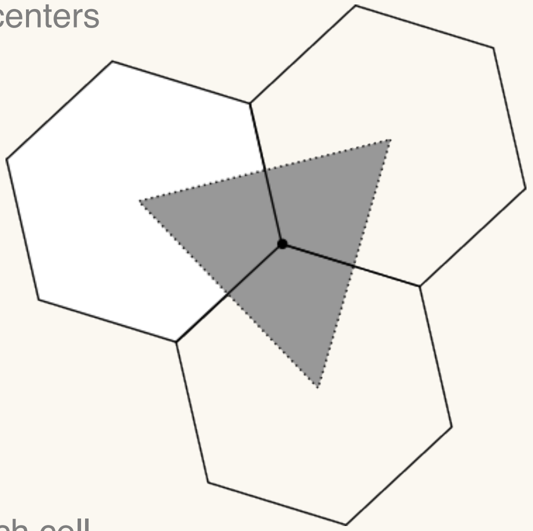
The cross product of the positive u and v vectors always points upward, out of the plane or sphere (i.e., the right-hand rule)

MESH REPRESENTATION(3)

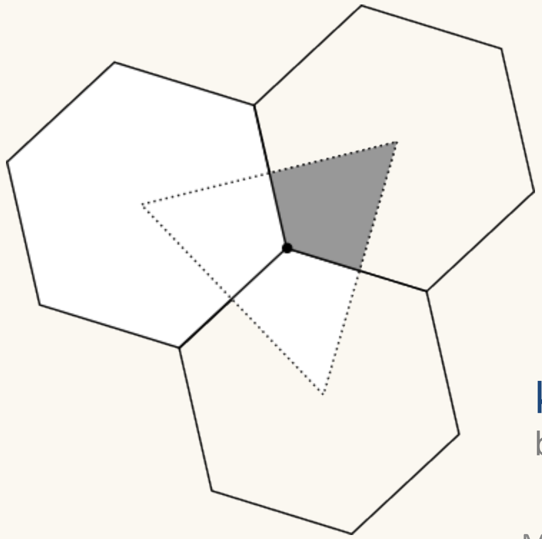
Other mesh geometry information is provided by the arrays:



dcEdge(nEdge) – distances between cell centers
dvEdge(nEdges) – length of each edge



areaCell(nCells) – area of each cell
areaTriangle(nVertices) – area of each dual-grid cell



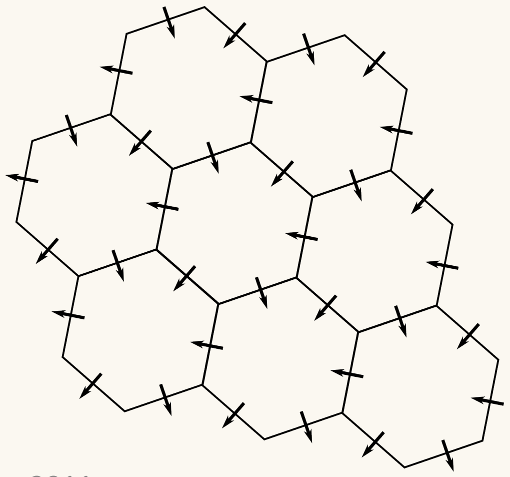
kiteAreasOnVertex(3,nVertices) – area of intersection between dual- and primal-mesh cells

EXAMPLE LOOP USING MPAS MESH DESCRIPTION

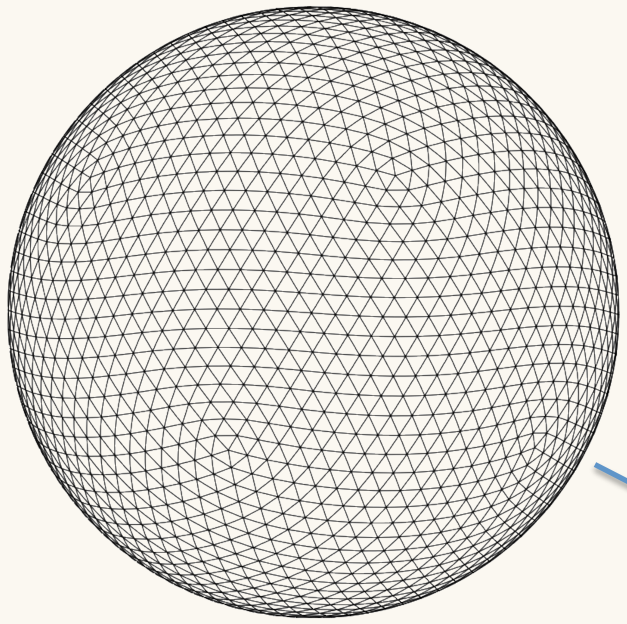
Divergence:

```
do iEdge = 1,nEdges
   do k=1,nVertLevels
      divergence(k,cellsOnEdge(1,iEdge)) += dvEdge(iEdge) * u(k,iEdge)
      divergence(k,cellsOnEdge(2,iEdge)) -= dvEdge(iEdge) * u(k,iEdge)
   end do
end do
do iCell = 1,nCells
   r = 1.0 / areaCell(iCell)
   do k = 1,nVertLevels
      divergence(k,iCell) *= r
   end do
end do
```

Edge loops can clearly be used to compute cell- or vertex-based fields, *but they do lead to reproducible sum problems (described later…)!*



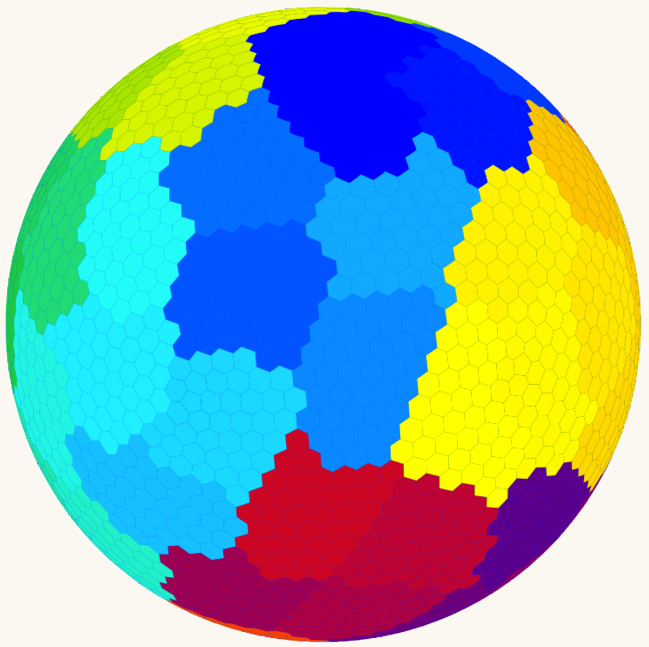
PARALLEL DECOMPOSITION



The *dual* mesh of a Voronoi tessellation is a Delaunay triangulation – essentially the connectivity graph of the cells

Parallel decomposition of an MPAS mesh then becomes a graph partitioning problem: ***equally distribute nodes among partitions (give each process equal work) while minimizing the edge cut (minimizing parallel communication)***

Graph partitioning



We use the Metis package for parallel graph decomposition

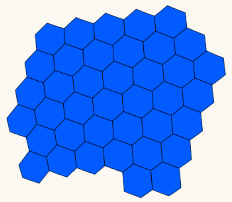
- Currently done as a pre-processing step, but could be done "on-line"

Metis also handles weighted graph partitioning

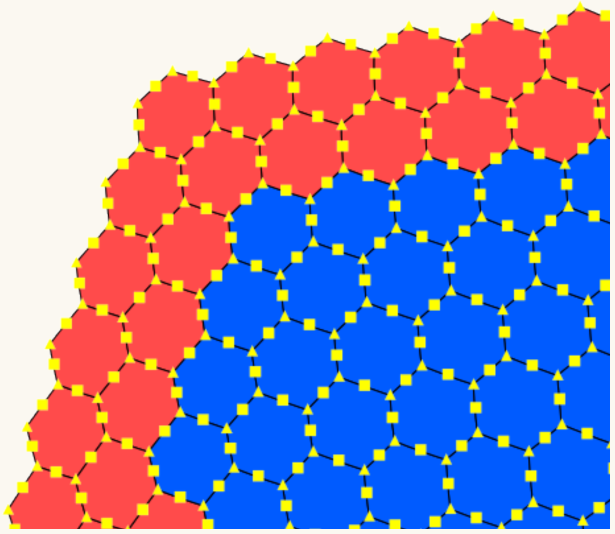
- Given *a priori* estimates for the computational costs of each grid cell, we can better balance the load among processes

PARALLEL DECOMPOSITION (2)

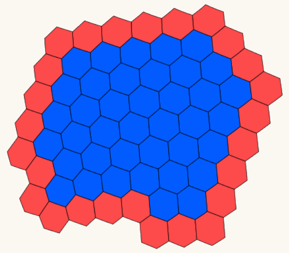
Given an assignment of cells to a process, any number of layers of halo (ghost) cells may be added



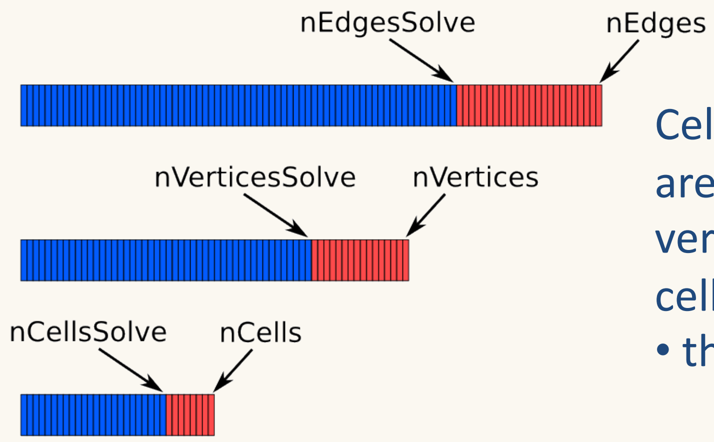
Block of cells owned by a process



With a complete list of cells stored in a block, adjacent edge and vertex locations can be found; we apply a simple rule to determine ownership of edges and vertices adjacent to real cells in different blocks

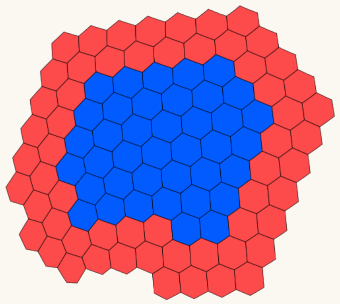


Block plus one layer of halo/ghost cells



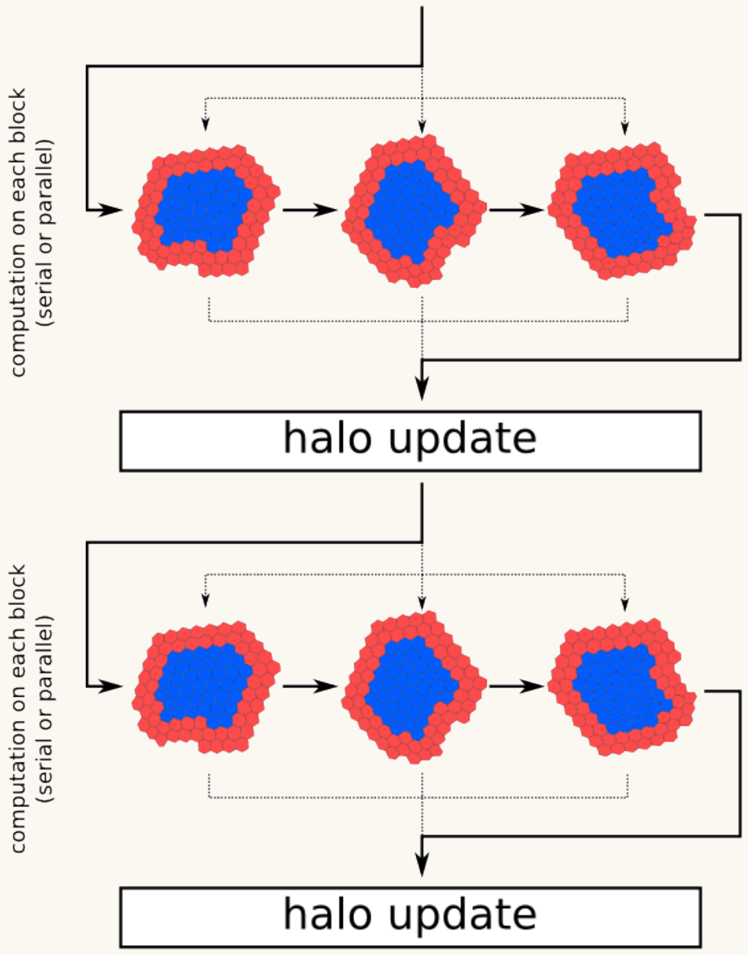
Cell-, edge-, and vertex-based fields are stored in a 1d array (2d with vertical dimension, etc.), with halo cells at the end of the array

- theta(nVertLevels,nCells)



Block plus two layers of halo/ghost cells

USE OF BLOCKS IN THE CORE


computation on each block (serial or parallel)
halo update
computation on each block (serial or parallel)
halo update

Currently, the infrastructure supports just one block per MPI task

Ideally, though, we'd like the flexibility to assign one or more blocks to each task

- *Assign each block to a GPU/coprocessor*
- *Shared-memory parallelism*
- *Opportunities for load balancing*

Halo updates only handle one field at a time

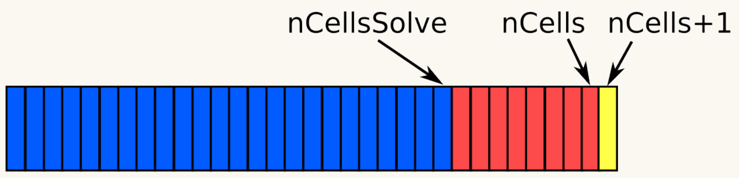
- Probably benefits to aggregating communication for fields with same stencil
- We use non-blocking sends and recvs to avoid difficulty of working out optimal deadlock-free comm patterns
- Should be possible to avoid the use of a separate read buffer

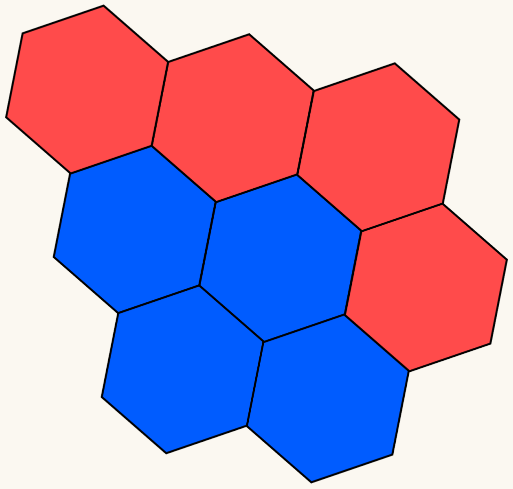
AVOIDING COMPLICATIONS IN LOOPS

Divergence:

```
do iEdge = 1,nEdges
   do k=1,nVertLevels
      divergence(k,cellsOnEdge(1,iEdge)) += dvEdge(iEdge) * u(k,iEdge)
      divergence(k,cellsOnEdge(2,iEdge)) -= dvEdge(iEdge) * u(k,iEdge)
   end do
end do
do iCell = 1,nCells
   r = 1.0 / areaCell(iCell)
   do k = 1,nVertLevels
   divergence(k,iCell) *= r
end do
```

We avoid an if-test in edge-based loops by allocating an extra "garbage cell" in field arrays and ensuring that cellsOnEdge(j,iEdge) = nCells+1 if the cell on side j of iEdge doesn't exist in a block.





CELL REORDERING

- Experiments applying a Hilbert space-filling curve ordering in MPAS shallow water core (single layer)
  - Reorder cells (ZC); reorder cells and edges (ZCE); reorder cells, edges, and vertices (ZCEV)
- Table shows % improvement in runtime over original ordering (positive numbers designate improvement)
  - 4 different problems/grid sizes
  - 16 processor runs

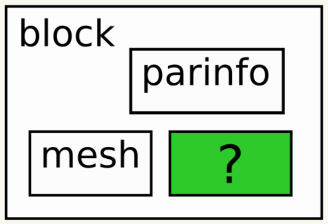
Grid size	ZC	ZCE	ZCEV
40962	-0.1%	-1.0%	1.1%
163842	13.4%	14.8%	15.6%
655362	19.0%	19.4%	17.7%
2621442	23.0%	24.9%	20.6%

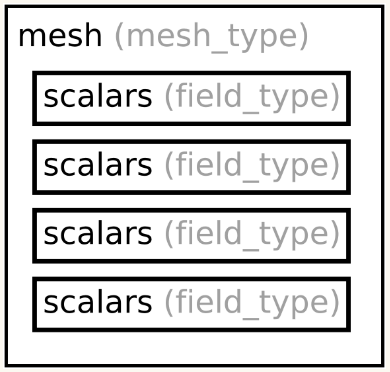
From Michael Wolf and Karen Devine (Sandia)

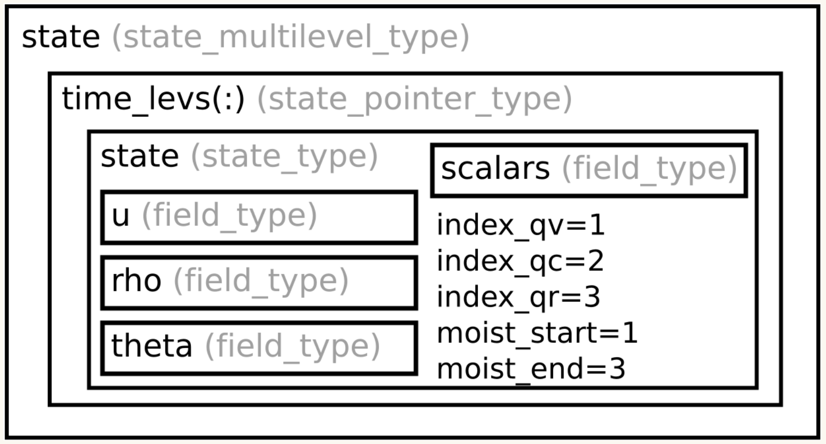
FIELD ORGANIZATION

Fortran derived types are used extensively throughout MPAS core and infrastructure

- Grouping of fields is specified in a *Registry* file

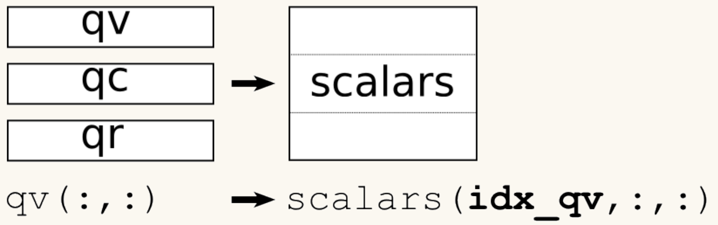






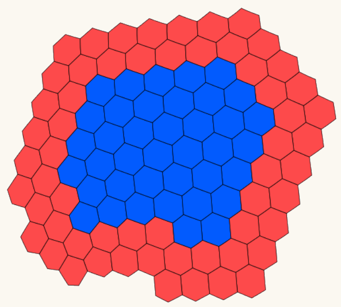
- Higher-level types like *state* are passed between subroutines
- Subroutines typically dereference field arrays once at the beginning:

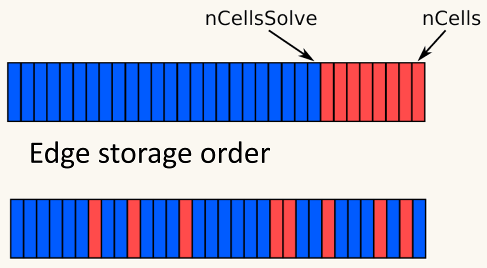
```
real, dimension(:,:), pointer :: rho
rho => state % rho % array
```



MPAS SOFTWARE STATUS AND ISSUES

- Bit-for-bit restartability (with physics in stand-alone MPAS-A)
  - Was simple to achieve
- Bit-identical results on different task counts (currently dynamics only in MPAS-A)
  - Required dealing with order-of-summation issues
  - Can be easily dealt with in ocean and hydrostatic cores, too
  - Changing to highlighted code gives ~11% performance hit on bluefire for the example loop shown below

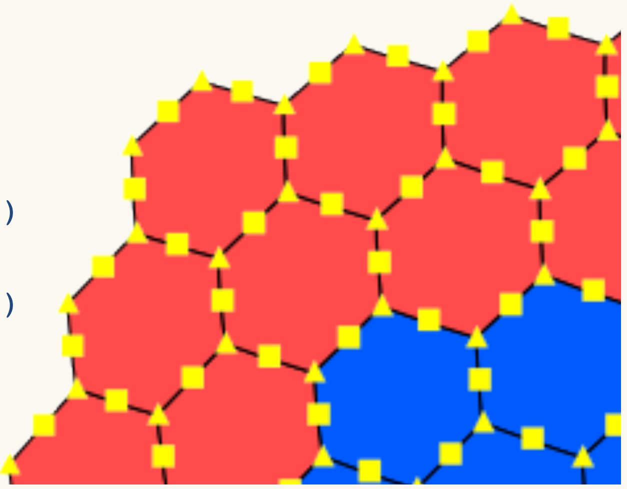




Edge storage order

Reproducible sum edge order

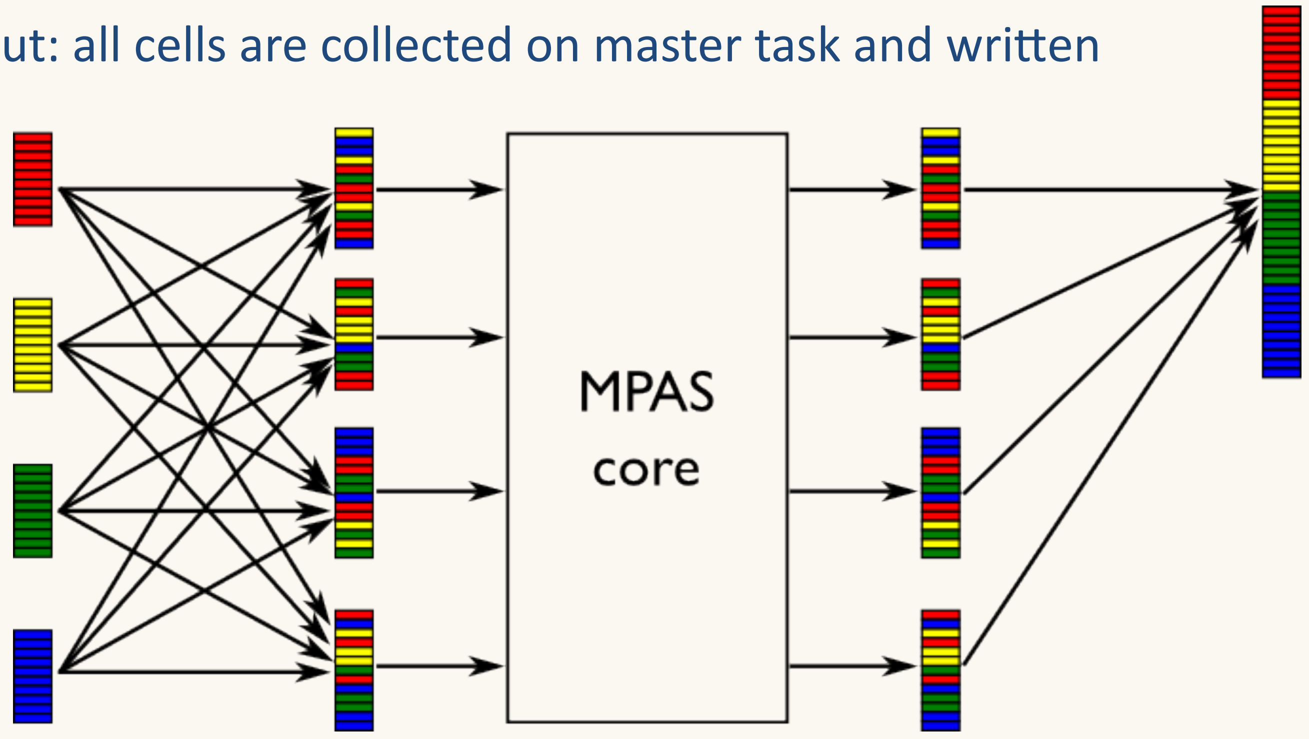
```
do iEdge = 1,grid % nEdges
do iEdge1 = 1,grid % nEdges
   iEdge = grid % edgePermutation % array(iEdge1)
   do k=1,nVertLevels
      circulation(k,verticesOnEdge(1,iEdge)) -= dcEdge(iEdge)
                                                * u(k,iEdge)
      circulation(k,verticesOnEdge(2,iEdge)) += dcEdge(iEdge)
                                                * u(k,iEdge)

   end do
end do
```



MPAS SOFTWARE STATUS AND ISSUES

Current I/O subsystem uses serial netCDF API

- Input: each task reads a contiguous range of nCells/mpi_size cells, cells are then redistributed to blocks (incl. halo cells)
- Output: all cells are collected on master task and written



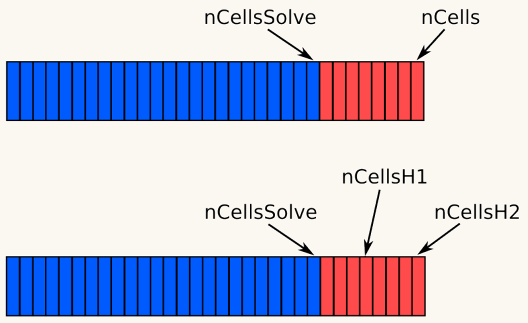
PIO looks like an attractive option for use in MPAS

MPAS SOFTWARE STATUS AND ISSUES

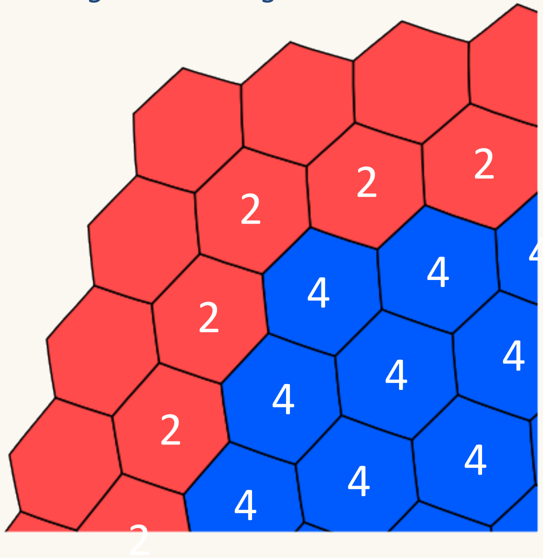
Misc. parallel performance issues

- Lack of provisions to update partial halos
  - Currently “all-or-nothing”
- Inability to loop over subsets of halo cells/edges/vertices
  - This leads to redundant calculation

Currently, we only distinguish between real and halo cells:



Below: derivative orders for a scalar field computable with two layers of ghost cells and no comm.; current loops for del2 cover all edges; loops for del4 require if-tests and only edges bordering owned cells.



Examples:

- Scalar transport
- Del4 hyper-diffusion

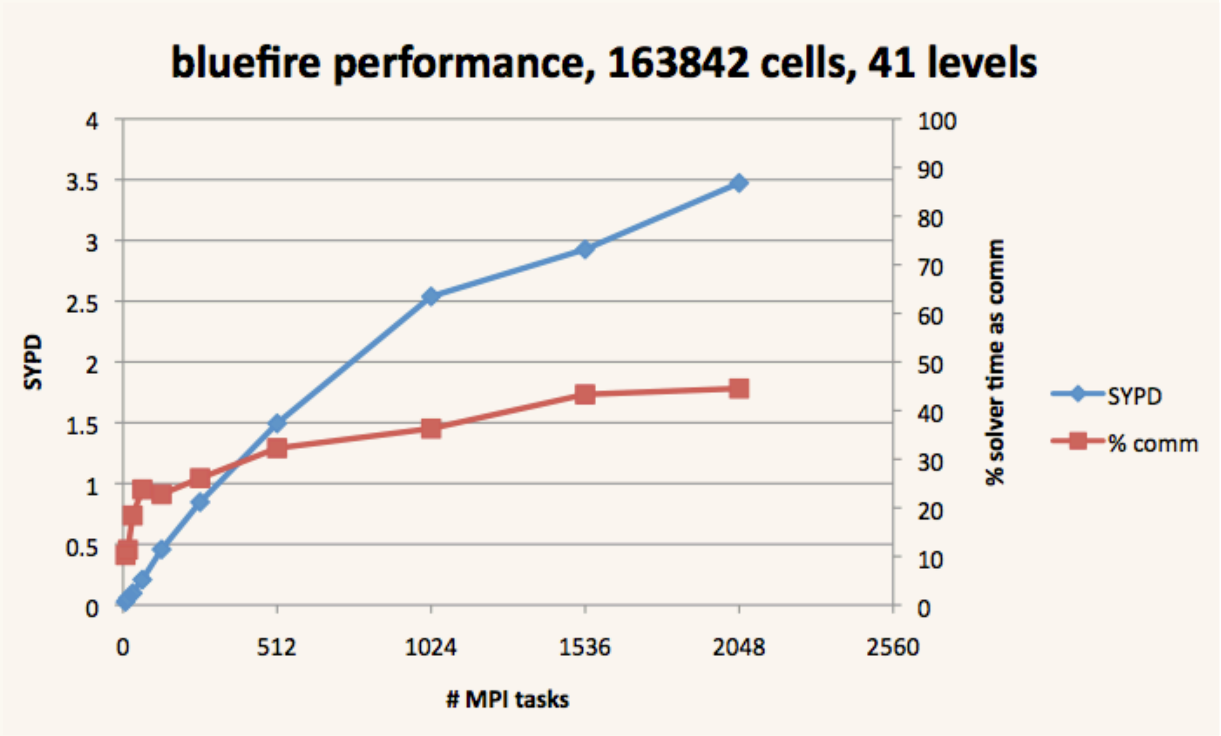
Edge index ranges to loop over include:

- Edges bordering owned cells
- Edges bordering first layer of ghost cells
- All edges

MPAS SOFTWARE STATUS AND ISSUES

NB: We've placed no emphasis on writing fast code so far! Correctness and rapid development have been foremost.

Right: Initial performance for MPAS non-hydrostatic atmosphere core on bluefire with a ~60-km global mesh; times for dynamical solver only; SYPD assumes a 300 s time step.



- Experience indicates that atmospheric solvers are about 2-3x slower than, e.g., CAM FV core
- What is considered good scaling on bluefire? Access to other hardware may be helpful
- My opinion: we have quite a few opportunities for improving performance and scalability in MPAS