High Performance Asynchronous I/O for Exascale Spectral Element Methods

F.D. Witherden

Department of Ocean Engineering, Texas A&M University

Motivation

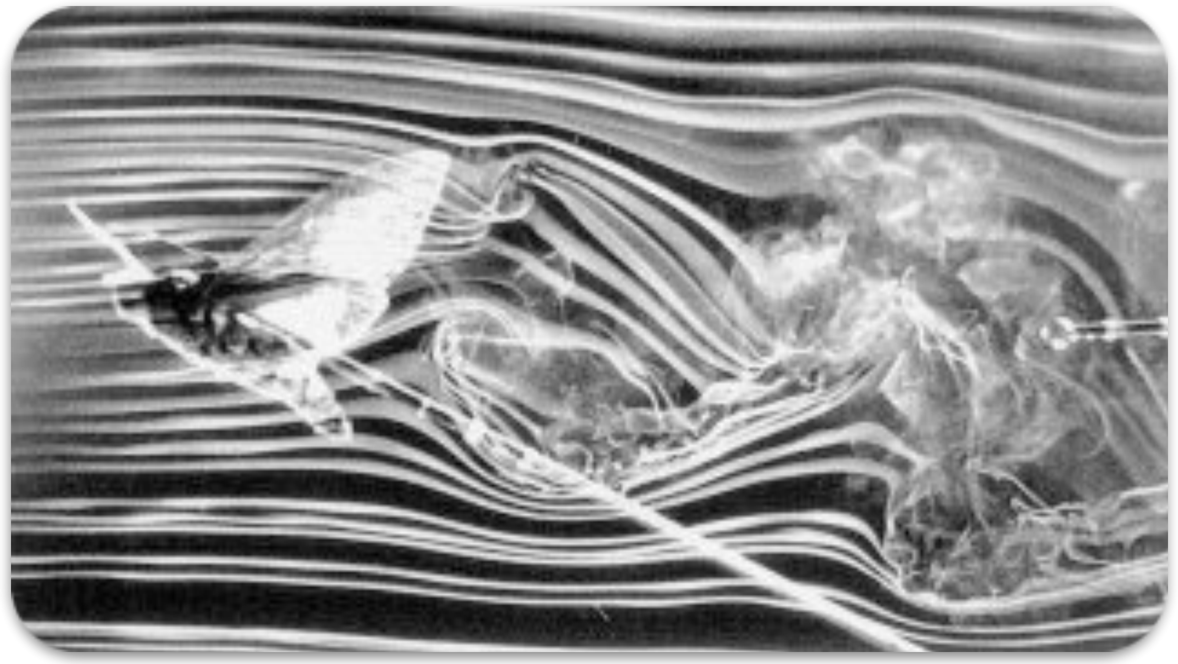
PyFR

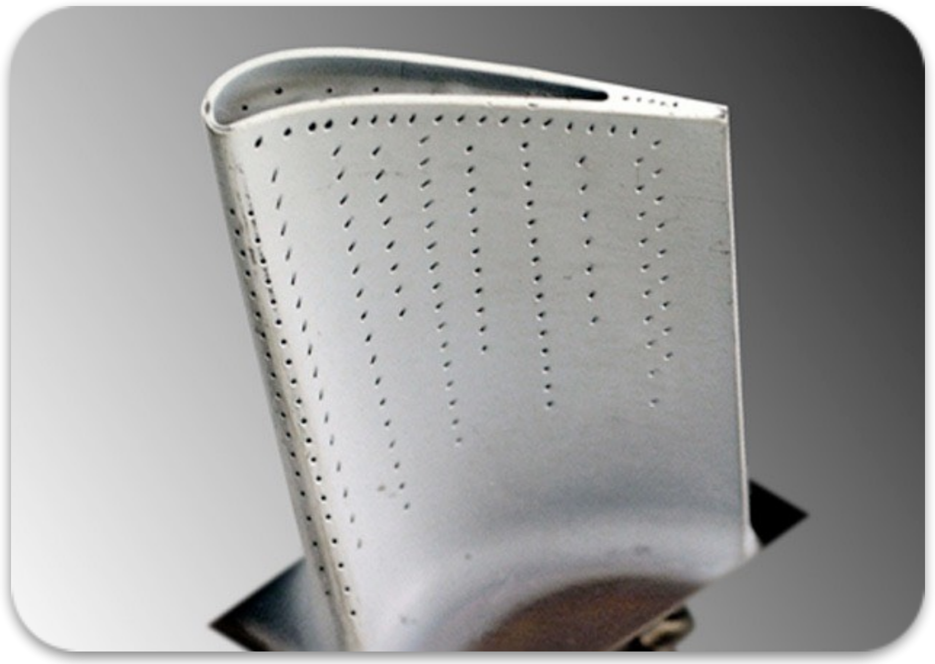
Requirements

Approach

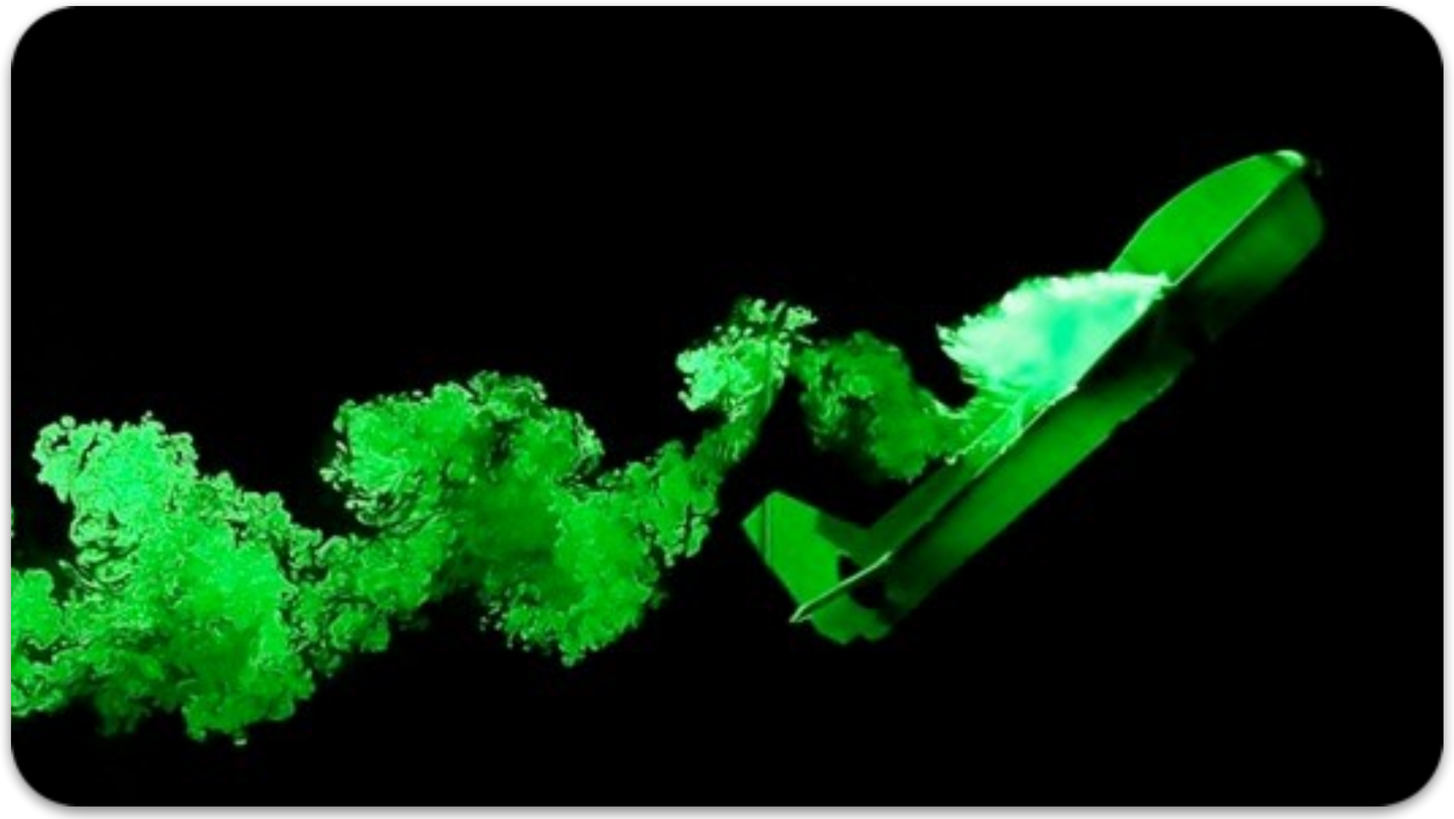
Conclusions

Motivation





- Interested in simulating unsteady, turbulent, flows.



Motivation

- Such large eddy simulations (LES) often require billions of degrees of freedom (DOFs).
- We routinely run unsteady turbulent simulations with **10 billion DOFs**.

Motivation

- For example, an MTU T161 LPT case had 90.68 million elements and was run with degree four polynomials.
- This equates to **11.35 billion DOFs** per equation.
- Each checkpoint file is hence **423 GiB**.

Motivation

- Even worse are time-averaged statistics.
- For this case we averaged **206 quantities**.
- Each time-average file is hence **17,420 GiB!**

2.6.3. Time-Averaging

During the Data Extraction Period various quantities were time-averaged throughout the entire domain, including:

1. $\overline{v_i}$ (3)
2. $\overline{\rho}$ (1)
3. $\overline{\rho v_i}$ (3)
4. $\overline{\rho v_i v_j}$ (6)
5. $\overline{\rho v_i v_j v_k}$ (10)
6. $\overline{p}$ (1)
7. $\overline{p v_i}$ (3)
8. $\overline{p v_i v_j}$ (6)
9. $\overline{p v_i v_j v_k}$ (10)
10. $\overline{p^2}$ (1)
11. $\overline{p^2 v_i}$ (3)
12. $\overline{p^2 v_i v_j}$ (6)
13. $\overline{p^3}$ (1)
14. $\overline{p^3 v_i}$ (3)
15. $\overline{p^4}$ (1)
16. $\overline{v_i v_j v_k v_l}$ (15)
17. $\overline{\sqrt{v_i v_i}}$ (1)
18. $\overline{\sqrt{\gamma p / \rho}}$ (1)
19. $\overline{\sqrt{\frac{v_i v_i}{\gamma p / \rho}}}$ (1)
20. $\overline{p\left(1 + \frac{\gamma - 1}{2} \frac{\rho v_i v_i}{\gamma p}\right)^{\frac{\gamma}{\gamma - 1}}}$ (1)

Motivation PyFR Requirements Approach Conclusions

PyFR

Python + Flux Reconstruction

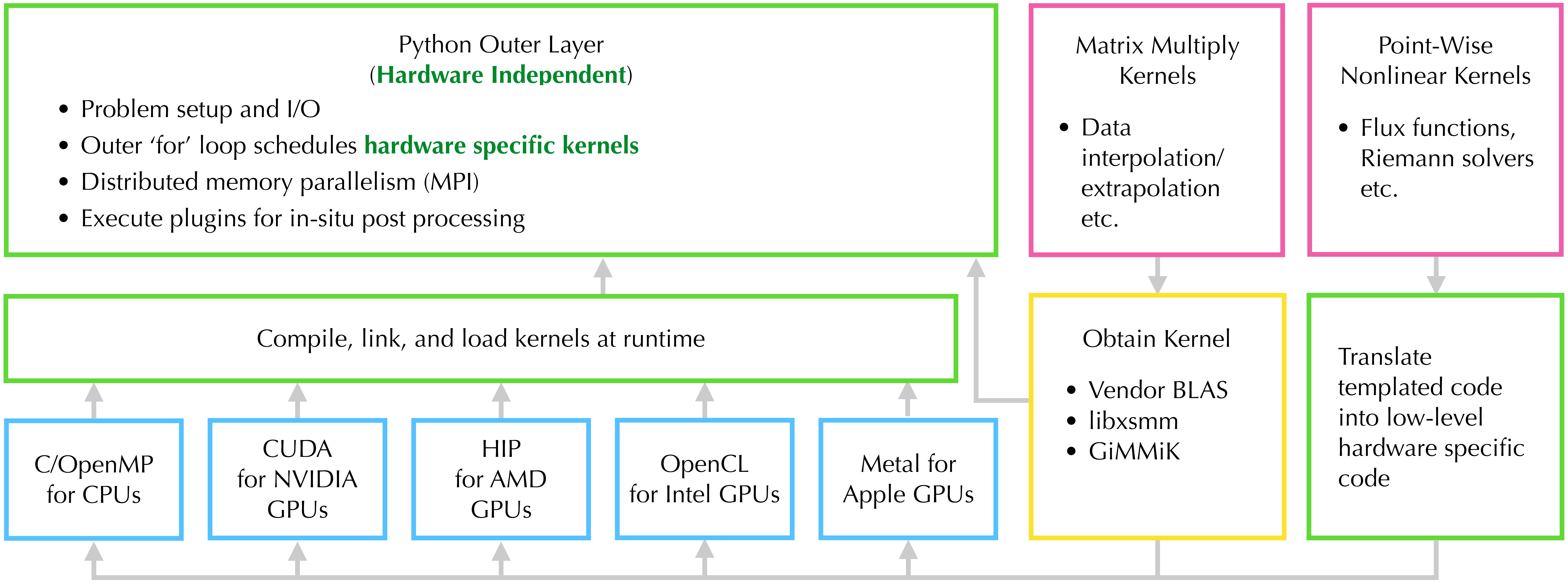
PyFR

- Features.

Governing Equations	Compressible and incompressible Euler and Navier Stokes
Spatial Discretisation	Arbitrary order Flux Reconstruction on **curved mixed unstructured grids** (tris, quads, hexes, prisms, pyramids, tets)
Temporal Discretisation	Explicit adaptive Runge-Kutta schemes and implicit SDIRK
Stabilisation	**Anti-aliasing**, artificial viscosity, modal filtering, entropy filtering
Shock capturing	Artificial viscosity and **entropy filtering**
Platforms	x86 and ARM CPU clusters AMD, Apple, Intel, and NVIDIA GPU clusters
Plugins	NaN checker, file writer, progress bar, fluid force, **turbulence generation**, Ffowcs–Williams Hawkings, **in-situ vis**, time averaging, point sampler, expression integrator

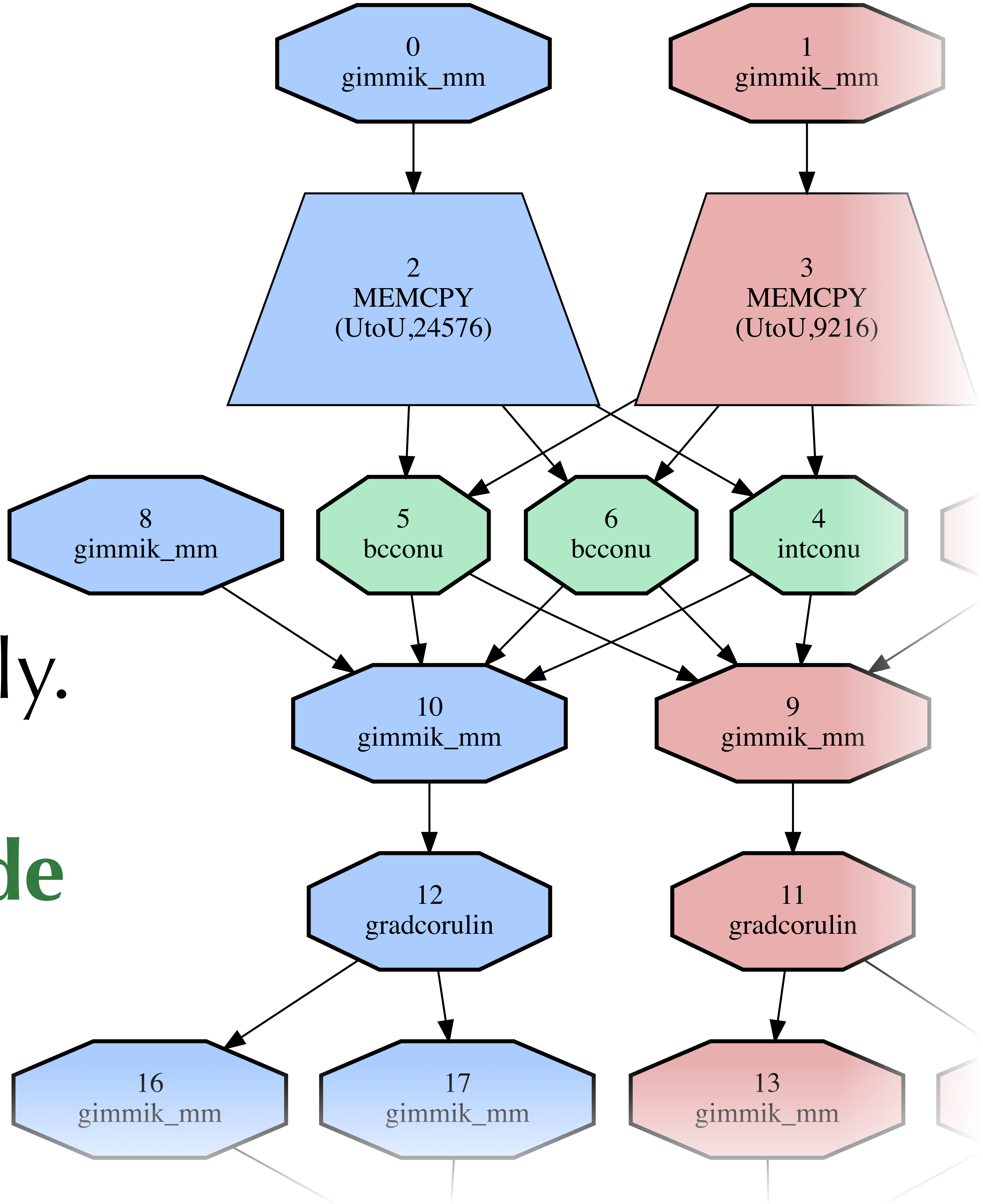
PyFR

- High level structure.



PyFR

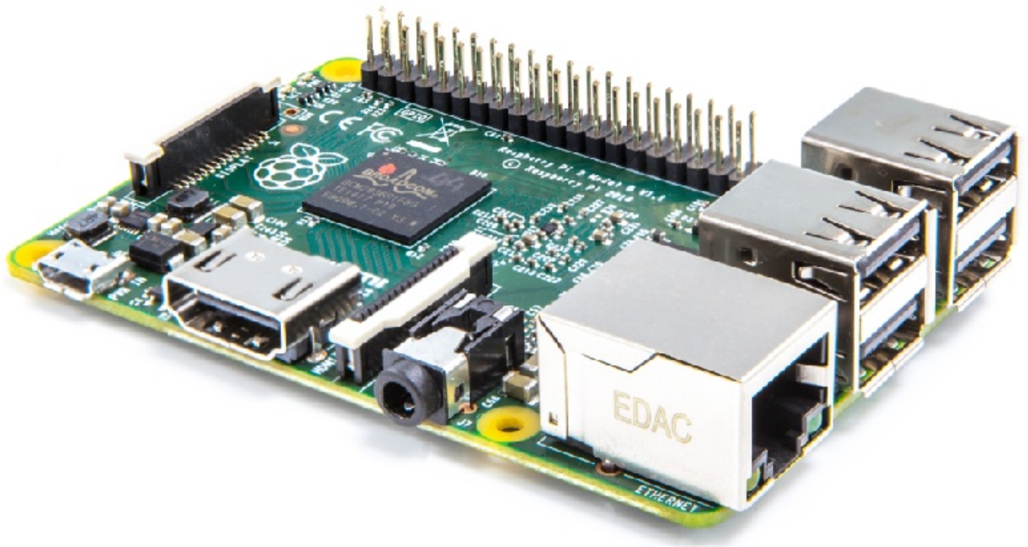
- Employ a **state-of-the-art implementation**.
- Use a **task-graph structure** internally.
- Make extensive use of **run-time code generation with auto-tuning.**



PyFR

- Enables **heterogeneous computing** from a homogeneous code base.

PyFR



Performance

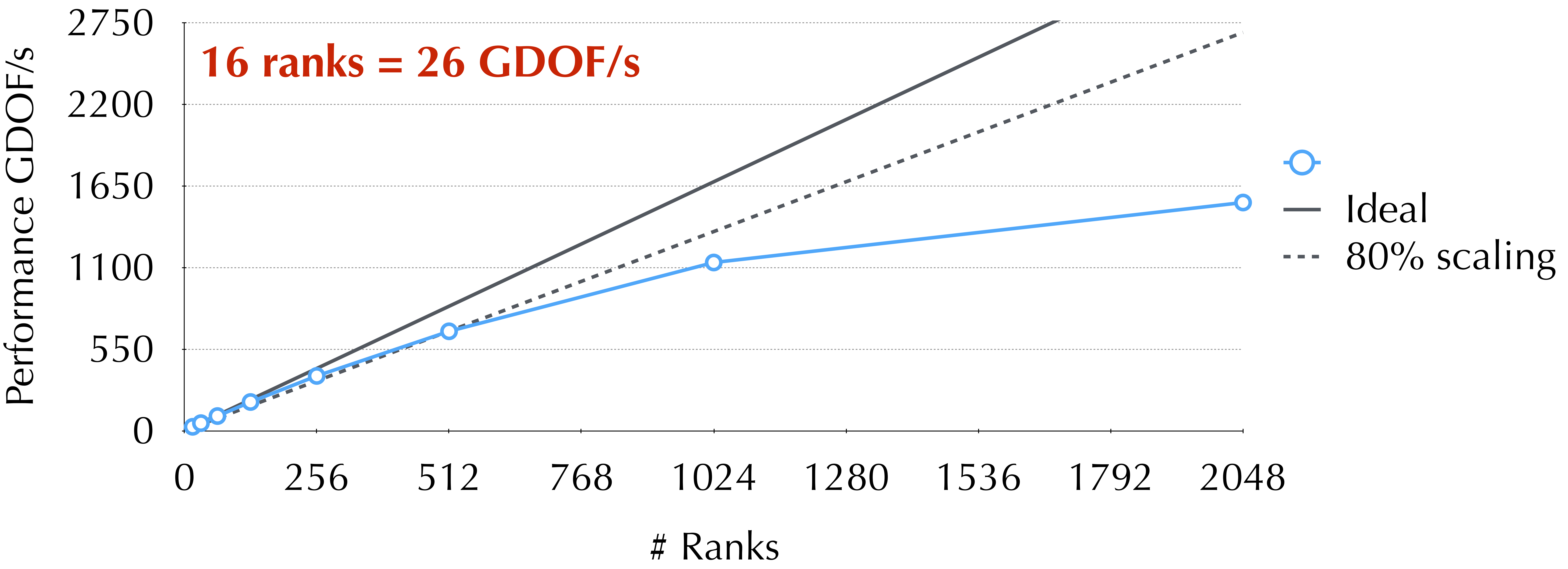
- Consider a tetrahedral grid with **degree seven solution polynomials** and explicit Navier–Stokes.
- Employ a mesh with 12×105^3 elements.
- Use **Frontier** (AMD MI250X) and **Alps** (NVIDIA GH200).

Performance

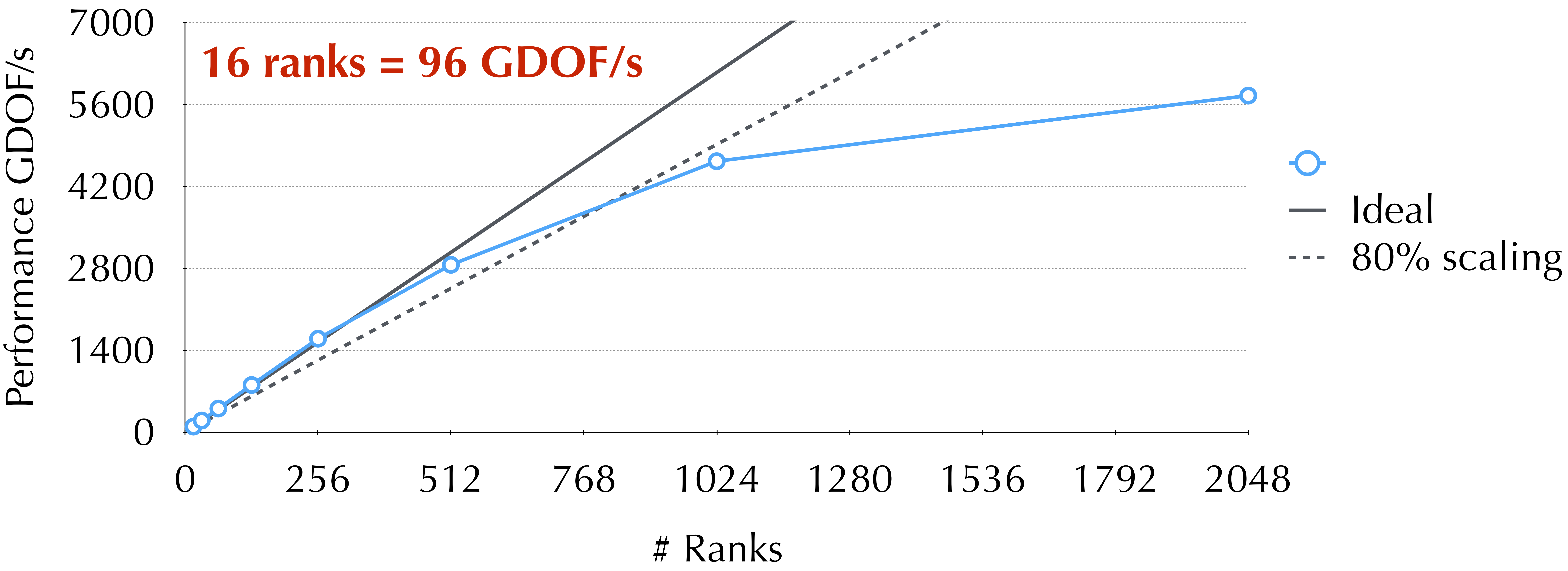
- Measure performance in **GDOF/s**.
- Translation: if we have **10^9 DOF** in our simulation and need **100,000 RK4** time steps to collect good statistics then our runtime will be:

$$(10^9 \times 10^5 \times 4 \,/\, \mathbf{X}) \text{ seconds.}$$

Performance on Frontier



Performance on Alps



Disk I/O

- Legacy format based around **parallel HDF5**.
- Separate mesh and solution files (good).
- Internal structure of mesh and solution files are **decomposition dependent** (bad).

Motivation PyFR **Requirements** Approach Conclusions

Requirements

- Our design specification for the new format was based around **five key requirements**.

Requirements: Archival

1. The format must be built upon a well established and self-documenting **archival-grade container**.

   - Aids in portability and ensures that files will in principle remain **readable for decades**.

Requirements: Compact

2. Exhibit **minimal redundancy** except for when it greatly improves usability or reduces processing time.

- Reduces storage and bandwidth requirements.

Requirements: Scalable

3. Be compatible with the **‘quirks’ of parallel file systems**.
    - Needed for scalability on leadership class machines.

Requirements: Robust

4. Capable of delivering good application performance **even in sub-optimal environments**.

    - User **misconfiguration is common** and storage is a shared resource that is subject to frequent abuse by machine learning jobs.

Requirements: Simple

5. Be simple enough to enable **ad-hoc post processing** without bespoke middleware.

    - Each simulation has its own post-processing requirements and engineers all have their favourite languages (Python, Julia, R, MATLAB, …).

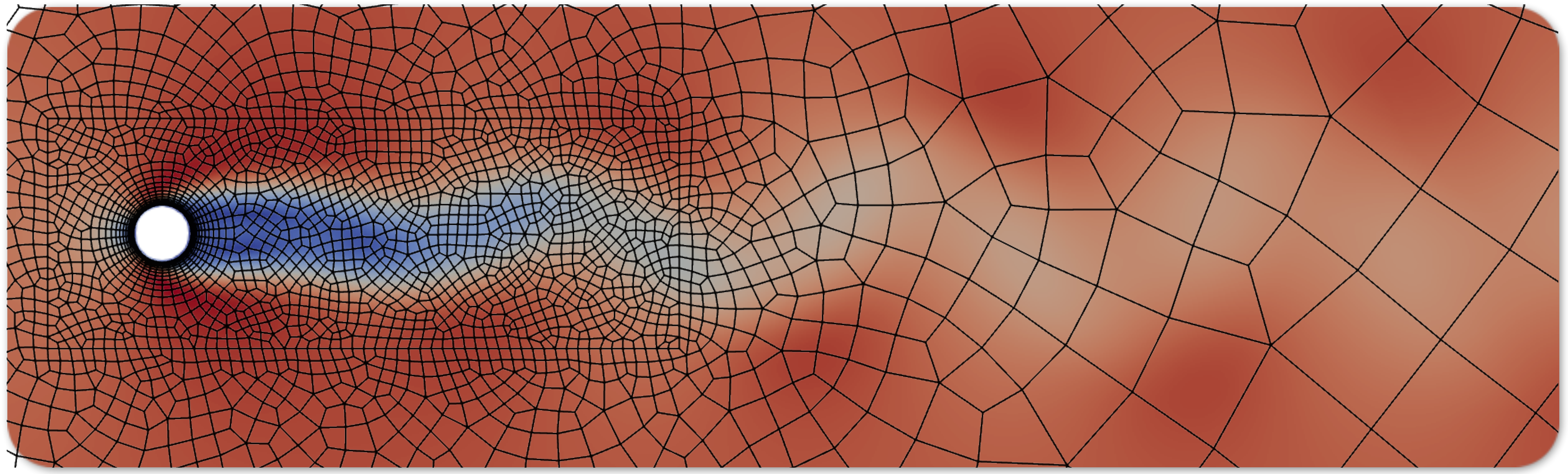
Motivation PyFR Requirements **Approach** Conclusions

Approach

- The most natural means of satisfying requirements 1 (archival) and 5 (simple) is with **HDF5**.
- While there are trendier formats (e.g. BP5) they are **not as widely deployed** and lack the proven track record of HDF5.

Approach

- Requirement 2 (compact) strongly suggests **decoupling the geometry and the solution**.



Mesh

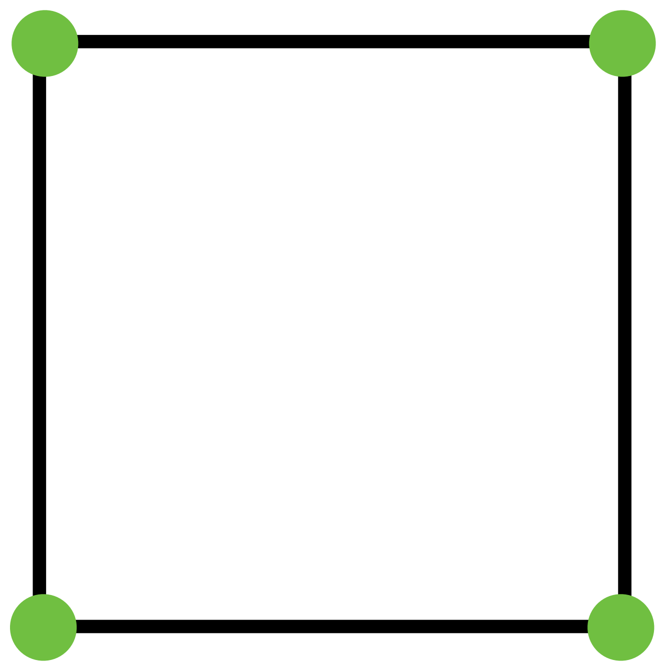
Solution

Approach

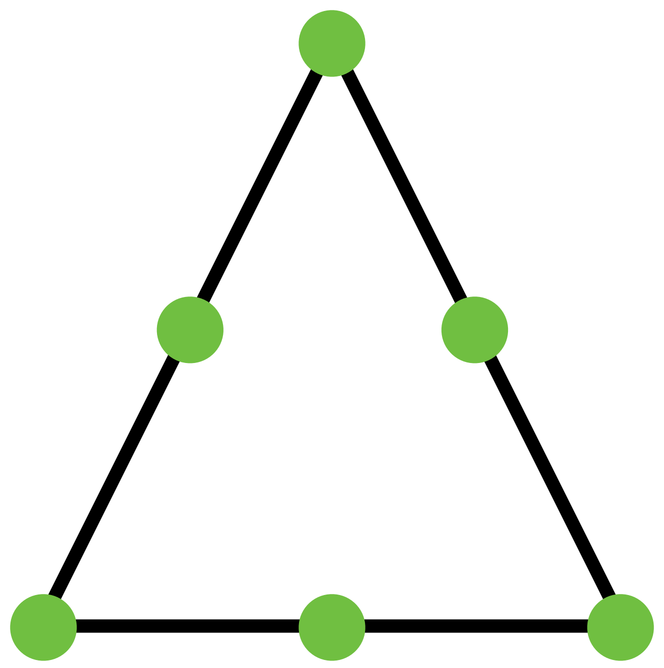
- As mesh files are **write-once read-many** and only used at start-up they are **comparatively easy to design**.
- The two major considerations are:
  - How are elements represented?
  - How is connectivity represented or reconstructed?

Approach: Mesh

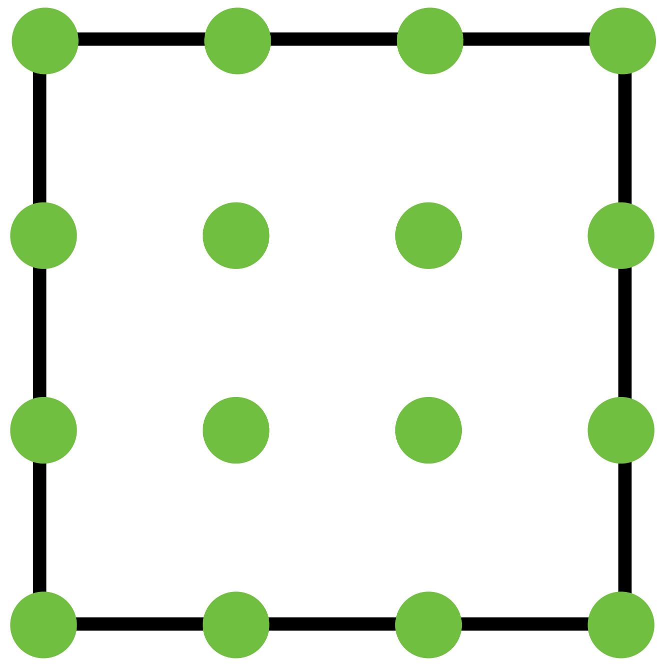
- A mesh is a collection of elements defined by nodes.



Quad
$p = 1$



Tri
$p = 2$



Quad
$p = 3$

Approach: Mesh

- Using **quadratically curved triangles** as an example the most direct representation is:

```
struct {
    double locs[6][2];
} tris[Nt];
```

- However, it is also wasteful as **many points are repeated**.

Approach: Mesh

- A more efficient approach is to represent elements in terms of **node numbers** whence:

```
double nodes[M][2];
struct {
    long locs[6];
} tris[Nt];
```

where M is the **number of distinct nodes**.

Approach: Mesh

- With this we have **one array per element type** indexing into a single global nodes array.
- Number of nodes per element is fixed by the **highest degree of curvature**.

Approach: Mesh

- While connectivity information can be derived from node numbers alone this is **extremely inefficient**.

- It is hence better to embed the information whence:

```
struct {
    long locs[6];
    struct { short cidx; long off; } conn[3];
} tris[Nt];
```

Approach: Mesh

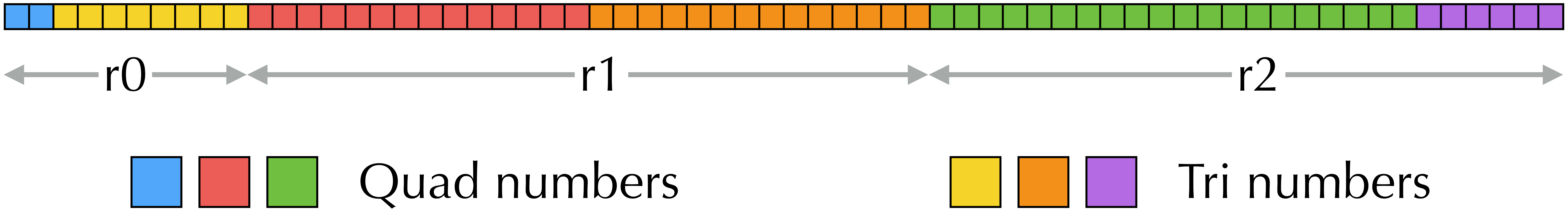
- The `cidx` is an index into a **codec array** which provides a flexible means of stating **what one is connected to**:

```
codec = ['/eles/tri/0', '/eles/tri/1',
         '/eles/tri/2', '/bc/outflow', …];
```

- For all non-boundary connections `off` is the **offset into the connected elements array**.

Approach: Mesh

- Parallel decompositions can be specified as a **single array of element numbers**.
- For example with a three partition mesh with quads and tris:

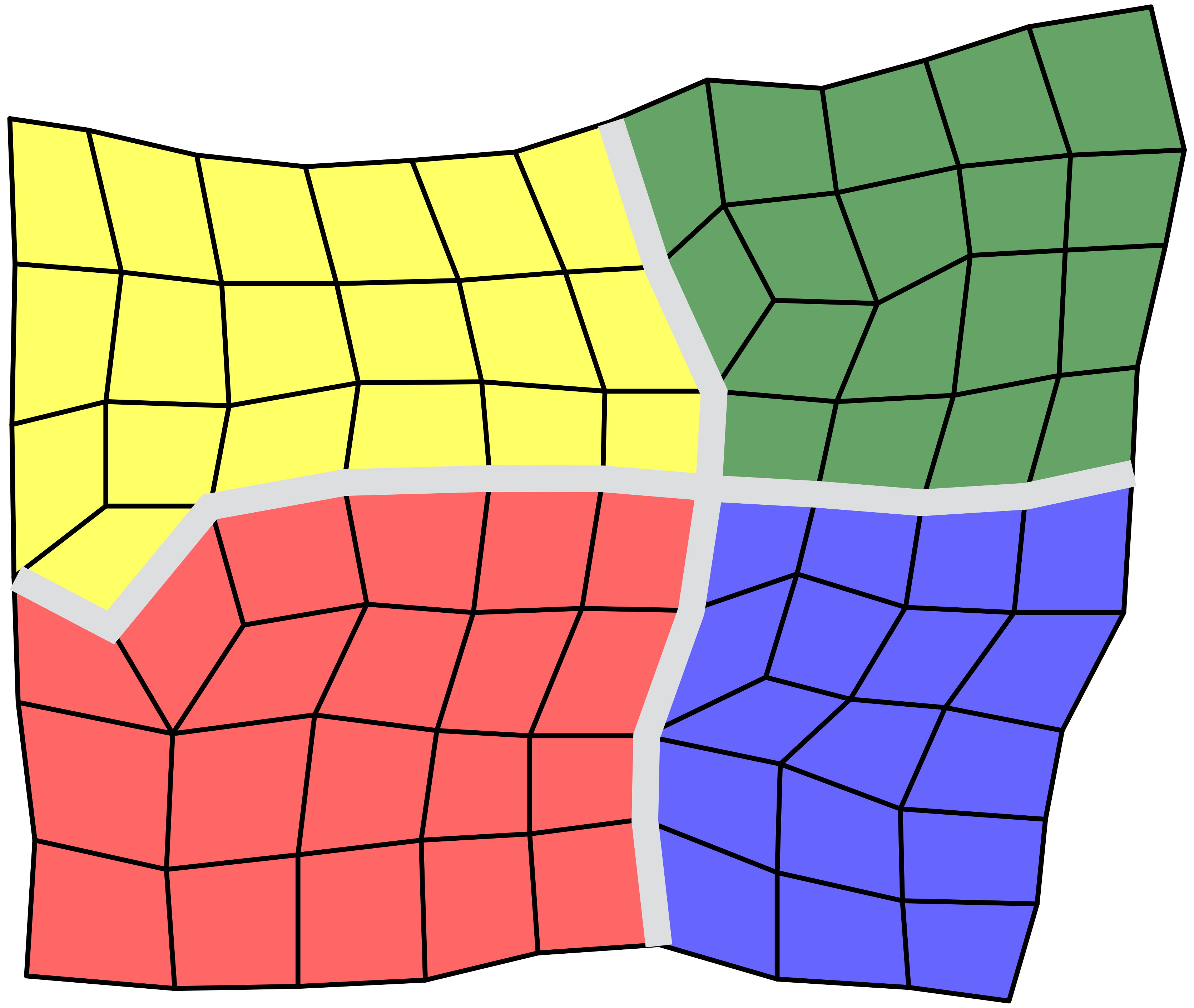


Approach: Mesh

- The elements arrays can be efficiently read by having each rank **read a contiguous chunk** of each array.
- An **Alltoallv** can then be used to redistribute the data.
- This approach can also be used for reading the nodes.

Approach: Mesh

- In parallel it is also necessary to construct **cross-partition interface connectivity** arrays.
- For each face on a partition boundary we need to determine **which rank has its neighbour**.

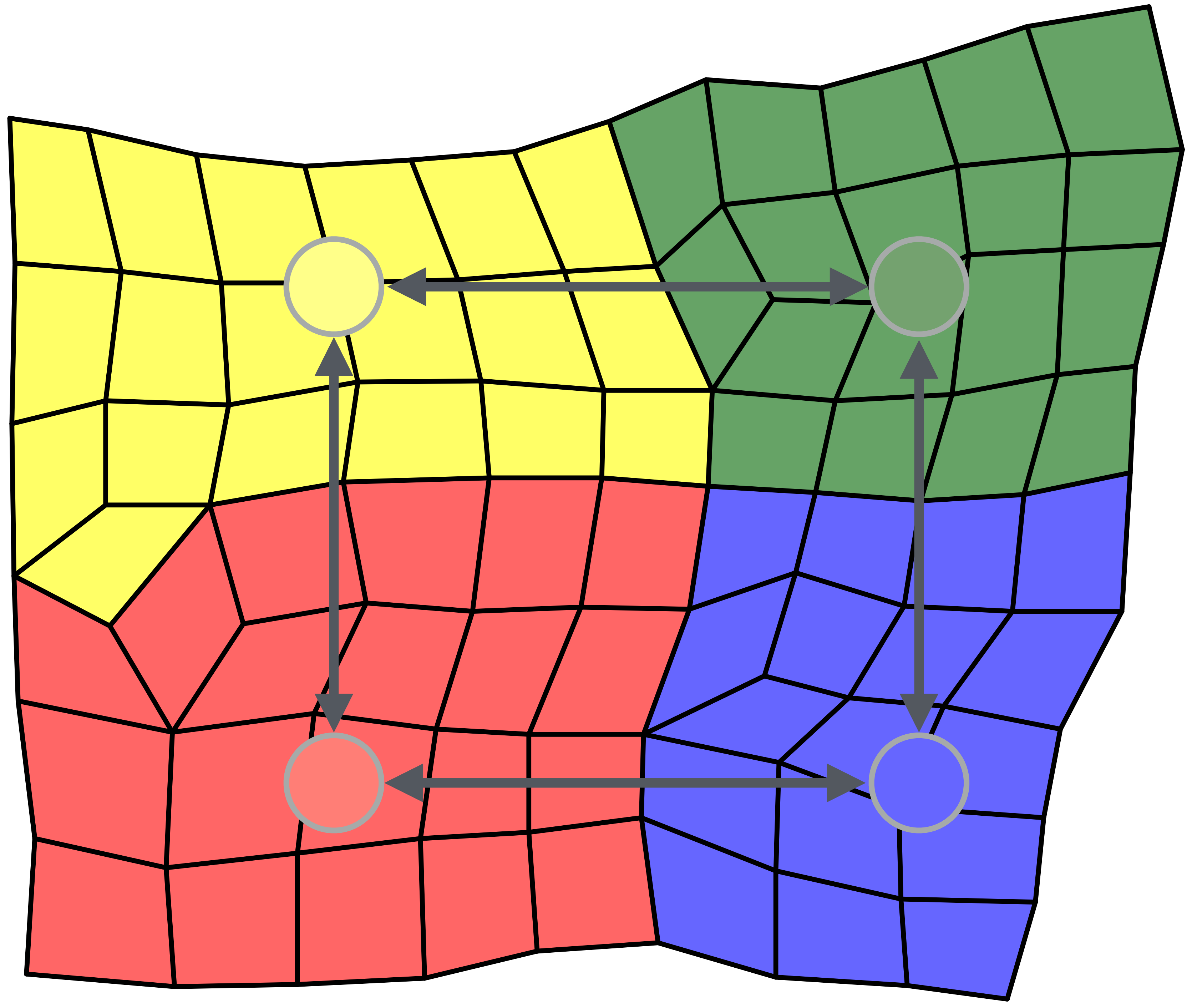


Approach: Mesh

- Simplest solution is for each rank to **maintain a list of unpaired faces**.
- This list can then be distributed with an **Allgatherv**.
- Ranks can then see which of these faces they have and respond accordingly via an **Alltoallv**.

Approach: Mesh

- This approach **does not scale**, however, and breaks down in the limit of one-element per rank.
- A neat fix for this is to augment the partitioning array with its **neighbour connectivity graph**.



Approach: Mesh

- This graph can be passed to **Neighbor_allgatherv** from MPI-3 such that only partitions we're actually connected to receive our unpaired faces.
- For a well partitioned mesh the number of neighbours is $\mathcal{O}(1)$ and thus **the approach scales**.

Approach: Solution

- With the mesh handled we now turn our attention to solution files which are **write-once read-sometimes**.

- Hence, our goal is to **maximise the speed** at which they can be written out to disk.

- This requires a short interlude on **parallel file systems**.

Approach: Lustre

- One of the more infamous parallel file systems is Lustre.
- It is **loved by users** and sysadmins alike!

Approach: Lustre

- A useful mental model is that of RAID 0.
- Instead of disks we have **object storage targets** (OST's).
- Moreover, the number of OST's and the stripe size is **configurable on a per-file basis**.

Approach: Lustre

- 6 OST's.
- 32 MiB file.
- 4 MiB stripe size.
- 3 stripes.

Approach: Lustre

- This architecture allows **I/O bandwidth to be scaled** by adding more OST's.
- Of course to fully exploit this we almost certainly need to have **multiple nodes writing simultaneously**.

Approach: Lustre

- Unfortunately several ‘quirks’ of Lustre make achieving high throughput **frustratingly difficult**.

Approach: Lustre

1. Default stripe counts are often inadequate.

   - It is not uncommon for files to default to a single stripe.

   - Hence, unless the user knows how to use `lfs` **performance will be poor**.

Approach: Lustre

2. Its baroque locking protocol can easily lead to thrashing—even when writes do not overlap.

- Given **acquiring and relinquishing locks is expensive** this can destroy performance.

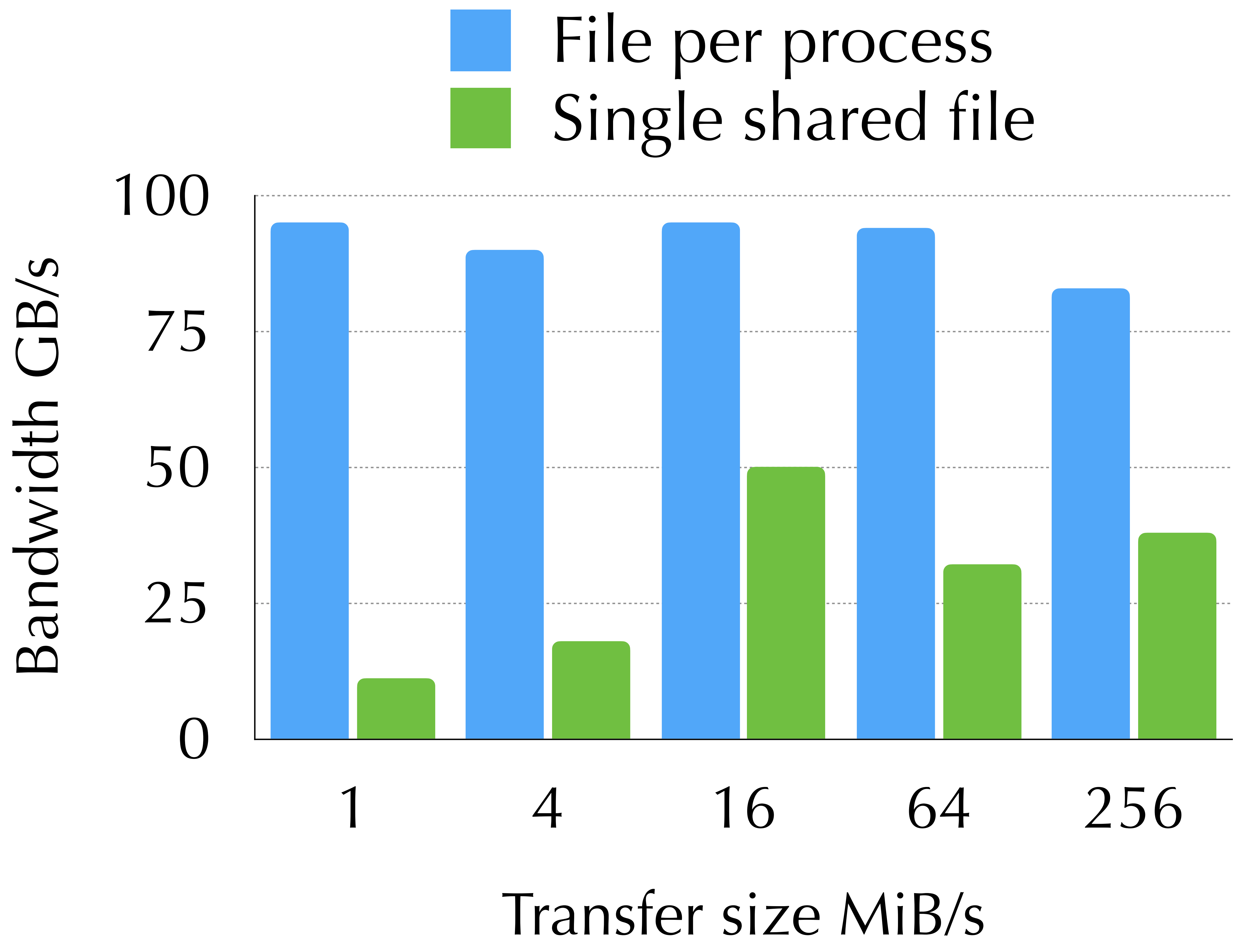
- Not uncommon for multiple independent writers to **deliver lower throughput than a single writer**.

Approach: Lustre

- The most obvious means of avoiding these issues is to have **each node write out its own file**.
- This **completely sidesteps the locking issue** and since files are usually assigned an OST at random **having a single stripe is not an issue**.

Approach: Lustre

- Data from **Moore et al.** (2018).
- 192 ranks.
- 24 OSTs.
- POSIX I/O.



Approach: Lustre

- This is not a workable solution, however.
- Firstly, having solutions split across many files **goes against requirement 5 (simple)**.
- Secondly, and more importantly, it **doesn’t scale**…

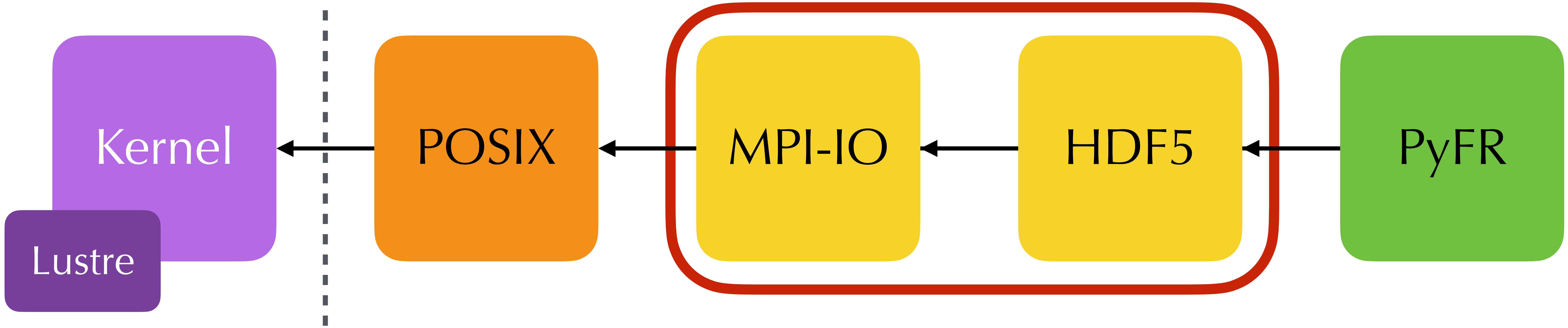
Approach: Lustre

3. Its metadata performance is awful.
    - Often an order of magnitude worse than NFS.
    - Practically limited to **~1,000 files per directory**.

Approach: Lustre

- We could try for a **hybrid solution** where a small number of ranks share a file.
- But, this isn't simple and leaves us with a tunable.
- So, the question becomes how can we resolve issues 1 and 2 such that a **single file is viable**?

Approach: Lustre



- The source of these issues are due to the **imperative rather then declarative nature of our I/O libraries**.

Approach: Solution

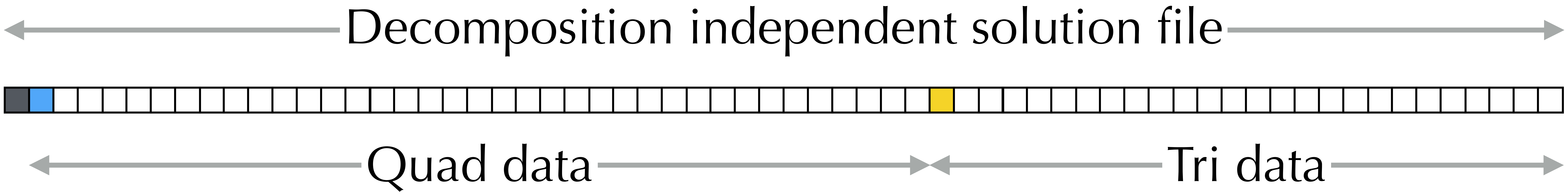
- By taking full ownership of the I/O stack **stripe count and size issues can be trivially resolved**.
- We know how big our solution file will be so can simply have the root rank ensure it is created properly.
- Can even be done with raw ioctl's to **avoid a dependency on the Lustre library**.

Approach: Solution

```
// On the root rank
int fd = open(path, O_CREAT | O_EXCL | O_WRONLY |
                    O_LOV_DELAY_CREATE);
struct lov_user_md opts = {
    .lmm_magic = LOV_USER_MAGIC,
    .lmm_stripe_size = 128*1024*1024,
    .lmm_stripe_count = -1,
};
ioctl(fd, LL_IOC_LOV_SETSTRIPE, &opts);
```

Approach: Solution

- The root rank can then reopen the file with **serial HDF5 to stub out the relevant solution arrays**.
- Then, the **on-disk offset** of each array can be queried and broadcast to all other ranks.



Approach: Solution

- The locking issues can be avoided by having all ranks **acquire a group lock**.
- This comes at the cost of POSIX semantics but given our use case is **write only** this is not an issue…
  - …although it is an issue for MPI-IO and HDF5.

Approach: Solution

```
// On each rank
int fd = open(path, O_RDWR);
ioctl(fd, LL_IOC_GROUP_LOCK, group_id);

// Write out our portion of each array
pwrite(fd, quad_buf, quad_size, quad_off);
pwrite(fd, tri_buf, tri_size, tri_off);
```

Approach: Solution

- Further, as we control the stack we can **improve robustness** by spawning a thread to issue the write calls.
- This gives us **guaranteed asynchronous I/O**.
    - *Cf*. OpenMPI.

Approach: Solution

- With this **I/O is no longer in the critical path**.
- This makes us extremely robust to abused or under-provisioned file systems.

Approach: Solution

- This I/O stack also has packaging benefits.
- Being in Python we use h5py for wrapping HDF5.
- This is often installed with `pip install h5py` which **bundles its own version of HDF5**.
- This bundled version is typically a **serial build**.

Approach: Solution

- **Preliminary results** compared with our existing parallel HDF5 file format in terms of wall-clock time reductions:
  - ~5% for I/O lite simulations on well-configured systems.
  - ~15% for I/O heavy simulations in sub-optimal environments.

Motivation PyFR Requirements Approach **Conclusions**

Conclusions

- Have described a high-performance file format suitable for **exascale discontinuous spectral element simulations**.
- Outlined techniques for avoiding typical I/O pitfalls to enable **scalable single-file operation**.
- Code available in the develop branch of PyFR.

Acknowledgements

- **Air Force Office of Scientific Research** for support under grant FA9550-23-1-0232.

Backup Slides

File Creation

- Consider the following snippet:

```
❯ cat create.py
import h5py

with h5py.File('file.h5', 'w') as f:
    r = f.create_dataset('r', shape=(2**36))
    r[-1] = 0.0

❯ time python create.py
python create.py … 0.101s total
```