Preparing XGC for Exascale Science on Aurora

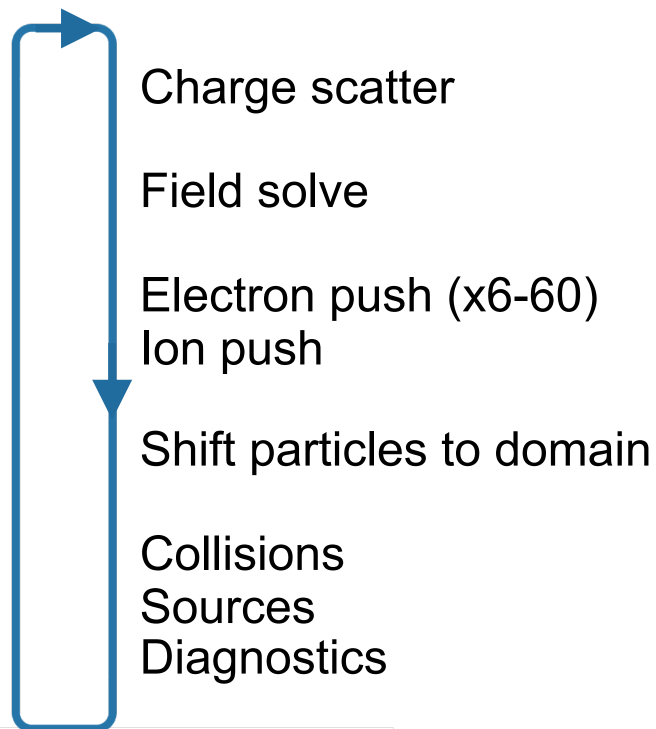
A. Scheinberg[1], T. Williams[2], E. Suchyta[3], K. Huck[4], S. Ethier[5], CS Chang[5]

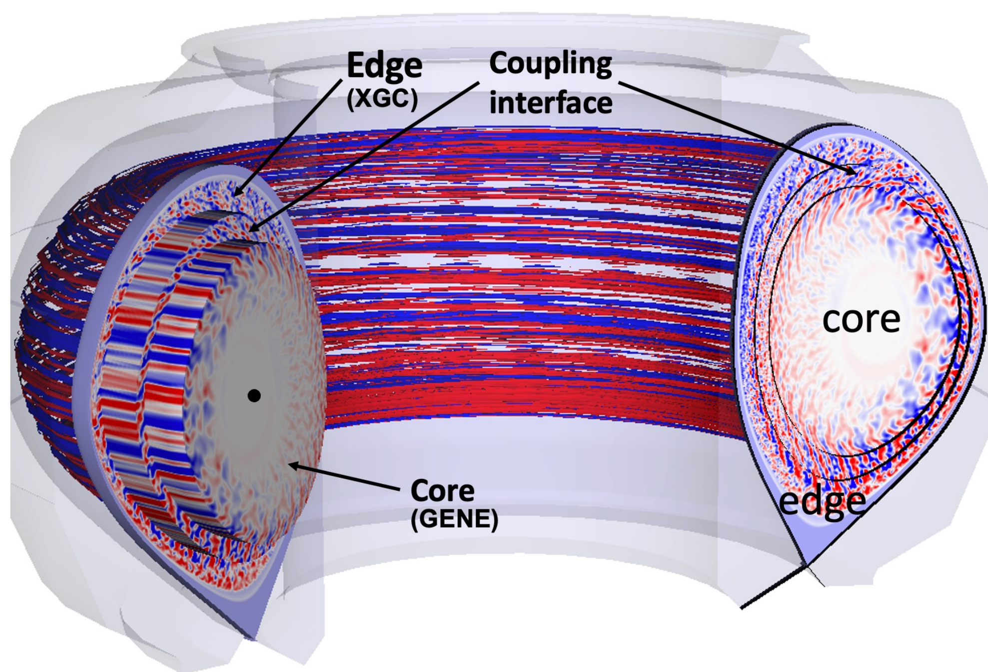
May 24, 2023

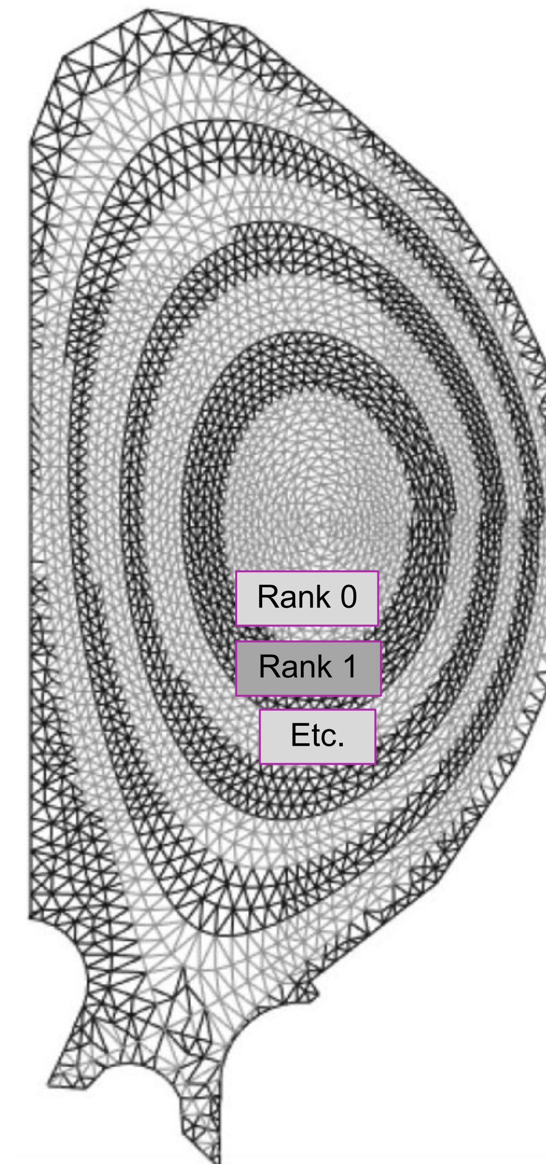
[1]Jubilee Development
[2]Argonne National Laboratory
[3]Oak Ridge National Laboratory
[4]University of Oregon
[5]Princeton Plasma Physics Laboratory

XGC introduction

- Tokamak plasma physics code specializing in edge physics and realistic geometry
- Gyrokinetic (i.e. 6D → 5D via analytic reduction using gyro-averaging)
- Particle-in-cell with an **unstructured 2D grid** and **structured toroidal dimension**
- Domain decomposition: toroidally sliced, then each MPI rank handles a subset of the grid

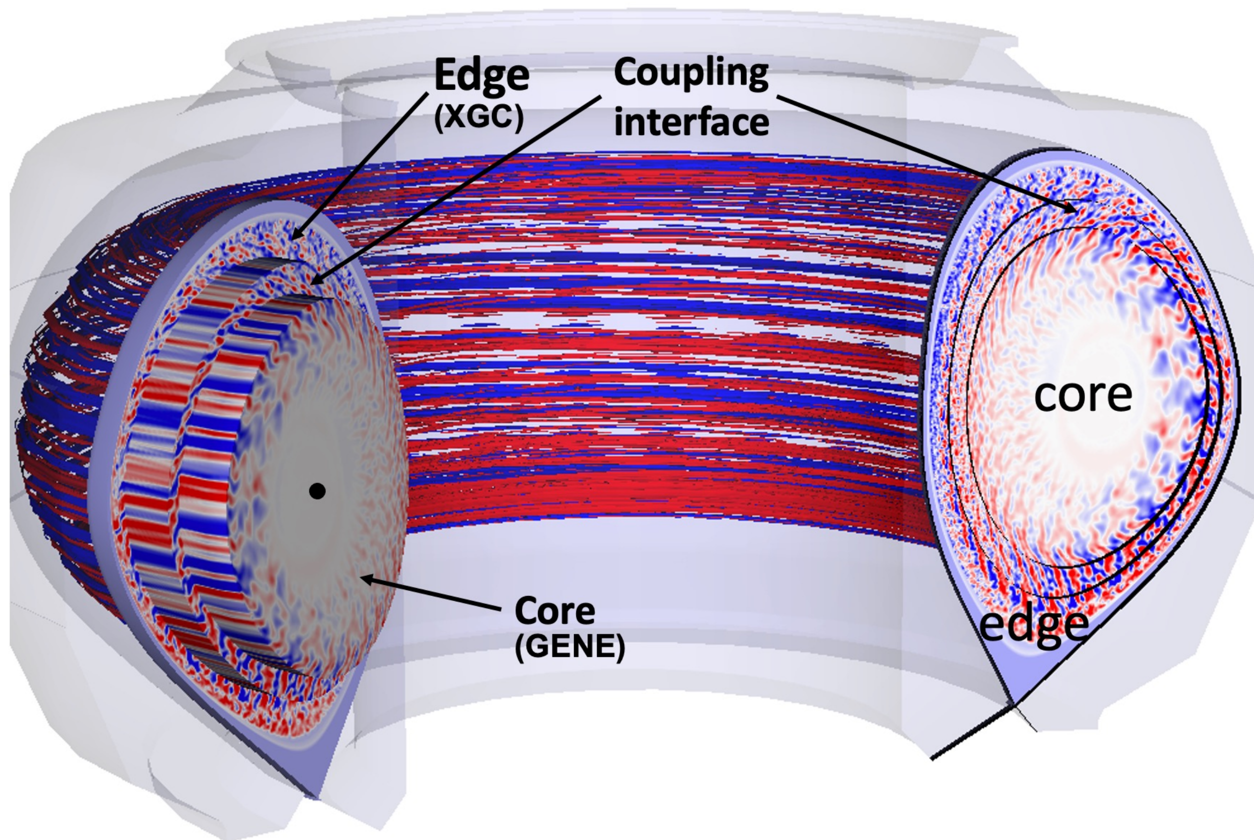


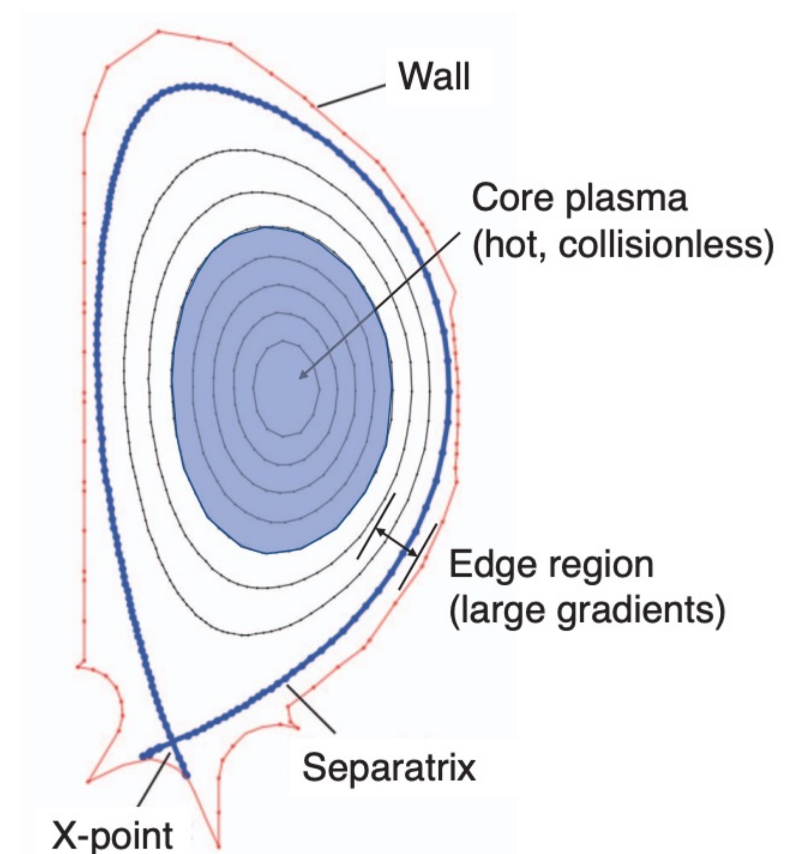




Whole Device Model (WDMApp)

- ECP-WDM project
- Couples XGC with a core code (GENE or GEM) for ”whole device modeling”
- The vast majority (>90%) of time spent is spent in XGC, so its optimization is most critical





XGC engineering challenges

- A wide array of physics features and modes must be supported, e.g.:
  - Delta-f (perturbation from Maxwellian) and full-f
  - Electrostatic (magnetic field perturbations due to plasma ignored) and electromagnetic
  - Axisymmetric (“XGCa”)
  - Impurities
  - Neutral particles with atomic cross-sections
  - Coupling (GENE, GEM, XGC, in-situ analysis)
- These different modes of operation can drastically alter landscape of performance bottlenecks
- Physics in constant state of development
  - Some changes are modular additional features
    - e.g. new sources
  - But others are (sometimes fundamental) structural modifications, e.g.:
    - Stellarator
    - 6D
    - Split-weight scheme
    - Multirate timestepping
    - Time telescoping
    - Implicit timestepping

Target architectures

Machine	Cori KNL	Summit	Perlmutter	Frontier	Aurora
Testbed				Crusher	Sunspot
Vendor	Intel	Nvidia	Nvidia	AMD	Intel
“Native” language		Cuda	Cuda	HIP	SYCL
GPU resources per rank		1 V100	1 A100	½ MI250X	
Host memory per rank	96 GB	85.3 GB	64 GB	64 GB	
Device memory per rank		16 GB	40 GB	64 GB	

Trade-offs – memory, computation, communication

- “Distributed calculation + gather” vs “Full calculation on each process” **(comms vs computation)**
    - (Incidentally, makes FLOPS comparisons even less meaningful)
- Pre-computation vs on-the-fly recalculation **(memory vs computation)**

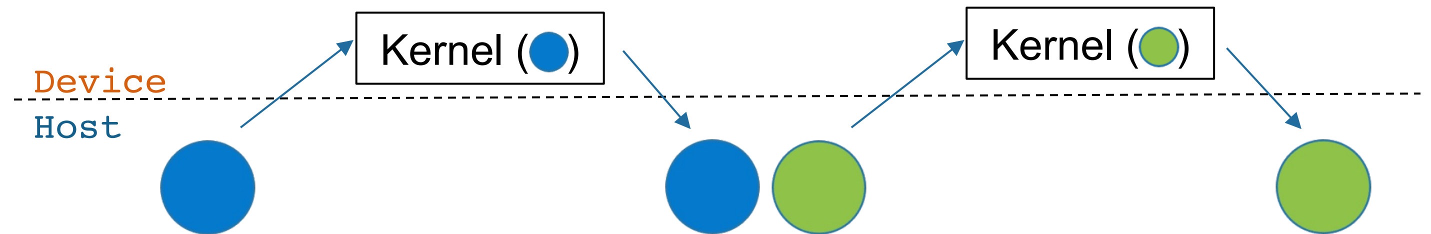
Some data is better off stored on device memory if available, but otherwise must be transferred frequently between host and device

Particle memory management: Reside in host or device memory?

- Different optimal memory management for particles on different architectures
  - Depends on available memory per GPU and per MPI rank, and communication rate

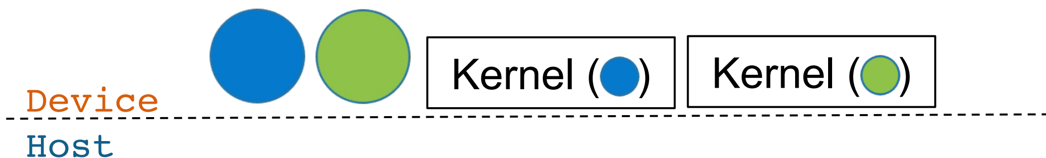
Particles sent to device for each kernel

- More particles possible – only one species needs to fit on the GPU at a time
- Extra communication time



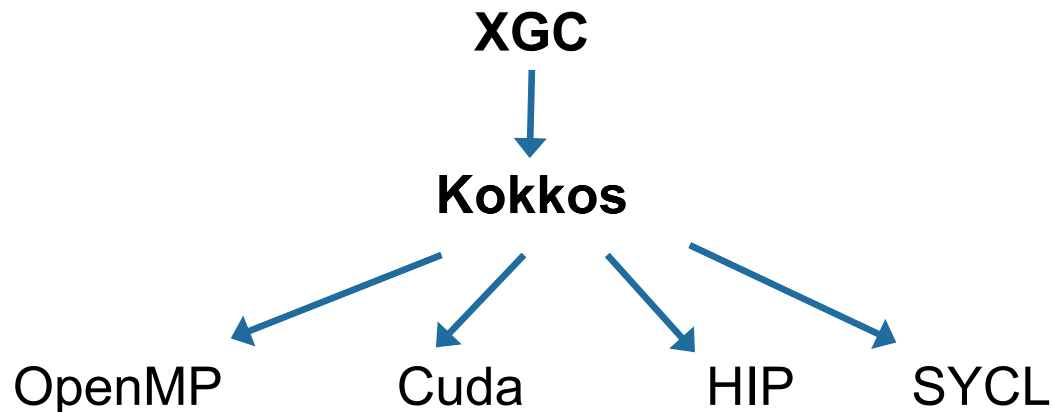
All particles reside permanently on GPU

- No time spent on communication
- Number of particles per species limited by GPU memory



Exascale Preparation: Kokkos and C++

Kokkos: a portability abstraction layer that maps to OpenMP, Cuda, HIP, and SYCL



XGC Timeline

Pre 2019
Fortran code with 3 versions of dominant kernels:
- OpenACC collisions and
Cuda Fortran electron push for GPUs
- Vectorized CPU version,
- Simple reference CPU version

2019
Fortran code using wrappers and macros to offload with Kokkos
- Tedious and inflexible
- Unclear for AMD/Intel GPUs

Present day
C++ code with non-critical components left in Fortran

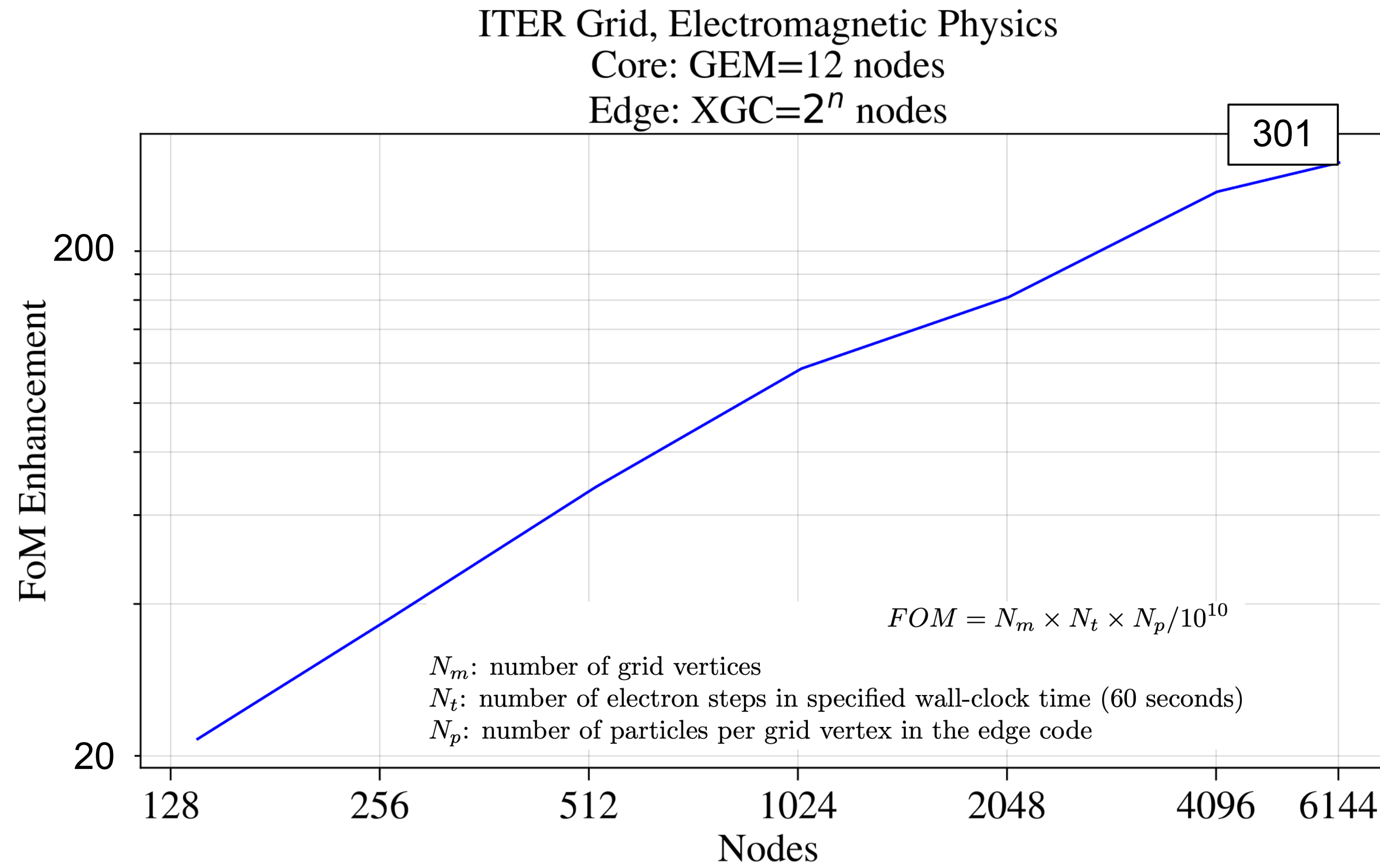
EXASCALE COMPUTING PROJECT

XGC engineering approaches

- Portability with Kokkos and Cabana (ECP-CoPA particle library)
- Major focus on encapsulation/modularity
- Templating
  - e.g., electron push and ion push are quite different (electrons subcycle and are drift kinetic, ions are gyrokinetic) but use the same code
  - Easier than before to experiment/swap out options
- Stand-alone kernels
  - Most major code components can be run independently
  - Use the same code base (no copies!):
    - Never outdated
    - Don't require extra maintenance
    - Improvements immediately benefit the full code
- Testing/CI
  - Unit tests, kernel regression tests, and run test on every pull request
  - Automated physics testing still in progress

Key Frontier result: ECP-WDM KPP-FOM achieved

- Performance requirement: Using a DOE exascale platform, achieve **50X** performance improvement over the original simulations running XGC alone on Titan
- Measured: **301X** enhancement on Frontier with XGC-GEM coupled code



Coupling Data Between Codes on Frontier

Experimented with several I/O code-coupling strategies with EFFIS

- File-based:
  - Used in our FOM runs for simplicity
- Memory-based: MPI, RDMA, or TCP
  - MPI:
    - `MPI_Open_port` did not work (needed in MPI-based coupling orchestration)
    - `MPI_Init_thread` was unstable (especially > 4K node count), and sometimes errors mentioning `MPI_Init_thread` were triggered when setting `--threads-per-core=2` for the job
  - RDMA:
    - Libfabric for RDMA with ADIOS2 did not work
  - TCP:
    - Successful

Implications for Aurora

- Maintaining multiple coupling methods will be helpful for getting up-and-running fast

Transient system issues encountered

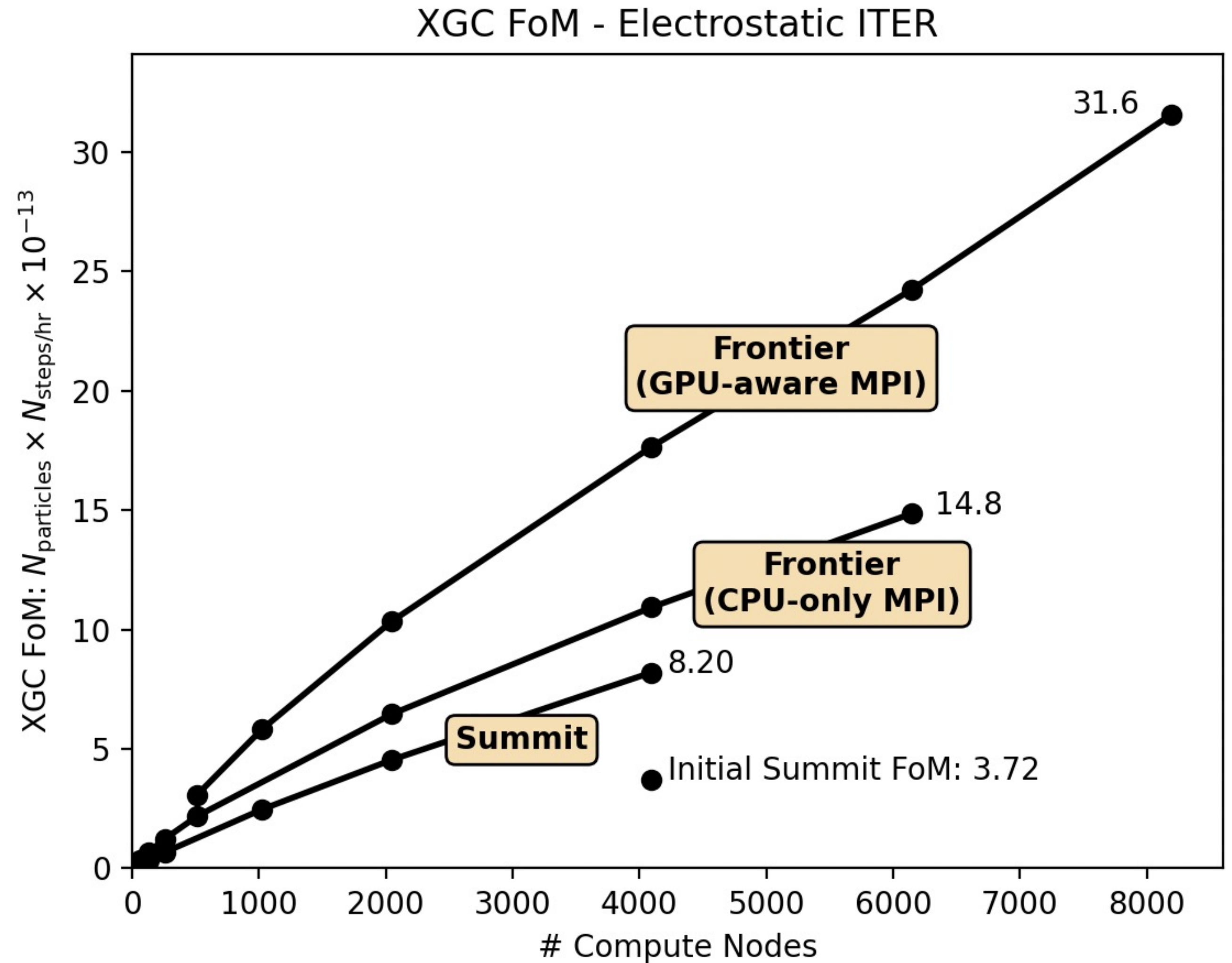
- File system issues
- Network/MPI issues
- Node failures

Implications for Aurora

- Expect intermittent failures beyond ones control
- Lots of re-running of identical simulation
- Optimize simulation initialization

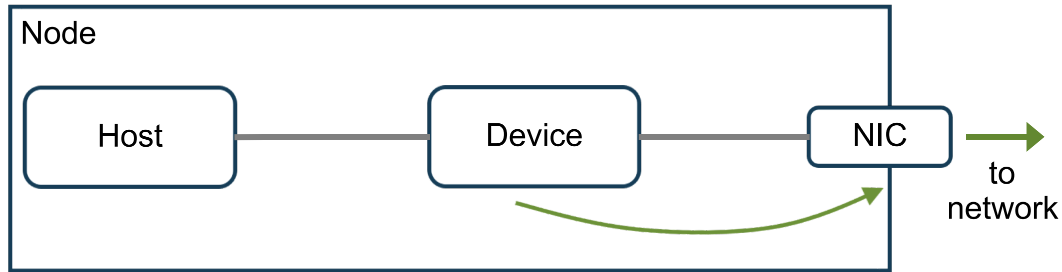
XGC electrostatic benchmark on Frontier

- Performance enhancement from *initial* Summit to *initial* Frontier: **8.5x**
  - Initial to current Summit: **2.2x**
  - Current Summit to Frontier: another **3.9x**
    - vs 9x theoretical peak FLOPS
- **GPU-aware MPI** drastically improves performance

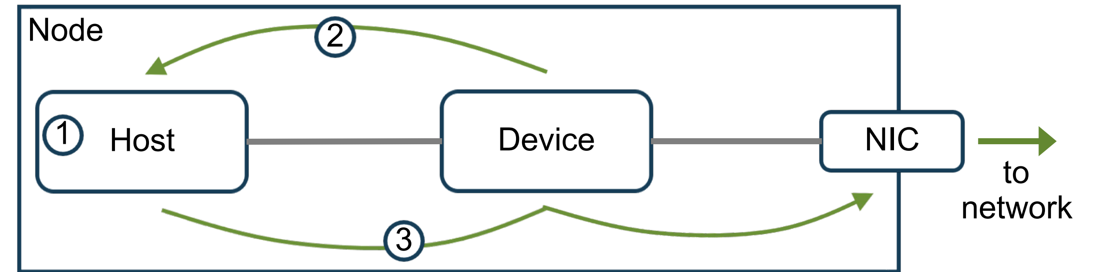


Why is GPU-aware MPI so much better on Frontier?

- NICs are connected directly via GPUs



(No extra allocations/copies required)



- CPU-only MPI requires extra steps

1. Allocate host memory
2. Send data from device to host
3. Do MPI comms (via GPUs)

(And reverse for received data)

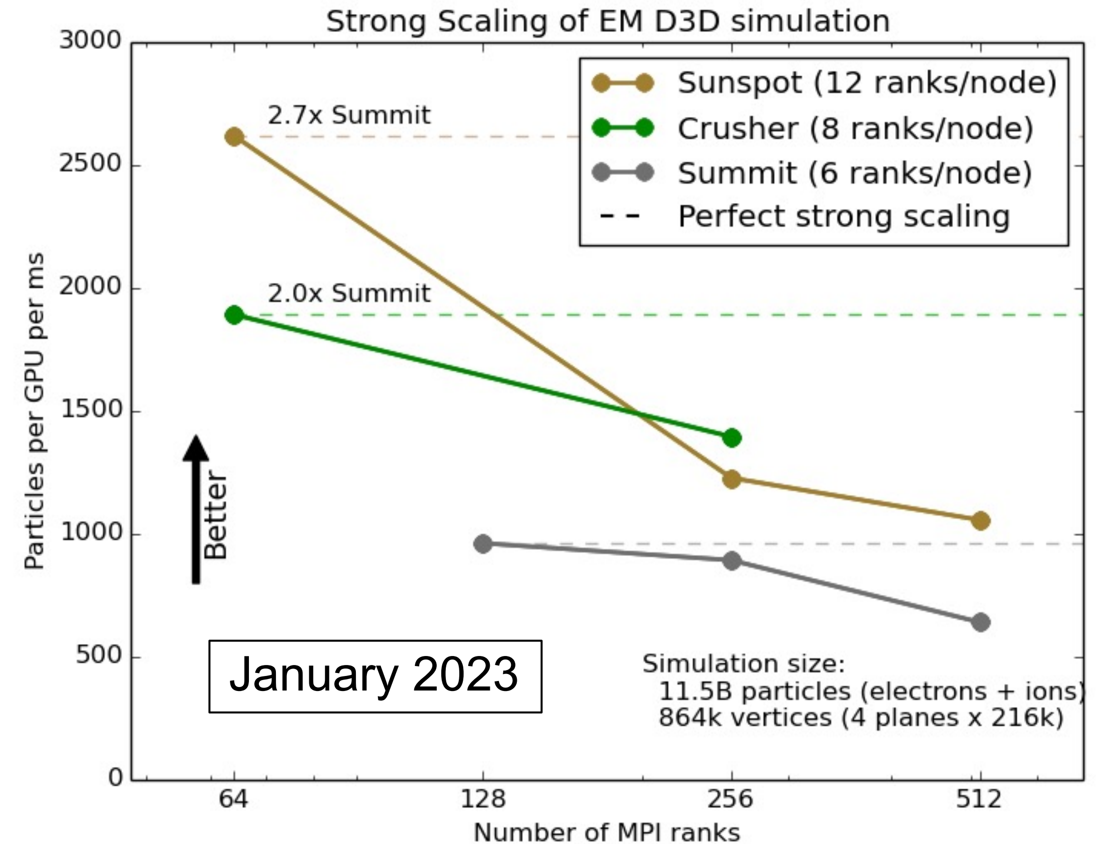
Implications for Aurora

- GPU-aware MPI will be worth using (but maybe not as dramatically)

Strong Scaling

Same simulation size, different amount of resources

- Fewer compute nodes → less communication → more efficient resource usage
- Perfect strong scaling: no efficiency gains from using fewer compute nodes
  - For XGC, improvement would probably require overlapping communication and computation



Implications for Aurora

- Should pack simulation into fewest nodes possible if trying to optimize efficiency rather than overall wall-clock time

EXASCALE COMPUTING PROJECT

Did Frontier behavior match extrapolations from Crusher?

(Likewise, what can we infer from Sunspot?)

- Very similar performance at same scale (~100 compute nodes)
  - GPU-aware MPI correctness bug identified on Crusher, workaround found which helped on Frontier
- Unexpected challenges at large scale (>2,000 nodes)
  - Theoretical GPU memory: 64 GB per MPI process
  - Actual available memory still unclear:
    - Sufficient memory must be available for GPU-aware MPI operations
    - Encountered a bug/apparent memory leak: memory used by MPI(?) is not relinquished
  - This prevented us from packing larger simulation size into fewer ranks for additional efficiency

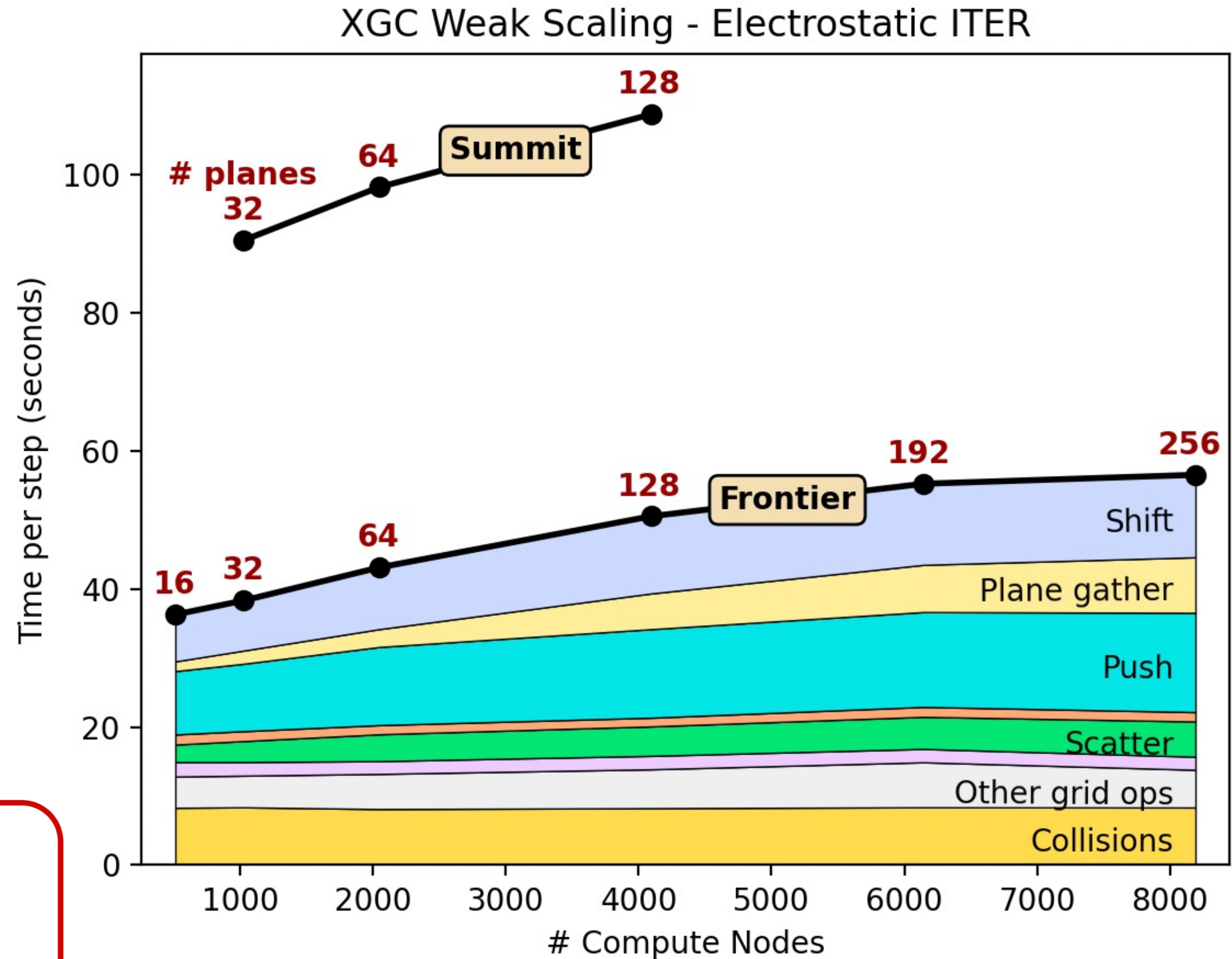
Implications for Aurora

- Safer to leave a generous memory margin for initial simulations
- Maintain less performant but less demanding options
  - CPU-only MPI
  - CPU-resident particles

Weak scaling on Frontier

New challenges due to high toroidal resolution

- Particle push
  - Typically scales perfectly
  - Higher toroidal resolution may result in worse memory access patterns; toroidal sorting might be needed
- "Plane gather"
  - Domain decomposition in toroidal "planes"
  - 2x planes = 2x the time
  - Starting to impact time-to-solution
  - New algorithm introduced: sends less data, but more (duplicate) computations done locally.



Implications for Aurora

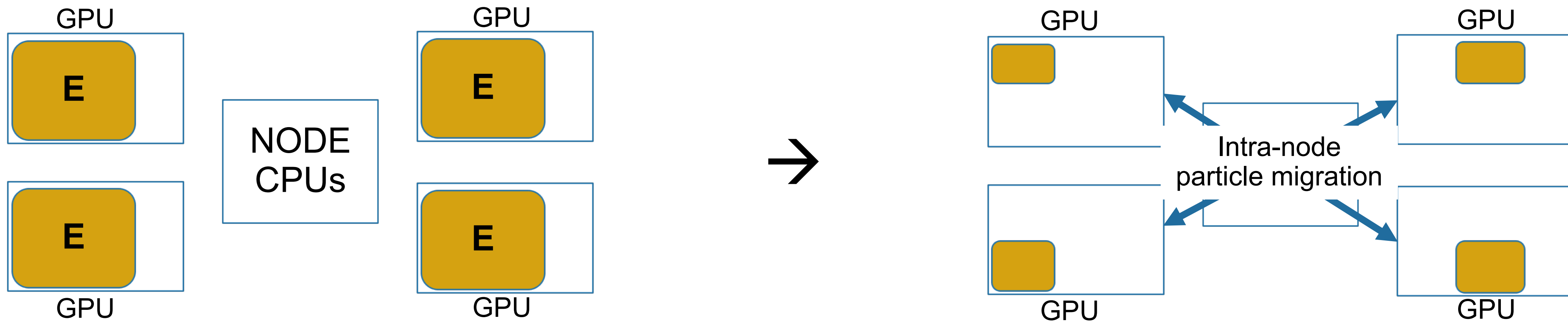
- New trade-offs may become worthwhile at scale
- Stand-alone component kernels should be designed to imitate large scale

Plane gather: computation vs communication trade-off

- "Plane gather" gathers the electric field **E** from all planes, since the full domain's field is currently needed on every MPI process
- Original algorithm
  1. Each plane computes its local **E** from Φ
  2. Sends resulting **E** to all other planes
- New algorithm
  1. Each plane sends its local Φ to all other planes
  2. Computes **E** from Φ for *all* planes
- New algorithm is **2.7x** faster on 4,096 Frontier nodes
  - 6x less communication; N_{planes}x more computation

In progress: Intra-node domain decomposition of EM fields

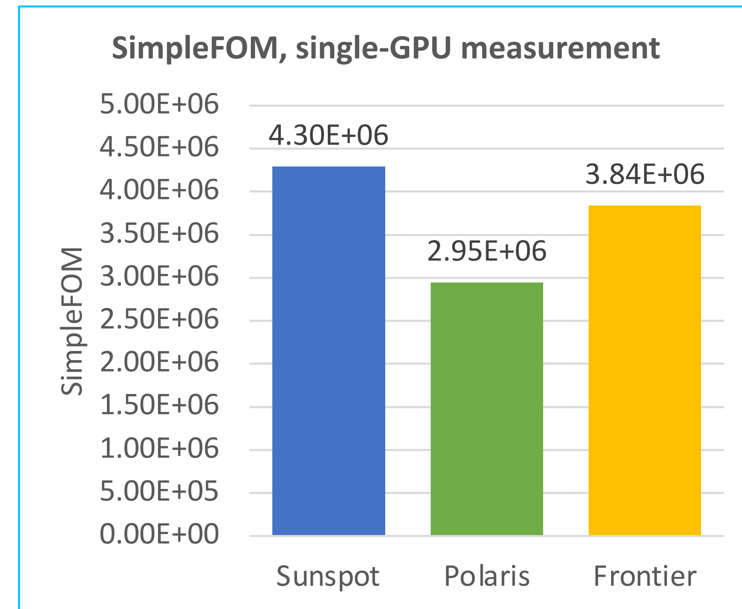
- Electrons need full domain; network-wide particle migration between each electron subcycle too expensive
- Frontier/Aurora science plans involve higher resolution → more field data
  - Currently the limiting factor in simulation resolution



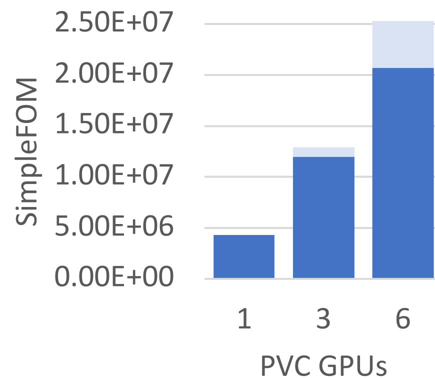
- Experimental solution: Intra-node domain decomposition
  - Intra-node particle migration may be cheap enough to be worth it
  - But there are some subtleties involved because of our gather/push algorithm
  - Gets complicated quickly, e.g. local load balancing clashing with network-wide load balancing

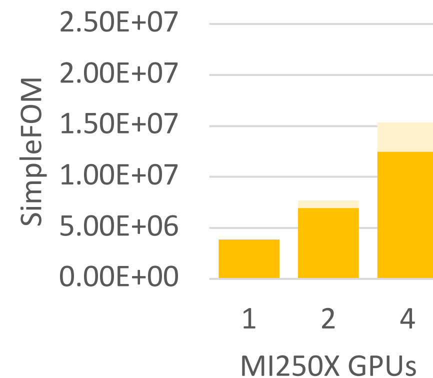
Sunspot status and comparison

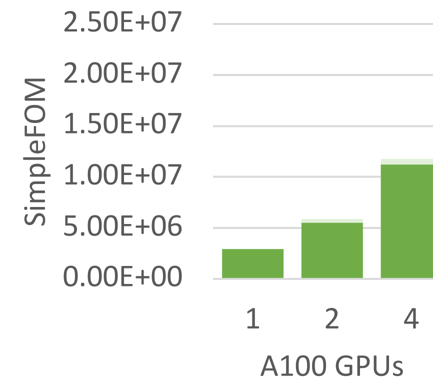
- Performance comparable to Polaris and Frontier
- Recently investigating bug (nondeterministic memory corruption)
  - Unclear if due to XGC changes, Sunspot changes, or combination
- Very slow link times
  - ~10 minutes, makes debugging process difficult



Single-Node Weak Scaling







Summary

- For optimized exascale, XGC needed not just GPU offloading, but also algorithmic flexibility
- Looking forward to science on Aurora!

- Maintaining multiple coupling methods will be helpful for getting up-and-running fast

- Expect intermittent failures beyond ones control
- Lots of re-running of identical simulation
- Optimize simulation initialization

- GPU-aware MPI will be worth using (but maybe not as dramatically)

- Should pack simulation into fewest nodes possible if trying to optimize efficiency rather than overall wall-clock time

- Safer to leave a generous memory margin for initial simulations
- Maintain less performant, less demanding options
  - CPU-only MPI
  - CPU-resident particles

- New trade-offs may become worthwhile at scale
- Stand-alone component kernels should be designed to imitate large scale