Code development ideas

- Framework ideas
  - Abstract blocks
  - Accelerator directives
- Performance ideas
  - Block preconditioning
  - Asynchronous FFT communication

Abstract current block structures

- Fortran 2003
- Block objects would be defined as an abstract interface with common methods.
  - This object abstraction is similar to the “module procedure” abstraction used extensively for functions.
- Loops over blocks would not need to know the underlying block type (e.g. rblock, tblock, trigblock, uquadblock, etc.).
- Common interfaces would require FE integration, nodal operation, assignment and arithmetic methods along with the block export for the external solvers.

Example code

```
DO ibl=1,nrb
  IF (ibl<nrbl) THEN
    rblock_get_rhs(rb(ibl), …)
  ELSE
    tblock_get_rhs(tb(ibl), ...)
  ENDIF
ENDDO
```

```
DO ibl=1,nrb
  bl(ibl)%get_rhs(... )
ENDDO
```

Abstract blocks (2)

- Pros:
  - Once abstraction is in place, implementation of additional block types becomes “easy”.
  - Creates extremely flexible code (essentially unlimited basis function flexibility).
  - New block types could bring geometric and convergence advantages.
- Cons:
  - Significant development time as many routines have hard-coded rblock assumptions.
- Development time: ~ **8 months** (e.g. abstract tblocks)
- Probability of success: ~ **99%** (50% with closures)

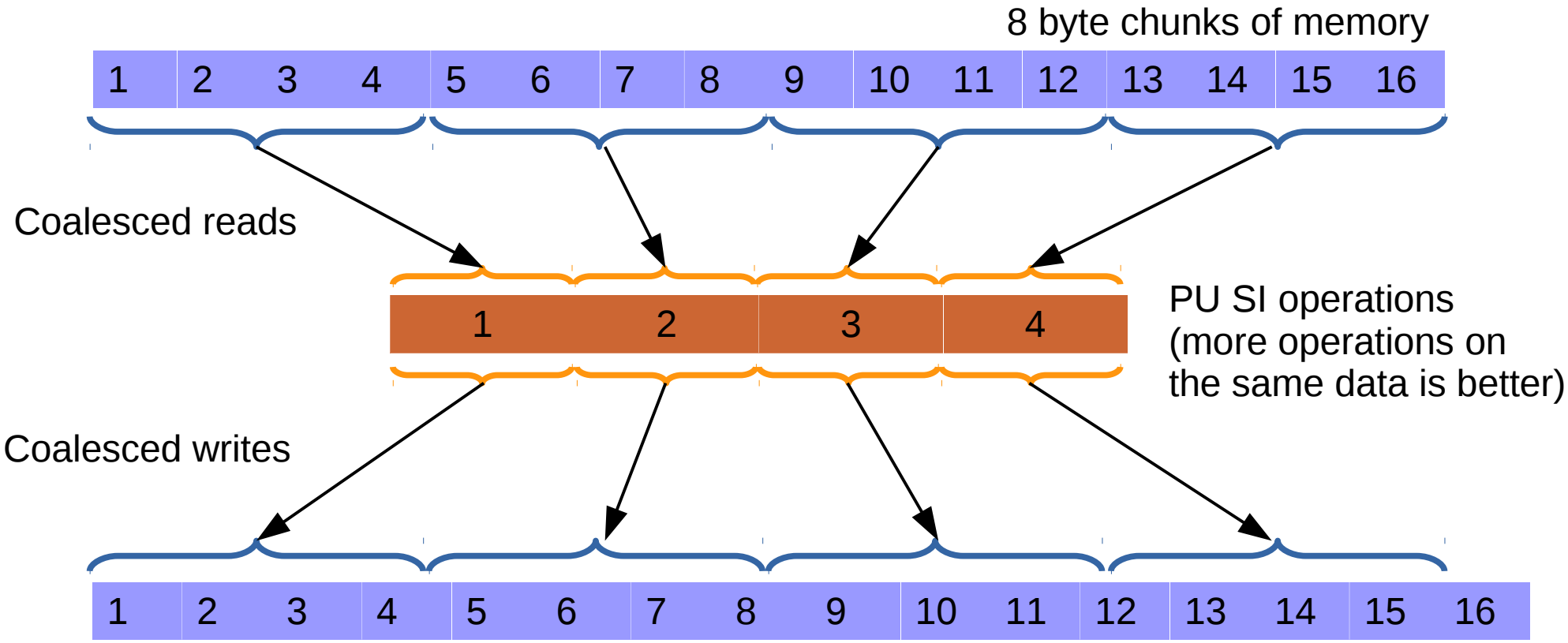
Accelerator directives

- Accelerated computing [e.g. Titan (ORNL, Nvidia K20s) and Cori (NERSC, Xeon Phi)] is the future.
  - Our current parallelism may not be able to effectively use the Xeon Phi, and can not use the K20s (no CUDA, OpenCL or OpenACC directives).
  - It is likely the memory bandwidth with be a limiting factor for the intranode scaling on the Xeon Phi. This bandwidth will be improved on next generation devices (*Knight's landing* as planned for Cori) and **may not** be a limitation.

Aside on parallelism

- NIMROD currently uses block parallelism (both for MPI and OpenMP threads)
  - This is a MIMD (multiple instruction, multiple data) model.
- Accelerators may be memory bound, instead of compute bound, with this approach.
  - We are memory bound with our intra-node scaling on Hopper, Edison, Titan CPUs, etc.
  - Accelerators work best with a SIMD (single instruction, multiple data) approach and coalesced memory access patterns (at a minimum this is true for CUDA but it can also help with vector PUs). For example, consider an array operation that assigns array indices to different processors.

Aside on memory access patterns



Reality is of course more complicated: there's RAM bandwith, Caches, context switching (with over-subscription of PUs), etc.

Fortran array syntax helps write memory friendly code.

Accelerator directives (2)

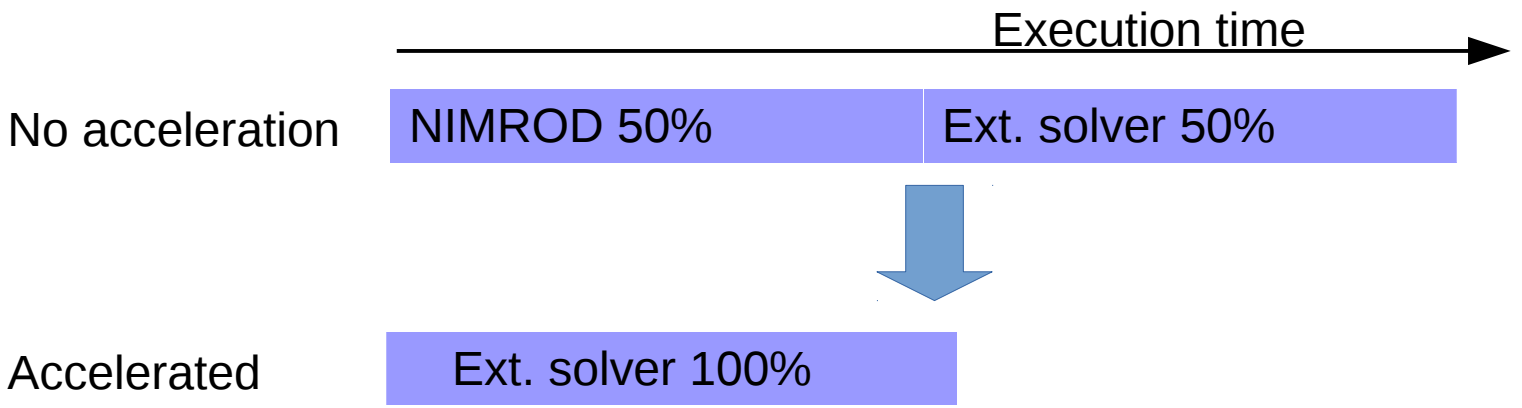
- Goal would be to provide a framework that separates the accelerator directives from the physics equation coding.
  - For example, place OpenACC directives in get_rhs (desired) or block_get_rhs (less desirable, but possibly necessary) and ptr_set/initialization routines.
  - Hidden from physics kernels through layering:
    - adv_be → get_rhs → block_get_rhs → bhmhd_rhs, etc.
  - Need to accelerate a substantial portion of the code to minimize data movement over the PCI bus, otherwise there will be a likely slowdown of the code.

Accelerator directives (3)

- Pros:
  - Appears to be the future of HPC. It is likely acceleration will be necessary for INCITE-level awards.
  - Faster execution.
- Cons:
  - Purely a computation exercise for code execution speed. No modeling benefits unless execution time is the problem.
- Development time: **15 months**
- Probability of success: **30%** (speedup could be small (<2x) without accelerated external solvers)

Aside on an application of Amsdahl's law

- Consider the effect infinite acceleration of NIMROD kernels for a job that uses the external solver 50% of the time.

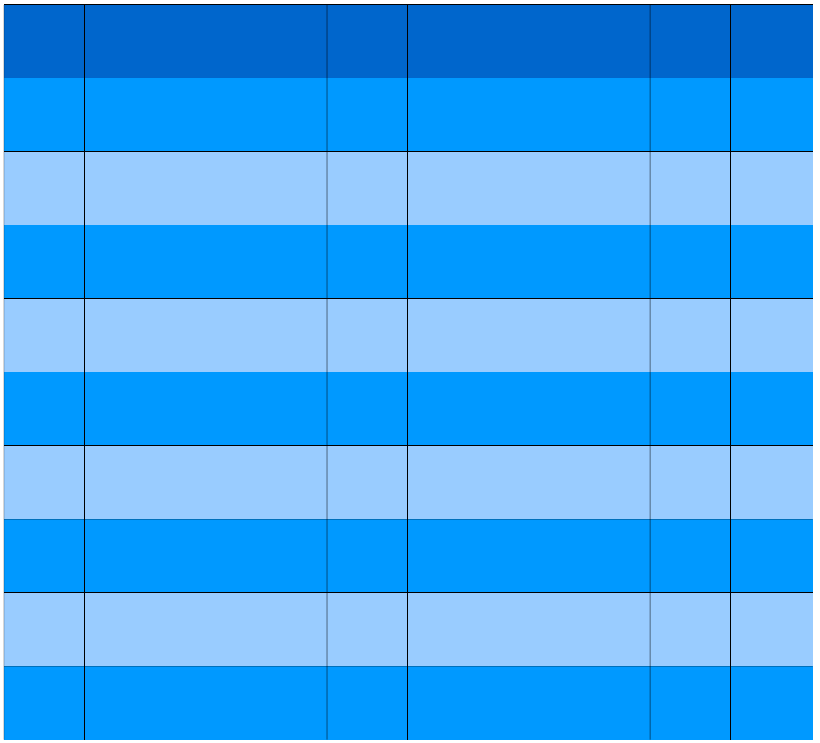


- A lot of work for a 2x speedup.

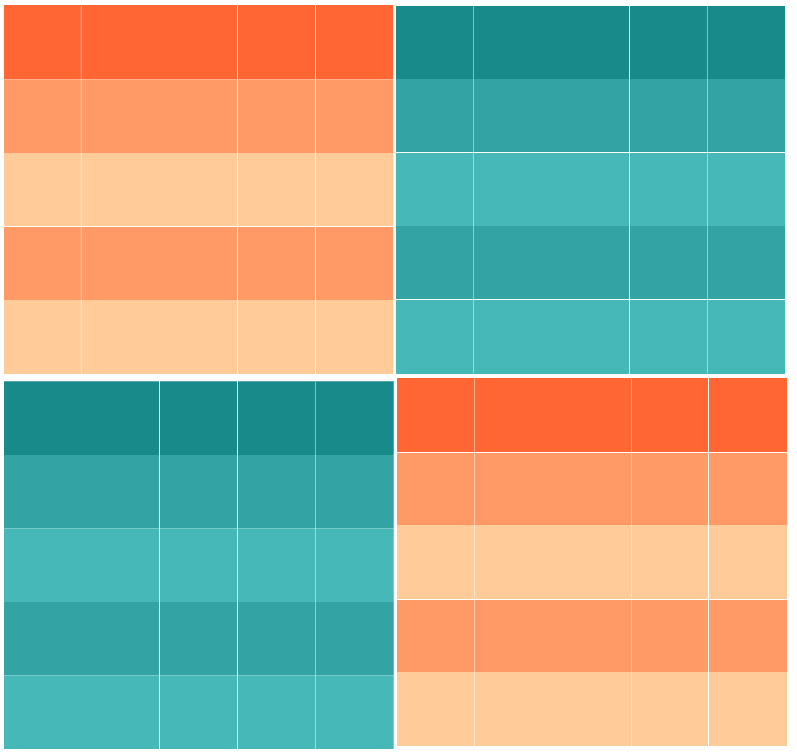
Block Preconditioning

- Breaks FE mesh into (potentially overlapping) subdomains.
- May not effectively precondition for the exchange of global information.
- Need to treat the submatrix BCs carefully (allow energy/wave outflow instead of reflection between domains).

Global FE mesh



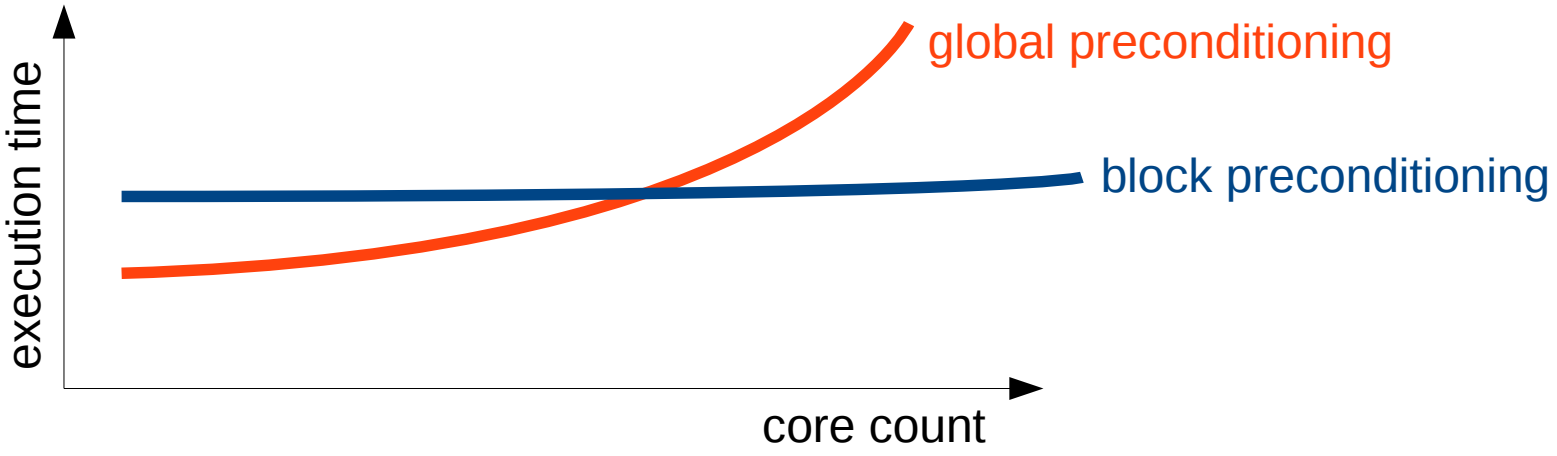
4 subdomains



Block Preconditioning (2)

- High polodial resolution cases require significant memory which can require many nodes.
  - The external solvers scale poorly with cpu counts of ~100.
- Block preconditioning (without external solvers) has been tried before in NIMROD and it was found that global preconditioning was more effective.
- Can we take one step back to take two steps forward?

theoretical weak scaling (flat line is perfect scaling)



Block Preconditioning (3)

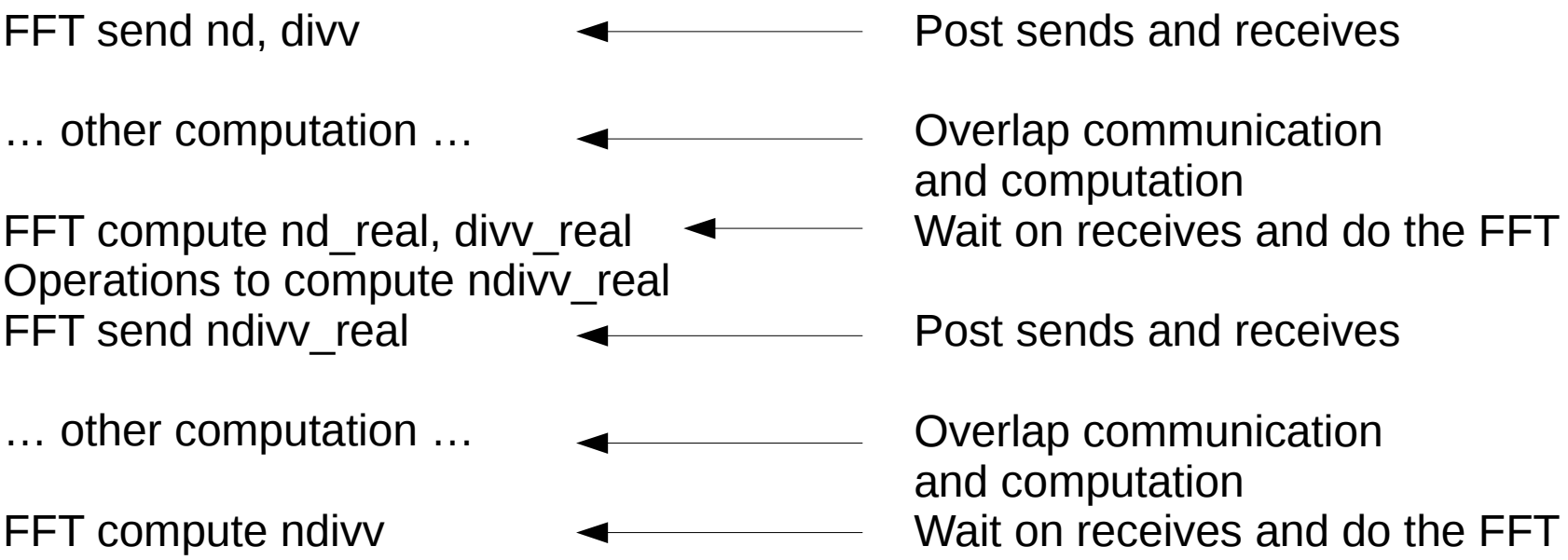
- Pros:
  - Reduces communication
  - Potentially better scaling
- Cons:
  - Increases computation (increased iterations to solution)
- Development time: **4 months (experimental version);** 8 months (robust version)
- Probability of success: **50%** (defined as better scaling plus a “crossing point” relative to global preconditioning at 100s of cores)

Extension: toroidal block preconditioning

- One could potentially implement element-wise dense block inversions in Fourier space.
- May help with violent nonlinear cases.
- Would need yet another set of integrand routines.

Asynchronous FFT communication

- New MPI standard supports asynchronous collective communication.
- Could split FFT calls into 4: FFT sends and compute for real and complex fields.



Asynchronous FFT comm (2)

- Pros:
  - Better Fourier modes parallel scaling
- Cons:
  - ??
- Development time: **2 months (no closures)**
- Probability of success: **95%** (defined as improved parallel scaling)

Topics can be synergistic

- Block preconditioning could allow for the use of an accelerated preconditioner (one block per accelerator).
- Refactoring required to implement one topic can benefit another.

Making refactoring easier

- Remove the predictor-corrector advance and associated routines
- Remove the density options (only full linear and nonlinear).
- Temporarily remove upwinding and closures until the new code is proven to be correct and useful.