TECHNISCHE UNIVERSITÄT DRESDEN

Center for Information Services and High Performance Computing (ZIH)

Introduction to the usage and application of the FD4 library

Developer School for HPC applications in Earth Sciences

10 Nov 2014, Trieste, Italy

Matthias Lieber (matthias.lieber@tu-dresden.de)

Center for Information Services and High Performance Computing (ZIH)
Technische Universität Dresden, Germany

ZIH
Center for Information Services & High Performance Computing

Outline

FD4: Four-Dimensional Distributed Dynamic Data structures
A library for parallelization, dynamic load balancing, and model coupling

- Motivation: Why FD4?
- FD4 Basic Concept
- FD4 Features
- FD4 Usage
- Outlook to Hands-on

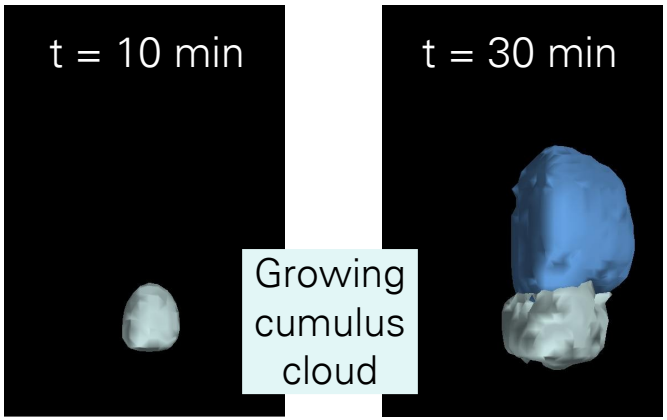
FD4 Motivation: COSMO-SPECS Performance

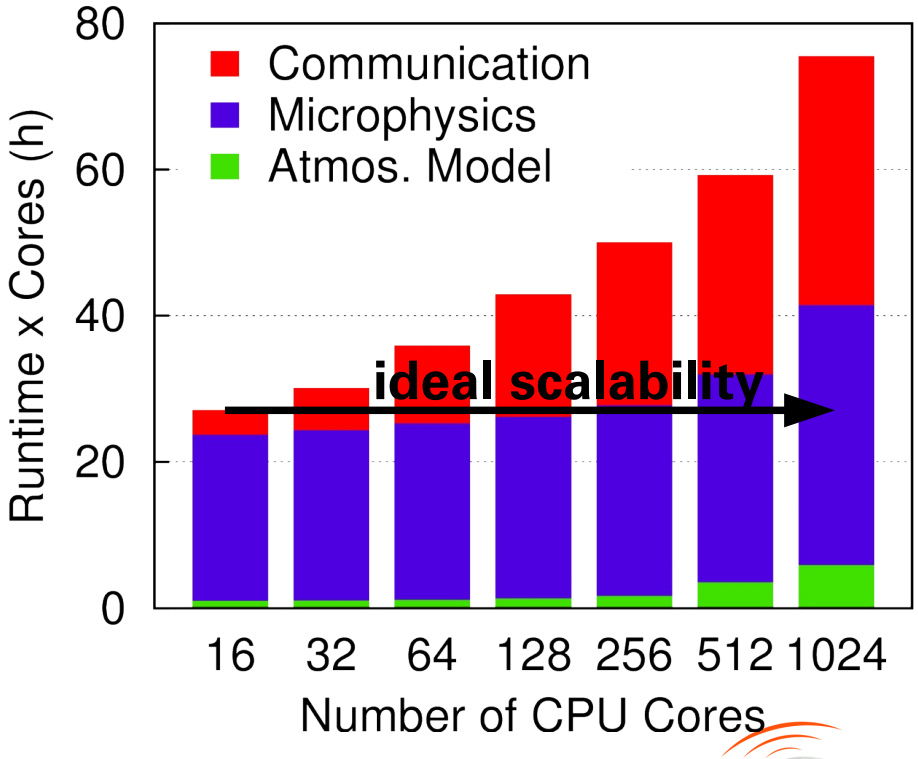
- COSMO-SPECS: Atmospheric model COSMO extended with highly detailed cloud microphysics model SPECS

CONSORTIUM FOR SMALL SCALE MODELING COSMO

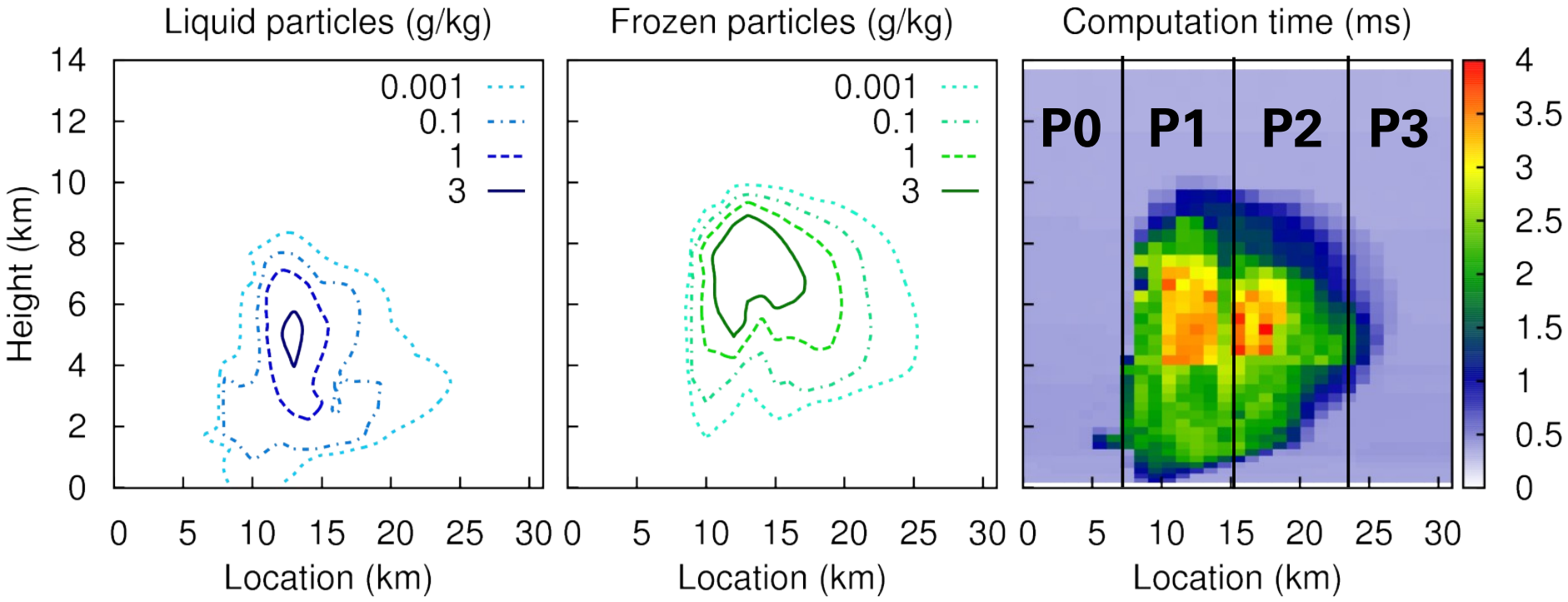
TROPOS
Leibniz Institute for Tropospheric Research

Small 3D case with 64x64x48 grid



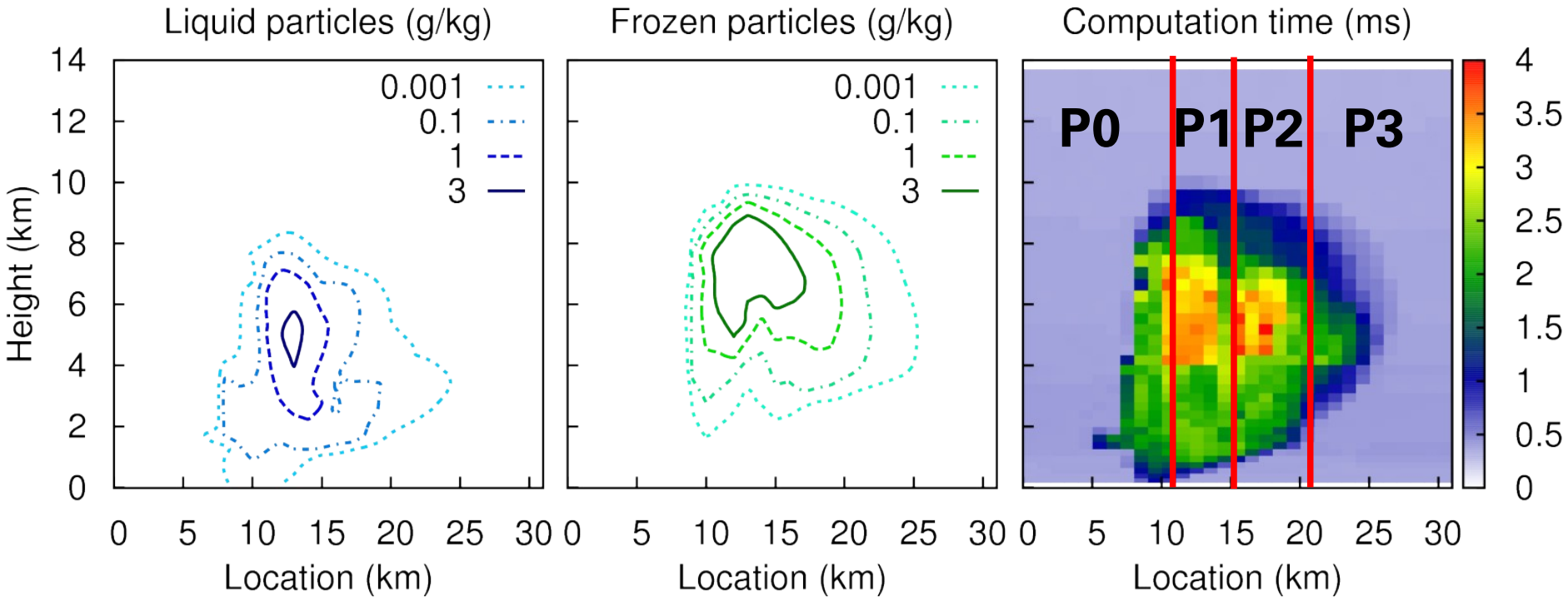


FD4 Motivation: COSMO-SPECS Load Imbalance



- SPECS computing time varies strongly depending on the range of the particle size distribution and presence of frozen particles
- Leads to load imbalances between partitions

FD4 Motivation: COSMO-SPECS Dynamic Load Balancing?



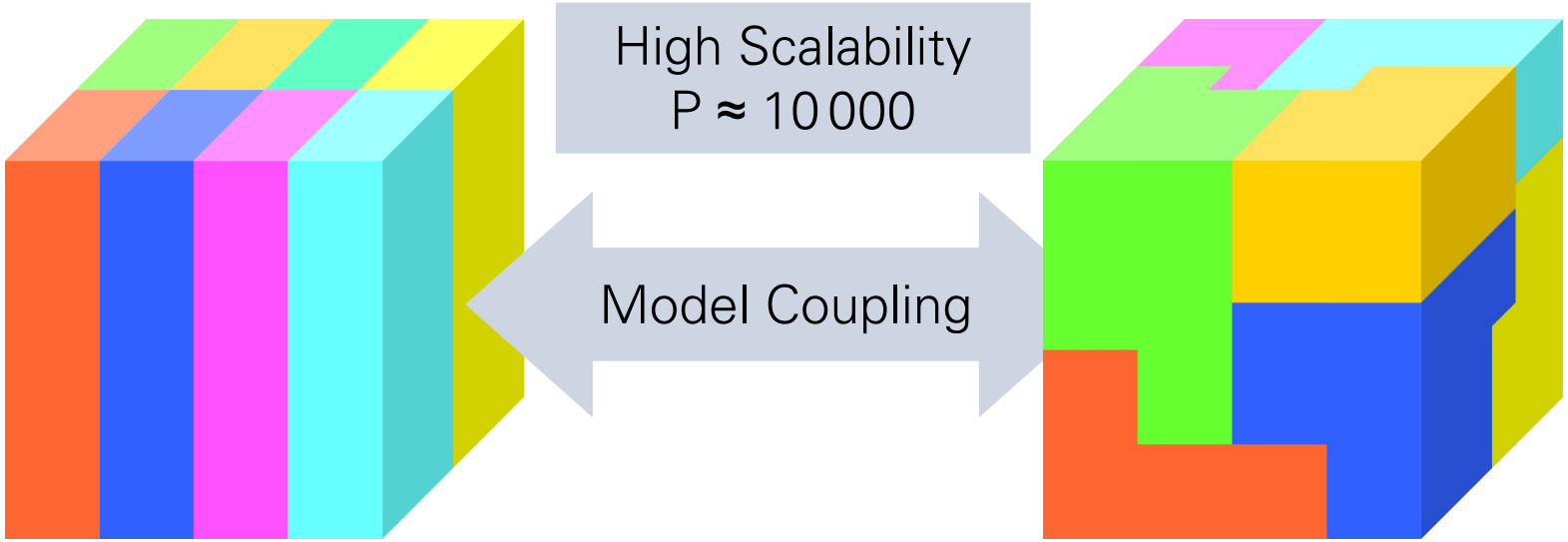
- Dynamic load balancing regularly adapts the partitioning of the grid to workload changes such that all MPI ranks have equal workload
- But COSMO model's data structures are static and not suited for dynamic load balancing

Outline

- Motivation: Why FD4?
- **FD4 Basic Concept**
- FD4 Features
- FD4 Usage
- Outlook to Hands-on

FD4 Basic Concept: Load-Balanced Coupling

Atmospheric Model & Cloud Microphysics | **Cloud Microphysics**



2D Decomposition

Static Partitioning

Block-based 3D Decomposition

Dynamic Load Balancing

Optimized Data Structures

FD4 Basic Concept: Load-Balanced Coupling

Implemented as independent library FD4

High Scalability
P ≈ 10 000

Model Coupling

Block-based 3D Decomposition

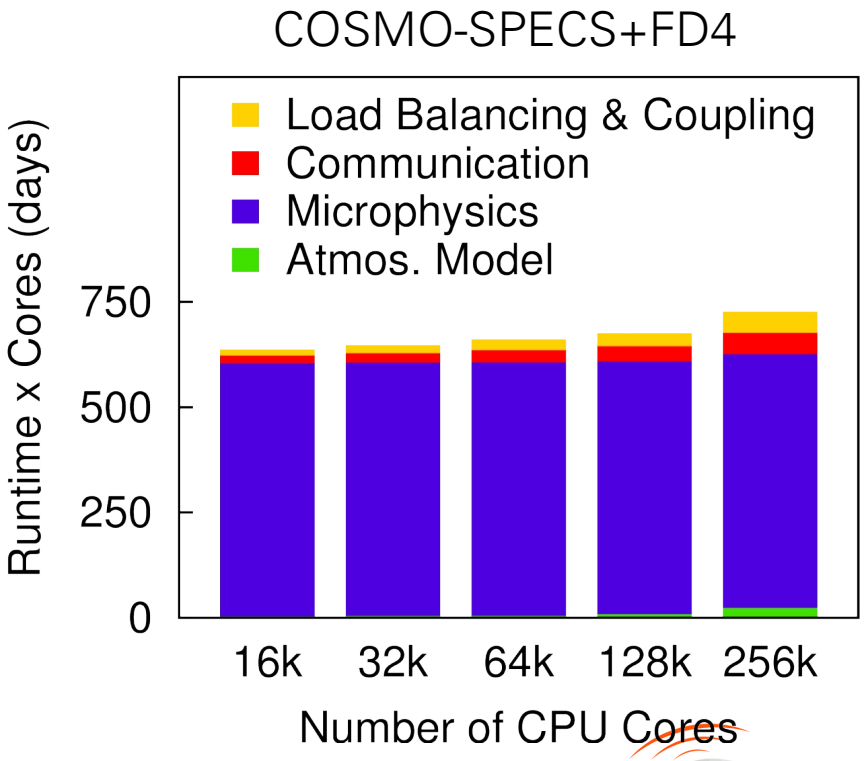
Dynamic Load Balancing

Optimized Data Structures

Scalability of COSMO-SPECS with FD4

- Grid size: 1024 x 1024 x 48 grid cells, > 3M blocks
- 256k: 30 min forecast in <5min (w/o init and I/O)
- Runs on Blue Gene/Q with up to 262 144 MPI ranks
- 14x speed-up from 16k to 256k

Lieber, Nagel, Mix, *Scalability Tuning of the Load Balancing and Coupling Framework FD4*, NIC Symposium 2014, pp. 363-370.

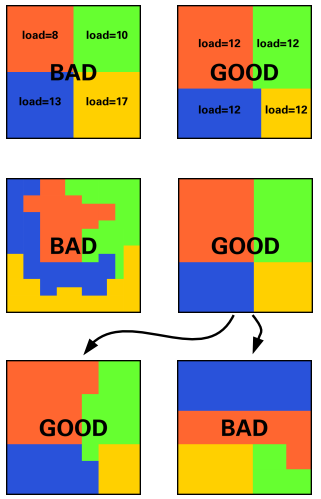


Outline

- Motivation: Why FD4?
- FD4 Basic Concept
- **FD4 Features**
  - Dynamic Load Balancing
  - Model Coupling
  - 4th Dimension
  - Adaptive Block Allocation
- FD4 Usage
- Outlook to Hands-on

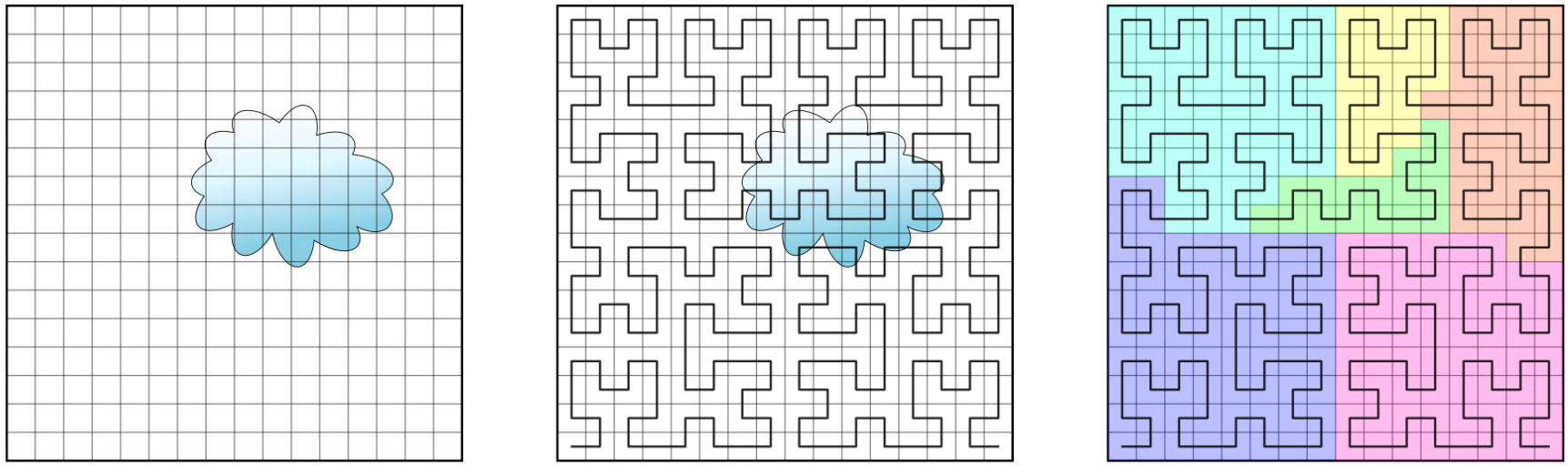
Dynamic Load Balancing

- Four objectives of dynamic load balancing
  - Balance workload
  - Reduce communication between partitions (due to data dependencies)
  - Reduce migration, i.e. communication when changing the partitioning
  - Compute partitioning as fast as possible
- Contradictory goals
- Existing methods (heuristics) provide different trade-offs between the four objectives
  - Bisection methods, space-filling curves, graph methods, diffusion methods, ...



Teresco, Devine, Flaherty, *Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations*, LNCSE, vol. 51, pp. 55-88, 2006.

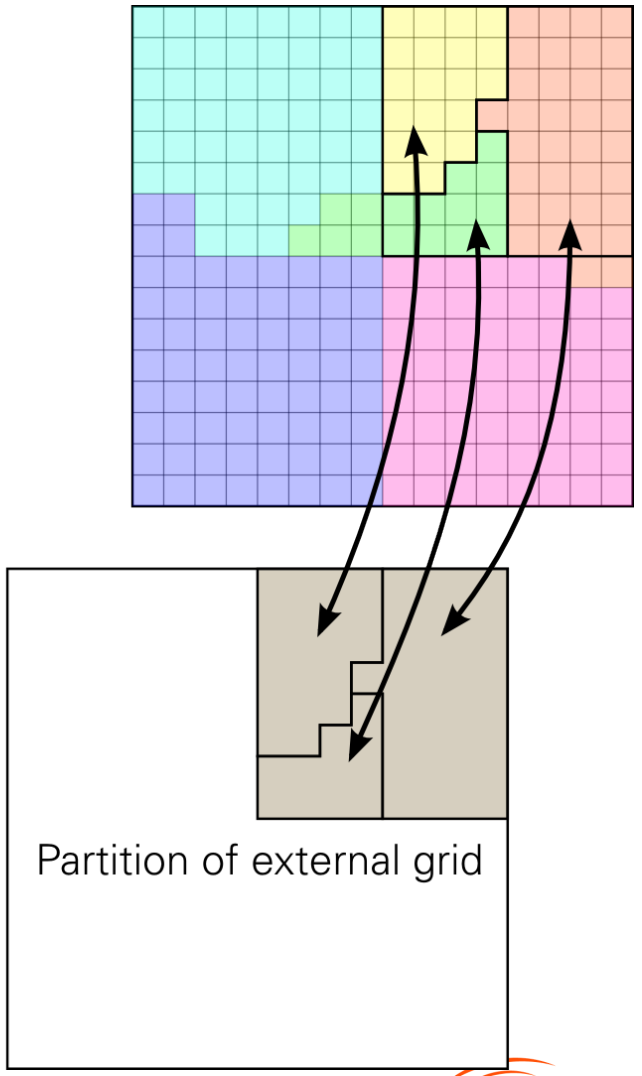
FD4 Features: Dynamic Load Balancing



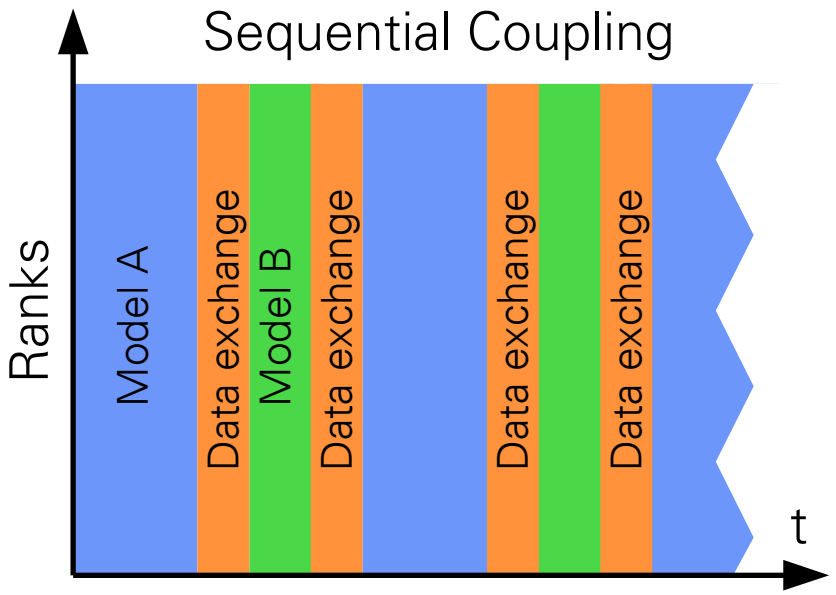
- 3D block decomposition of rectangular grid
- Each block has a computational weight
- Goal: Reduce the maximum workload assigned to a single rank
- Space-filling curve (SFC) partitioning well supported in FD4
- Alternative partitioning methods in FD4: Recursive bisection, graph partitioning using ParMetis

FD4 Features: Model Coupling

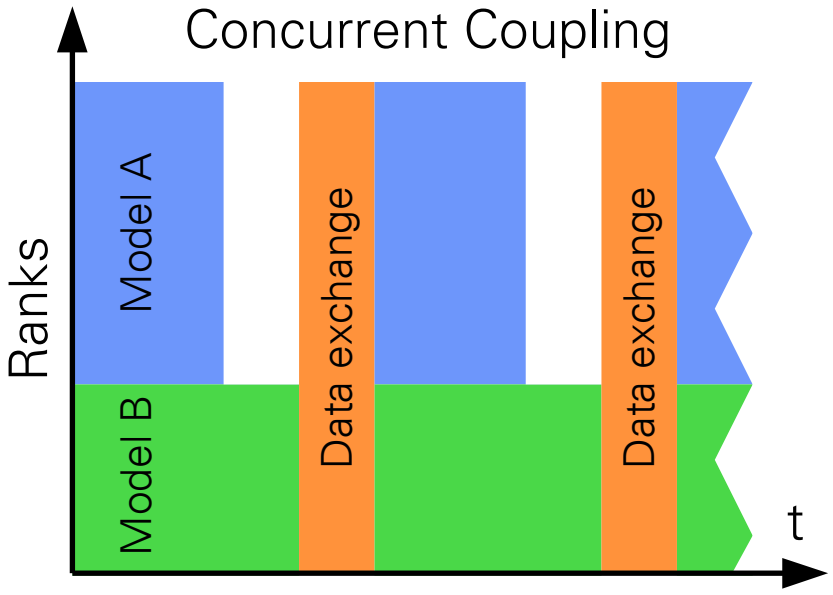
- Data exchange between FD4 based model and an external model
  - E.g. weather or CFD model
  - Transfer in both directions
- FD4 computes partition overlaps after each repartitioning of FD4 grid
  - Highly scalable algorithm
- No grid transformation / interpolation
  - External model must provide data matching the FD4 grid
- “Sequential” coupling only
  - Both models run alternately on same set of MPI ranks



Sequential vs. Concurrent Model Coupling



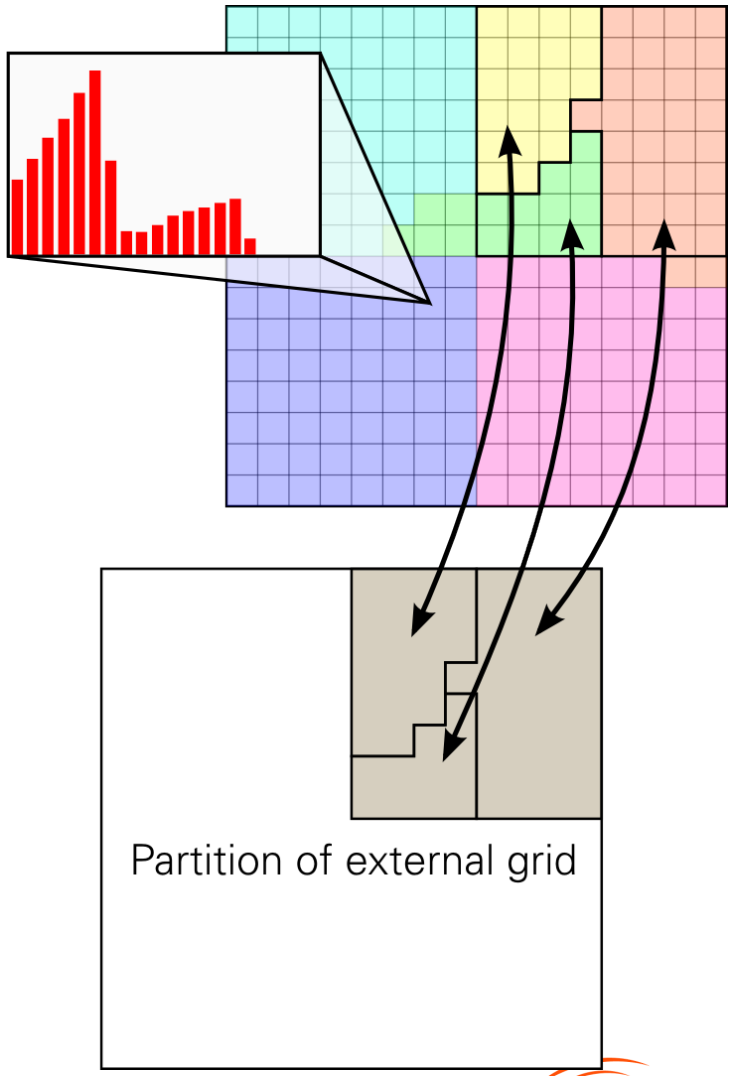
- Both models run alternately on same set of MPI ranks
- Allows tight coupling (data dependencies)
- Avoids load imbalances between models



- MPI ranks are split into groups
- Loose coupling, codes may be separate
- Scales to higher total number of ranks

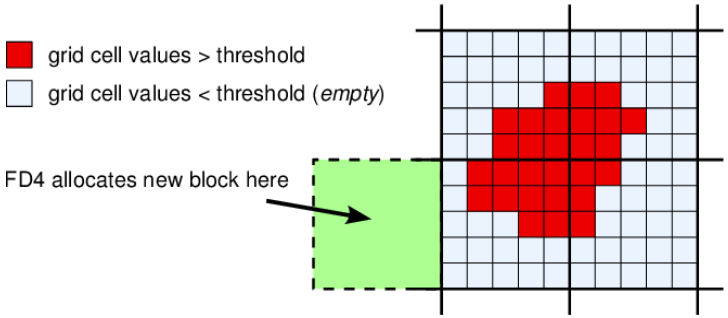
FD4 Features: 4th Dimension

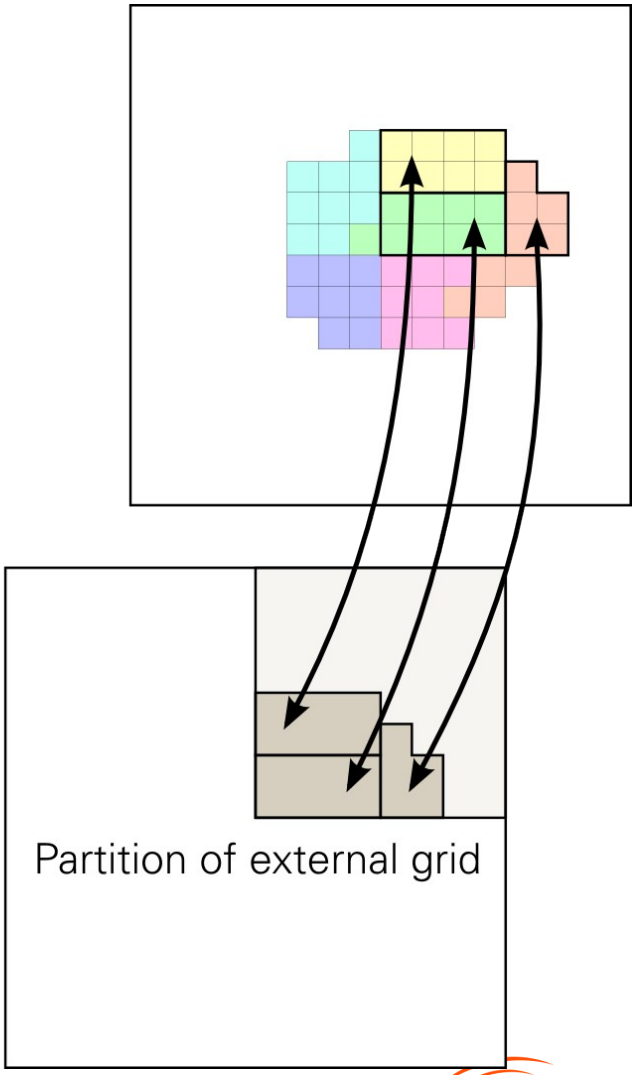
- Extra, non-spatial dimension of grid variables, e.g.
  - Size resolving models
  - Array of gas phase tracers
- COSMO-SPECS requires
  2 x 11 x 66 ~ 1500 values
- FD4 is optimized for a large 4th dimension



FD4 Features: Adaptive Block Allocation

- Grid allocation adapts to spatial structure of simulated problem
  - Save memory in case data and computations are required for a subset only
- For multiphase problems like drops, clouds, flame fronts
- FD4 ensures existence of all blocks required for correct stencil operations





Outline

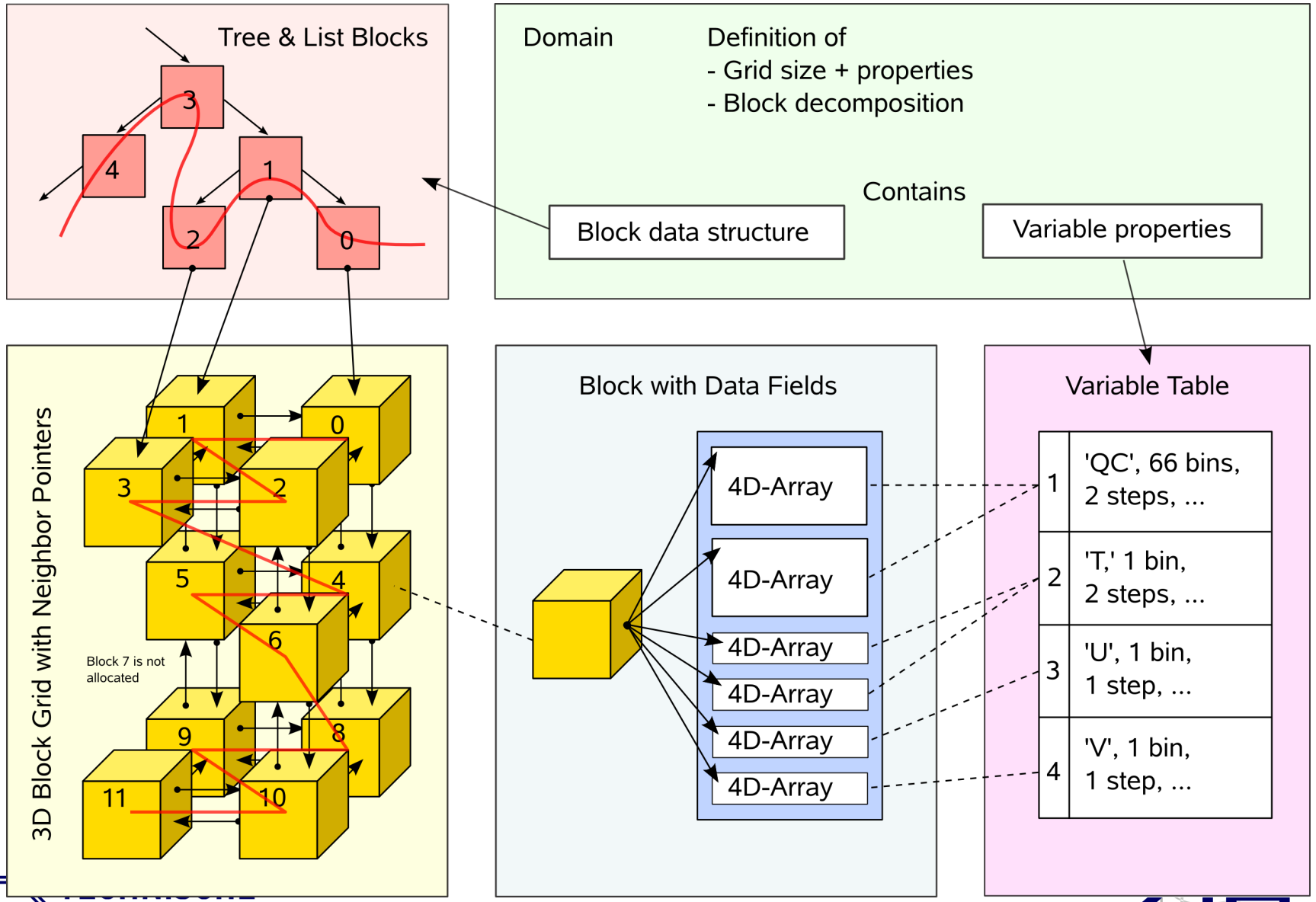
- Motivation: Why FD4?
- FD4 Basic Concept
- FD4 Features
- **FD4 Usage**
- Outlook to Hands-on

FD4 Usage: Requirements

- 3D regular Cartesian grid without refinement
- Stencil computations, i.e. data dependencies only between neighbor grid cells
- Implementation in Fortran 95 and MPI

- Typical for (regional) atmospheric models

FD4 Usage: Basic Data Structures



FD4 Usage: Integer and Real Kind Types

- FD4 defines integer and real kind types

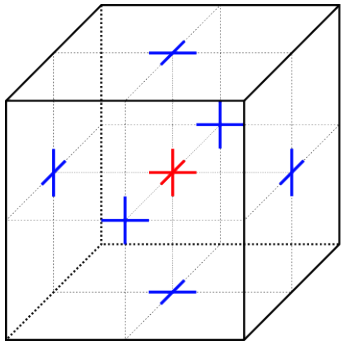
```
integer, parameter :: i4k = selected_int_kind(9)    !! 4 byte integer
integer, parameter :: i8k = selected_int_kind(18)   !! 8 byte integer
integer, parameter :: i_k = i4k                     !! default integer kind

integer, parameter :: r4k = selected_real_kind(6)   !! 4 byte real (single precision)
integer, parameter :: r8k = selected_real_kind(12)  !! 8 byte real (double precision)
integer, parameter :: r_k = r8k                     !! default real kind
```

- Default real type r_k for grid variables is double precision
  - Can be changed to single precision at compile time
- Use the default real type for arrays/pointers that handle grid variables

```
real(r_k), allocatable :: array(:,:,:,:)
```

FD4 Usage: Variable Table (fd4_vartab)

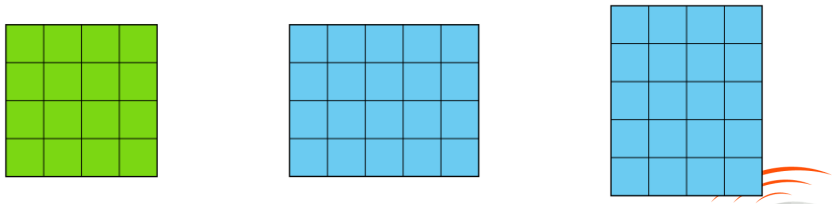
```
program fd4_demo_vartab

  use fd4_mod
  implicit none

  integer, parameter :: varT = 1, varP = 2, varQC = 3, varU = 4, varV = 5
  type(fd4_vartab) :: vartab(5)

  ! Create variable table using the derived type constructor
  ! defaults: nsteps=1, nbins=1, vnull=0.0, facevar=FD4_CELLC
  vartab(varT)   = fd4_vartab('Temperature', nsteps=2, vnull=273.15 )
  vartab(varP)   = fd4_vartab(   'Pressure', nsteps=2 )
  vartab(varQC)  = fd4_vartab(   'Droplets', nbins=100, nsteps=2)
  vartab(varU)   = fd4_vartab(     'u Wind', facevar=FD4_FACEX )
  vartab(varV)   = fd4_vartab(     'v Wind', facevar=FD4_FACEY )

  write(*,'(5(A24,I4,I4,L3,E11.3,E11.3,I3,/))') vartab

end program fd4_demo_vartab
```



cell-centered
face-centered

4 x 4 sized 2D block

cell-centered x face-centered y face-centered



FD4 Usage: Domain (fd4_domain)

```
program fd4_demo_domain

  use fd4_mod
  implicit none
  include 'mpif.h'

  type(fd4_domain), target :: domain
  [...]

  ! Create variable table
  [...]

  call MPI_Init(err)

  ! Create the FD4 domain - this does not allocate any blocks
  dsize(1:3,1) = (/  1,  1,  1/)    ! grid start indices
  dsize(1:3,2) = (/ 16, 16,  8/)    ! grid end indices
  bnum(1:3) =    (/  4,  4,  2/)    ! number of blocks in each dimension
  nghosts(1:3) = (/  2,  2,  2/)    ! number of ghost cells in each dimension
  peri(1:3) = .true.                ! periodic boundaries
  call fd4_domain_create(domain, bnum, dsize, vartab, nghosts, peri, MPI_COMM_WORLD, err)

  ! Allocate all blocks (distributed over all ranks)
  call fd4_util_allocate_all_blocks(domain, err)

  ! Delete the domain and finalize MPI
  call fd4_domain_delete(domain)
  call MPI_Finalize(err)

end program fd4_demo_domain
```

FD4 Usage: Block Iterator (fd4_iter)

```
type(fd4_iter) :: iter

[...]

! Initialize block iterator
call fd4_iter_init(domain, iter)
! iter%cur is a pointer to the current block
do while(associated(iter%cur))

  ! iter%cur%pos is the 3D position of the block in the block grid
  write(*,'(A,I4,A,3(I3))') 'rank ',rank,' iterates to block at (x, y, z) ',iter%cur%pos

  ! get offset from global domain indexes to block-local indexes (starting at 1)
  call fd4_iter_offset(iter, offset)

  ! loop over block's grid cells, iter%cur%ext is the block size
  do z=1,iter%cur%ext(3)
    do y=1,iter%cur%ext(2)
      do x=1,iter%cur%ext(1)
        write(*,'(A,I4,A,3(I3))') 'rank ',rank,' iterates to cell at (x, y, z) ', &
                                  offset(1)+x, offset(2)+y, offset(3)+z
      end do
    end do
  end do

  ! go to next block
  ! sets iter%cur to NULL if iteration is finished
  call fd4_iter_next(iter)

end do
```

FD4 Usage: Accessing Variables

- Variables are allocated in one continuous chunk of memory
  - For each discretization type (cell-centered, face{x,y,z})
- To access individual variables, use pointers to subarrays

```
iter%cur%fields(variable_index, time_level)%l(bin, x, y, z)
```

```
call fd4_iter_init(domain, iter)
do while(associated(iter%cur))
  call fd4_iter_offset(iter, offset)
  do z=1,iter%cur%ext(3)
    do y=1,iter%cur%ext(2)
      do x=1,iter%cur%ext(1)

        ! get global z coordinate of this grid cell
        gz = offset(3) + z
        ! set temperature depending on global z coordinate
        iter%cur%fields(varT, 1)%l(1, x, y, z) = 295.0 + f * REAL(gz)

      end do
    end do
  end do
  call fd4_iter_next(iter)
end do
```

FD4 Usage: Accessing Variables – Pointer Version

- Use a pointer for convenience

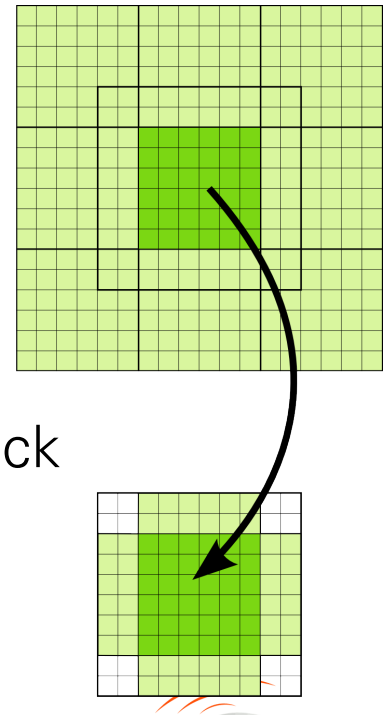
```
real(r_k), pointer :: t(:,:,:)
[...]

call fd4_iter_init(domain, iter)
do while(associated(iter%cur))
  call fd4_iter_offset(iter, offset)

  ! set pointer to variable t of current block
  t =>  iter%cur%fields(varT, 1)%l(1, :, :, :)

  do z=1,iter%cur%ext(3)
    do y=1,iter%cur%ext(2)
      do x=1,iter%cur%ext(1)

        ! get global z coordinate of this grid cell
        gz = offset(3) + z
        ! set temperature depending on global z coordinate
        t(x, y, z) = 295.0 + f * REAL(gz)

      end do
    end do
  end do
  call fd4_iter_next(iter)
end do
```

FD4 Usage: Accessing Ghost Cells

- Access to ghost cells is not directly possible
- This subroutine copies a variable of a block including ghost cells into a buffer array:

```
call fd4_iter_get_ghost(iter, var_index, time_level, buf_ext, buf)
```

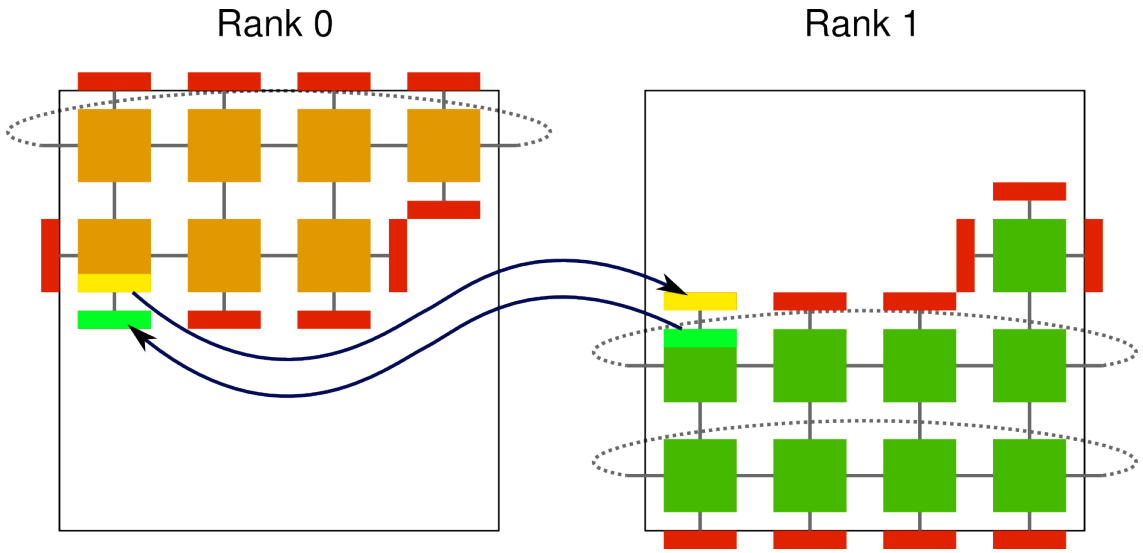
- Cell-centered variables only
- No data from diagonal neighbor blocks is copied
- Operation depends on neighbor block location
  - Present locally: Copied from neighbor block
  - Present on remote rank: Copied from ghost block
  - Not present (adaptive block mode):
    Filled with vnull



FD4 Usage: Accessing Ghost Cells

```
real(r_k), allocatable :: buf(:,:,:,:)

! Allocate the buffer array for a single block with ghost cells
call fd4_domain_max_bext(domain, bext(1:3), .true.)
bext(0) = 1 ! 4th dimension not used here
allocate( buf(bext(0),0:bext(1)-1,0:bext(2)-1,0:bext(3)-1) )

! iterate over all local blocks
call fd4_iter_init(domain, iter)
do while(associated(iter%cur))

  ! get temperature with ghost cells from current block
  call fd4_iter_get_ghost(iter, varT, now, bext, buf)

  ! loop over block's grid cells
  do z=1,iter%cur%ext(3)
    do y=1,iter%cur%ext(2)
      do x=1,iter%cur%ext(1)
        ! compute time step
        delta = ( buf(1,x-1,y,z) + buf(1,x+1,y,z) - 2*buf(1,x,y,z) ) &
              + ( buf(1,x,y-1,z) + buf(1,x,y+1,z) - 2*buf(1,x,y,z) ) &
              + ( buf(1,x,y,z-1) + buf(1,x,y,z+1) - 2*buf(1,x,y,z) )
        ! set updated temperature value
        iter%cur%fields(varT,new)%l(1,x,y,z) = buf(1,x,y,z) + delta * dt
      end do
    end do
  end do

  call fd4_iter_next(iter)
end do
```

FD4 Usage: Ghost Communicator (fd4_ghostcomm)

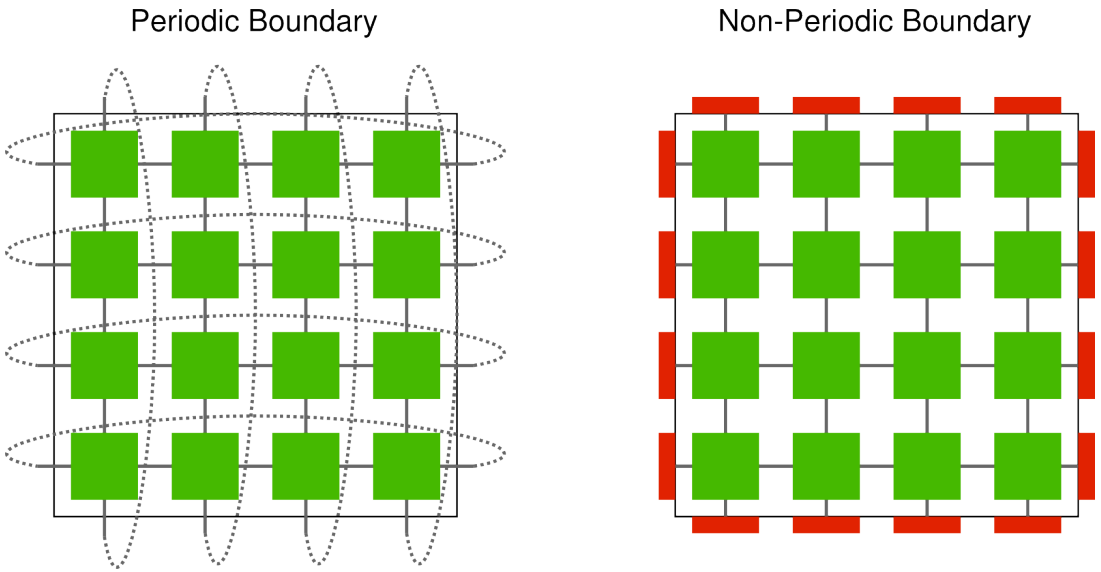


- Ghost communicators fills all ghost blocks with data from remote blocks
- Multiple variables and time levels can be defined in constructor
- Cell-centered variables only
- Aggregates all data between each pair of MPI ranks in one MPI message

FD4 Usage: Ghost Communicator (fd4_ghostcomm)

```
type(fd4_ghostcomm) :: ghostcomm(2)
[...]

! Create ghost communicator for variable varT (one for each time level)
call fd4_ghostcomm_create(ghostcomm(1), domain, 1, (/varT/), (/1/), err)
call fd4_ghostcomm_create(ghostcomm(2), domain, 1, (/varT/), (/2/), err)

do step=1, nsteps   ! Time stepping loop

  call fd4_ghostcomm_exch(ghostcomm(now), err)  ! exchange ghost cells for time level 'now'

  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))
    call fd4_iter_get_ghost(iter, varT, now, bext, buf)
    do z=1,iter%cur%ext(3)
      do y=1,iter%cur%ext(2)
        do x=1,iter%cur%ext(1)
          delta = ...
          iter%cur%fields(varT,new)%l(1,x,y,z) = buf(1,x,y,z) + delta * dt
        end do
      end do
    end do
    call fd4_iter_next(iter)
  end do

  now = 3 - now   ! swap time level indicators
  new = 3 - new

end do

! Delete the ghost communicators
call fd4_ghostcomm_delete(ghostcomm(1))
call fd4_ghostcomm_delete(ghostcomm(2))
```

FD4 Usage: Boundary Conditions (fd4_boundary)



- Periodic boundaries simply create a torus from the mesh
  - Using neighbor block pointers or ghost blocks
- Non-periodic boundaries are realized with *boundary* ghost blocks
  - User needs to fill boundary ghost blocks
  - FD4 can automatically set zero-gradient boundary conditions

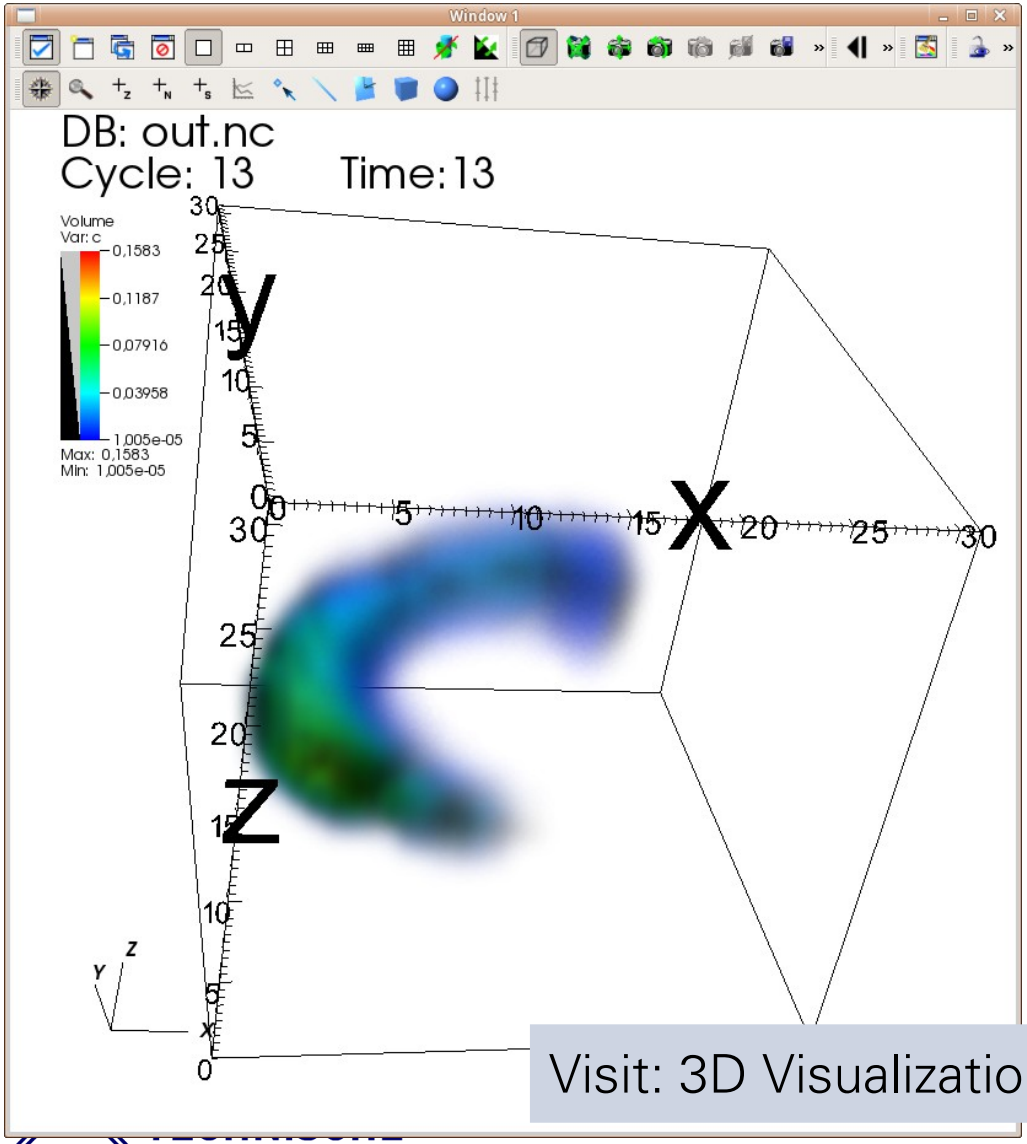
FD4 Usage: Boundary Conditions (fd4_boundary)

```
peri(1:3) = (/.true., .true., .false./) ! no periodic boundaries in z direction
call fd4_domain_create(domain, bnum, dsize, vartab, nghosts, peri, MPI_COMM_WORLD, err)
[...]

! set (fixed) boundary conditions for lower z
call fd4_boundary_spec(domain, (/varT,varT/), (/now,new/), FACE_Z, 1, (/0.0_r_k/))

do step=1, nsteps   ! Time stepping loop

  call fd4_ghostcomm_exch(ghostcomm(now), err)

  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))

    ! set zero-gradient boundary conditions for upper z for this block
    call fd4_boundary_zerograd_block(domain, iter%cur, (/varT/), (/now/), FD4_Z, opt_dir=2)

    call fd4_iter_get_ghost(iter, varT, now, bext, buf)
    do z=1,iter%cur%ext(3)
      do y=1,iter%cur%ext(2)
        do x=1,iter%cur%ext(1)
          delta = ...
          iter%cur%fields(varT,new)%l(1,x,y,z) = buf(1,x,y,z) + delta * dt
        end do
      end do
    end do
    call fd4_iter_next(iter)
  end do

  now = 3 - now   ! swap time level indicators
  new = 3 - new
end do
```

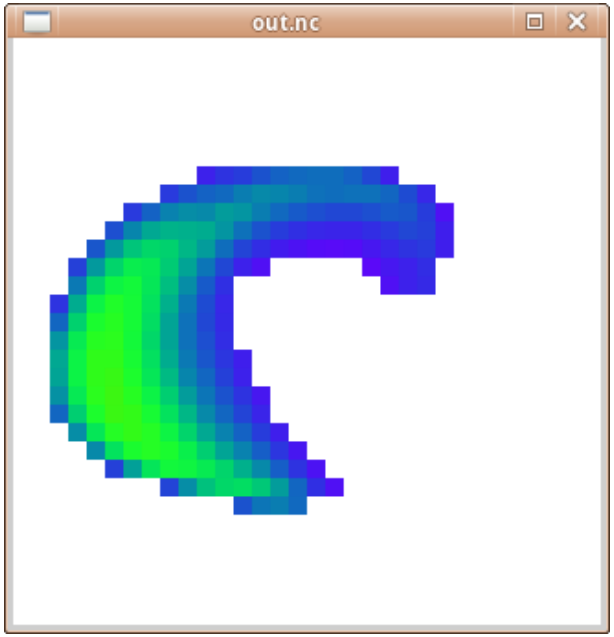
FD4 Usage: NetCDF Output (fd4_netcdf4_comm)

```
type(fd4_netcdf4_comm) :: nfcomm

[...]

! Initialize output
call fd4_netcdf4_open(nfcomm, domain, 'out.nc', 3, (/varT, varU, varV/), (/1, 1, 1/), err)

! Write initial data
call fd4_netcdf4_write(nfcomm, err)

do step=1, nsteps   ! Time stepping loop

  call fd4_ghostcomm_exch(ghostcomm(now), err)

  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))
    ! loop over grid cells of this block and update temperature
    [...]
    call fd4_iter_next(iter)
  end do

  ! Append current data to the NetCDF file, select time level 'new' of varT
  if(mod(step,100)==0) then
    if(rank==0) write(*,'(A,I5)') 'writing output at time step ',step
    call fd4_netcdf4_write(nfcomm, err, st_opt=(/new, 1, 1/))
  end if

  now = 3 - now   ! swap time level indicators
  new = 3 - new
end do

! Close output file
call fd4_netcdf4_close(nfcomm, err)
```

FD4 Usage: NetCDF Output (fd4_netcdf4_comm)



Visit: 3D Visualization



ncview: 2D Visualization

FD4 Usage: Dynamic Load Balancing

```
do step=1, nsteps   ! Time stepping loop

  call fd4_ghostcomm_exch(ghostcomm(now), err)

  call fd4_iter_init(domain, iter)
  do while(associated(iter%cur))

    ! time the start of computations on current block
    call fd4_iter_start_clock(iter)

    ! loop over grid cells of this block and update temperature
    [...]

    ! time the end of computations on current block, set block weight
    call fd4_iter_stop_clock(iter)

    call fd4_iter_next(iter)
  end do

  ! rebalance the workload based on the measured block weights
  call fd4_balance_readjust(domain, err)

  now = 3 - now   ! swap time level indicators
  new = 3 - new
end do
```

FD4 Usage: Dynamic Load Balancing + Stats

```
! FD4 load balancing statistics
type(fd4_balance_statistics) :: stats

do step=1, nsteps   ! Time stepping loop

  call fd4_ghostcomm_exch(ghostcomm(now), err)

  ! loop over blocks, computation, etc.
  [...]

  ! rebalance the workload based on the measured block weights
  call fd4_balance_readjust(domain, err, opt_stats=stats)

  if(rank==0) then
    write(*,'(A,A)')    'message from load balancing: ',stats%status
    write(*,'(A,F6.3)') 'load balance was: ',stats%last_measured_balance
    write(*,'(A,L2)')   'calculated new partitioning: ',stats%partition_changed
    write(*,'(A,I6)')   'migrated blocks: ',stats%migrated_blocks
  end if

  now = 3 - now   ! swap time level indicators
  new = 3 - new
end do
```

```
message from load balancing: not balanced, migrated blocks
load balance was:  0.898
calculated new partitioning:  T
migrated blocks:      7
```

FD4 Usage: Dynamic Load Balancing Parameters

```
call fd4_domain_create(domain, bnum, dsize, vartab, nghosts, peri, MPI_COMM_WORLD, err)

! Set some load balancing parameters
! compute new partitioning only if balance is below 0.7
call fd4_balance_params (domain, opt_lbtol=0.7)

! Use Morton space-filling curve instead of Hilbert space-filling curve
call fd4_balance_params (domain, opt_sfctype=FD4_PART_SFC_MORTON)

! Use recursive coordinate bisection instead of space-filling curve
call fd4_balance_params (domain, method=FD4_BALANCE_RCB)

! Use auto mode: FD4 decides whether load balancing is beneficial or not
! (overrides opt_lbtol). Decision is based on measuring the benefit of improved
! load balance vs. the overhead of repartitioning and block migration.
call fd4_balance_params (domain, opt_auto=.true.)

! Allocate all blocks (distributed over all ranks).
! Load balancing parameters matter already here!
call fd4_util_allocate_all_blocks(domain, err)

[...]

do step=1, nsteps   ! Time stepping loop

  [...]

end do
```

FD4 Usage: Coupling

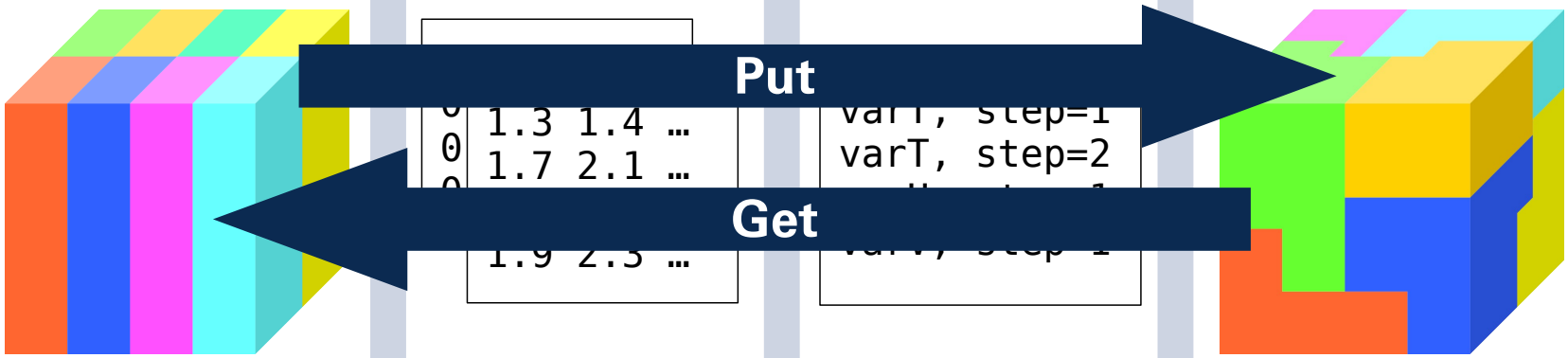
FD4 Couple Context

Position, size & owner rank of all coupled partitions

Pointers to local arrays containing coupled data

List of FD4 variables participating in coupling

Pointer to FD4 domain



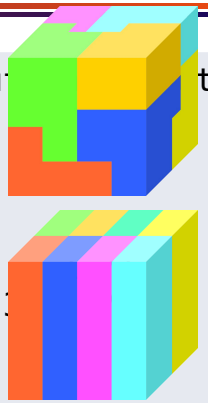
Internal data structures, MPI datatypes, ...

FD4 Usage: Coupling

```
! Create couple context for specified domain, optionally promise to use put only, not get
call fd4_couple_create(coupler, domain, err, opt_cpldir=FD4_CPL_PUT)

! add one partition per rank, simple 1D partitioning of 3D grid
do irank = 0, nproc-1
  arrayBounds(1:3,1) = (/   1+(irank*dsize(1,2))/nproc,          1,          1 /)
  arrayBounds(1:3,2) = (/ ((irank+1)*dsize(1,2))/nproc, dsize(2,2), dsize(3,2) /)
  call fd4_couple_add_partition(coupler, irank, arrayBounds, err)
end do

! Add two variables varT and varP to the couple context.
! FD4 returns the identifiers cplArrayIdxT and cplArrayIdxP for the local couple array.
call fd4_couple_add_var(coupler, varT, 1, err, cplArrayIdxT)
call fd4_couple_add_var(coupler, varP, 1, err, cplArrayIdxP)

! add local couple arrays to the couple context
arraypointer => t(:,:,:)  ! fd4_couple_set_local_3D_array only accepts pointers
call fd4_couple_set_local_3D_array(coupler, cplArrayIdxT, arraypointer, (/1,1,1/), err)
arraypointer => p(:,:,:)
call fd4_couple_set_local_3D_array(coupler, cplArrayIdxP, arraypointer, (/1,1,1/), err)

! commit the couple context, FD4 now checks the couple arrays and prepares MPI data types
call fd4_couple_commit(coupler, err)

! finally put data from arrays t and p to FD4's data structures
call fd4_couple_put(coupler, err)

! delete couple context
call fd4_couple_delete(coupler, err)
```

FD4 Usage: Coupling

```
! Create couple context for specified domain, optionally promise to use put
call fd4_couple_create(coupler, domain, err, opt_cpldir=FD4_CPL_PUT)

! add one partition per rank, simple 1D partitioning of 3D grid
do irank = 0, nproc-1
  arrayBounds(1:3,1) = (/   1+(irank*dsize(1,2))/nproc,          1,
  arrayBounds(1:3,2) = (/ ((irank+1)*dsize(1,2))/nproc, dsize(2,2), dsize(
  call fd4_couple_add_partition(coupler, irank, arrayBounds, err)
end do

! Add two variables varT and varP to the couple context.
! FD4 returns the identifiers cplArrayIdxT and cplArrayIdxP for the local
call fd4_couple_add_var(coupler, varT, 1, err, cplArrayIdxT)
call fd4_couple_add_var(coupler, varP, 1, err, cplArrayIdxP)

! add local couple arrays to the couple context
arraypointer => t(:,:,:)  ! fd4_couple_set_local_3D_array only accepts poi
call fd4_couple_set_local_3D_array(coupler, cplArrayIdxT, arraypointer, (/
arraypointer => p(:,:,:)
call fd4_couple_set_local_3D_array(coupler, cplArrayIdxP, arraypointer, (/I,

! commit the couple context, FD4 now checks the couple arrays and prepares MPI data types
call fd4_couple_commit(coupler, err)

! finally put data from arrays t and p to FD4's data structures
call fd4_couple_put(coupler, err)

! delete couple context
call fd4_couple_delete(coupler, err)
```



varT, step=1
varT, step=2
varU, step=1
varV, step=1

0.2 0.4 …
1.3 1.4 …
1.7 2.1 …
2.0 2.6 …
1.9 2.3 …

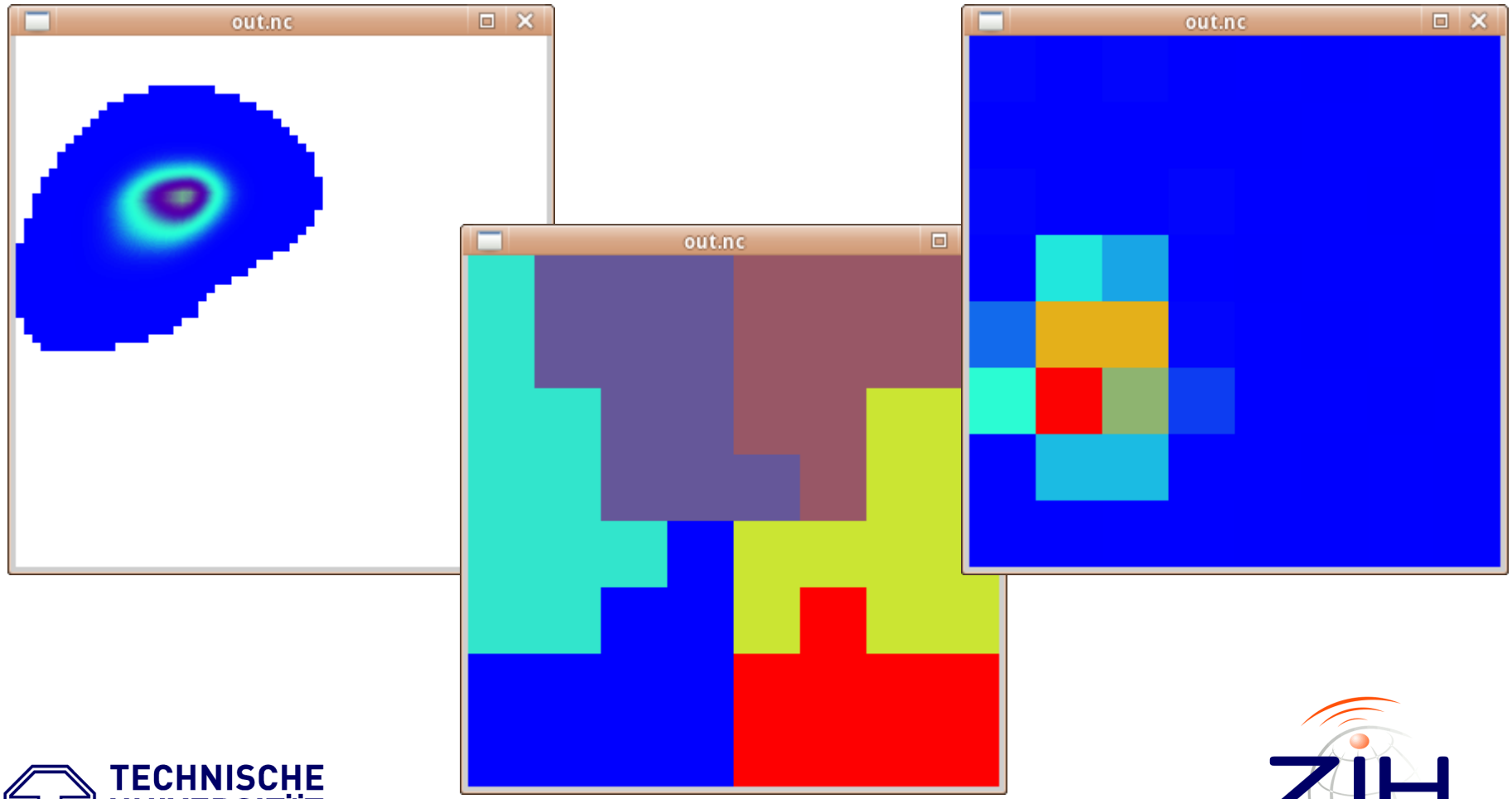
Put

Outline

- Motivation: Why FD4?
- FD4 Basic Concept
- FD4 Features
- FD4 Usage
- **Outlook to Hands-on**

Outlook on FD4 Library Hands-on

- Develop simplistic advection simulation with FD4
- Hands-on Tasks and Material: **tinyurl.com/fd4ictp**



Thank you very much for your attention!

Acknowledgments

Verena Grützun, Ralf Wolke,
Oswald Knoth, Martin Simmel,
René Widera, Matthias Jurenz,
Matthias Müller, Wolfgang E. Nagel

VAMPIR www.vampir.eu

lrz Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities

TROPOS Leibniz Institute for Tropospheric Research
www.tropos.de

DWD COSMO CONSORTIUM FOR SMALL SCALE MODELING
www.cosmo-model.org

PICon GPU HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF
picongpu.hzdr.de

Funding

DFG Deutsche Forschungsgemeinschaft

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Europa fördert Sachsen. ESF Europäischer Sozialfonds

IMDATA

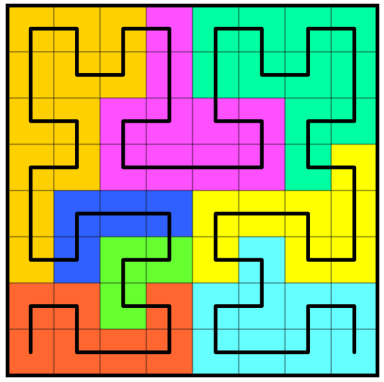
TECHNISCHE UNIVERSITÄT DRESDEN

ZIH Center for Information Services & High Performance Computing

Backup Slides

From SFC Partitioning to 1D Partitioning

- Space-filling curve (SFC) partitioning widely used
  - nD space is mapped to 1D by SFC
  - Mapping is fast and has high locality
  - Migration typically between neighbor ranks
- 1D partitioning is core problem of SFC partitioning
  - Decomposes task chain into consecutive parts
- Two classes of existing 1D partitioning algorithms:
  - Heuristics: fast, parallel, no optimal solution
  - Exact methods: slow, serial, but optimal



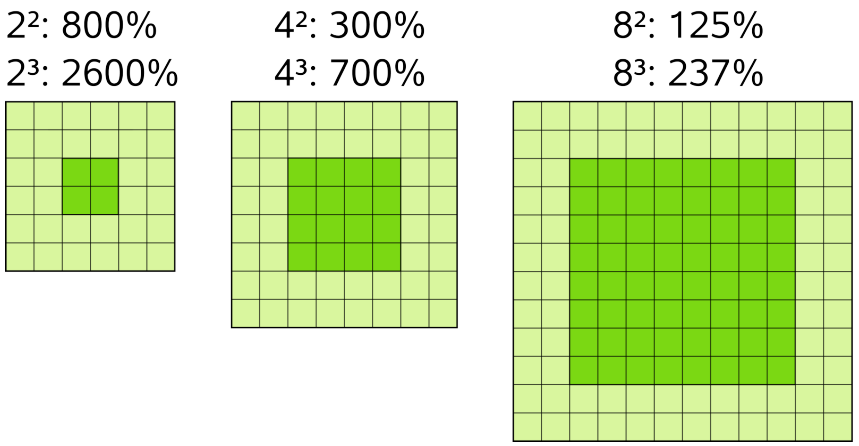
Hilbert SFC

Pilkington, Baden, *Dynamic partitioning of non-uniform structured workloads with spacefilling curves*, IEEE T. Parall. Distr., vol. 7, no. 3, pp. 288-300, 1996.

Pinar, Aykanat, *Fast optimal load balancing algorithms for 1D partitioning*, J. Parallel Distr. Com., vol. 64, no. 8, pp. 974-996, 2004.

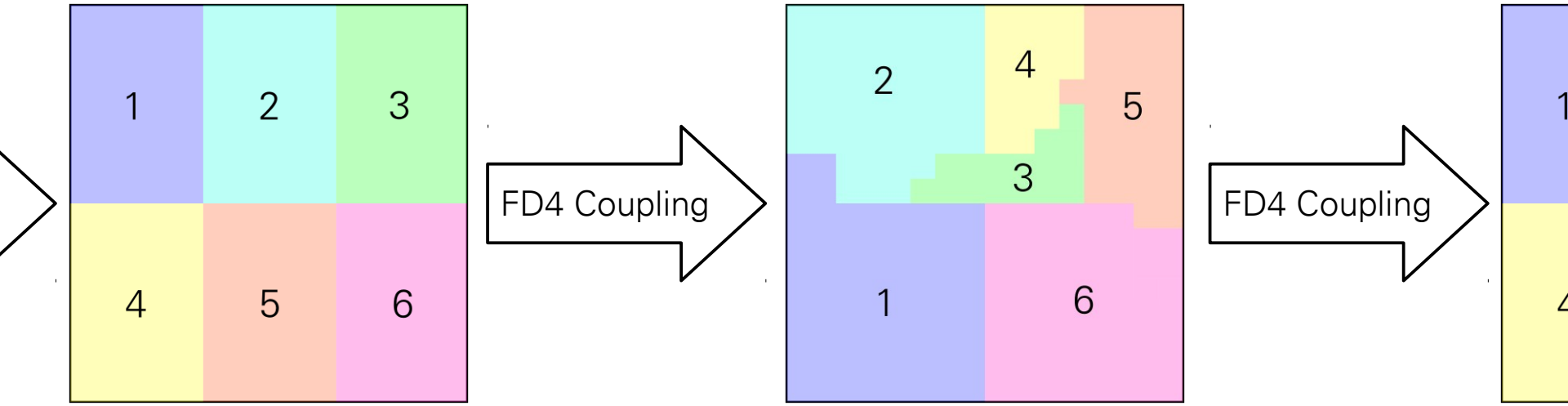
FD4 Features: 4th Dimension → Small Blocks

- A large number of small blocks are good for performance:
  - COSMO-SPECS: ~1500 variables per grid cell: Only small blocks do not exceed processor cache
  - Load balancing:
    #blocks > #ranks to enable fine-grained balancing
- Additional memory costs for a boundary of ghost cells
  - Too high for small blocks!
- Add ghost blocks at the partition borders only



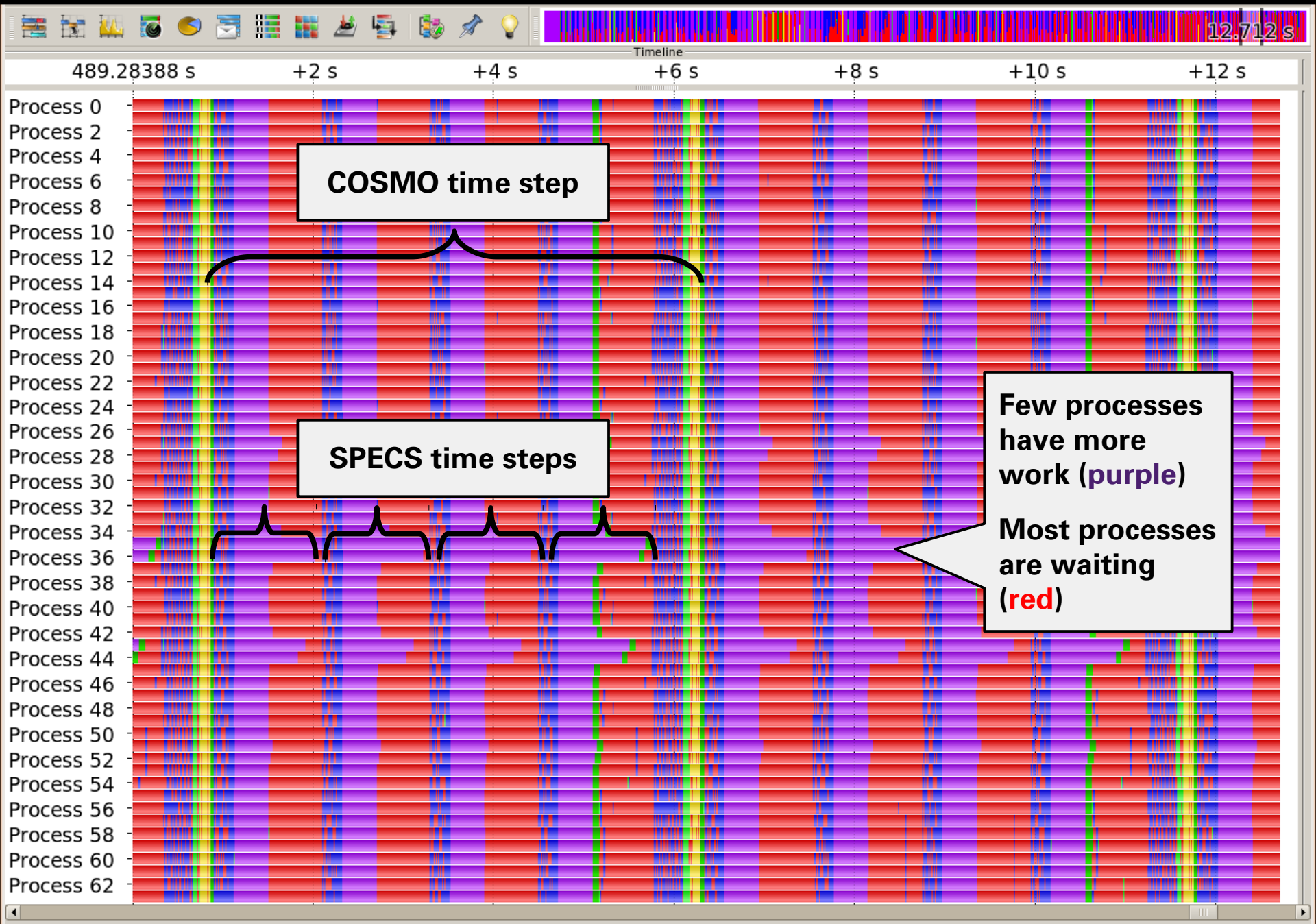
FD4 Basic Concept: COSMO-SPECS+FD4 Coupling Scheme

COSMO-SPECS+FD4 = COSMO-SPECS using FD4 for coupling

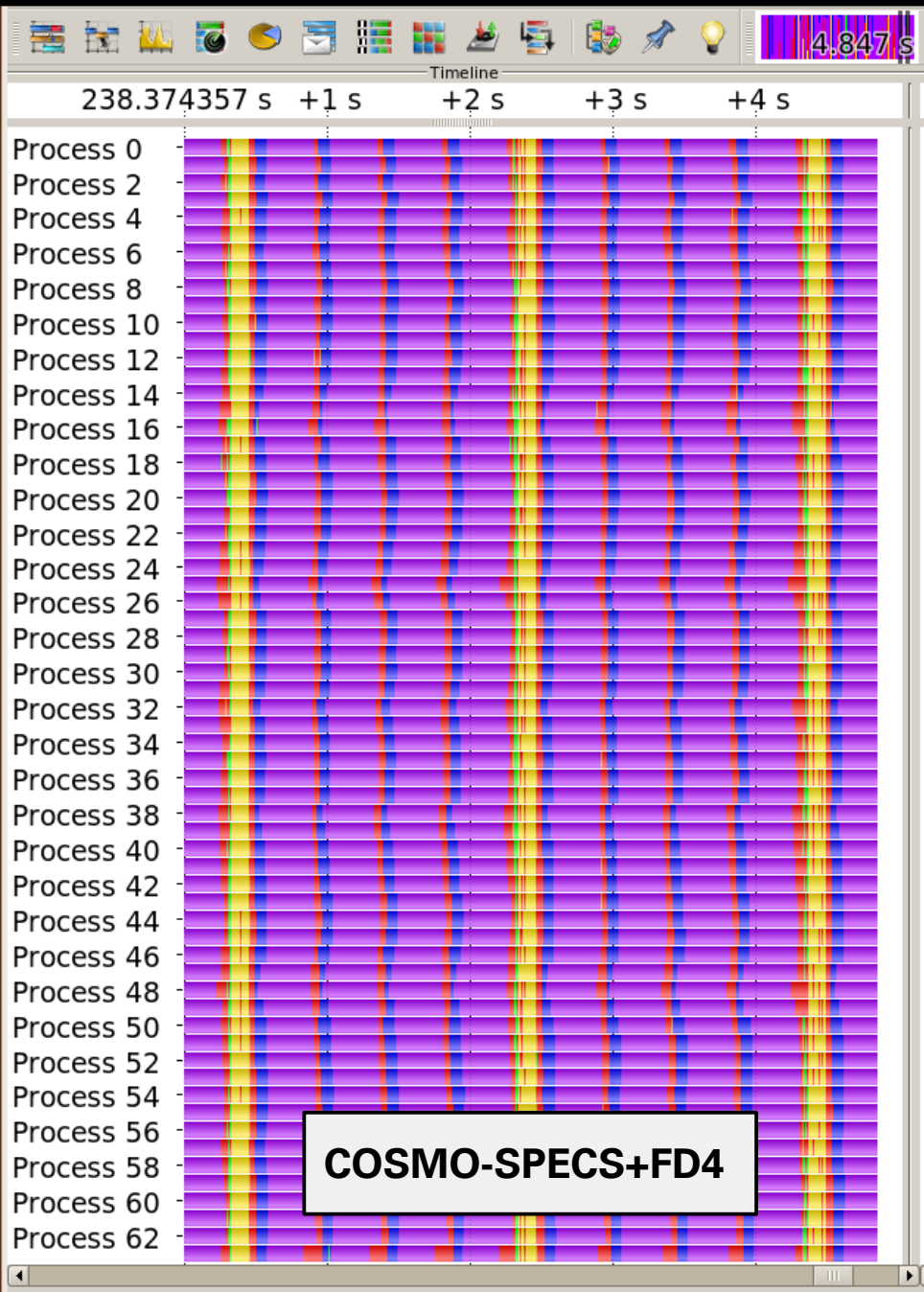


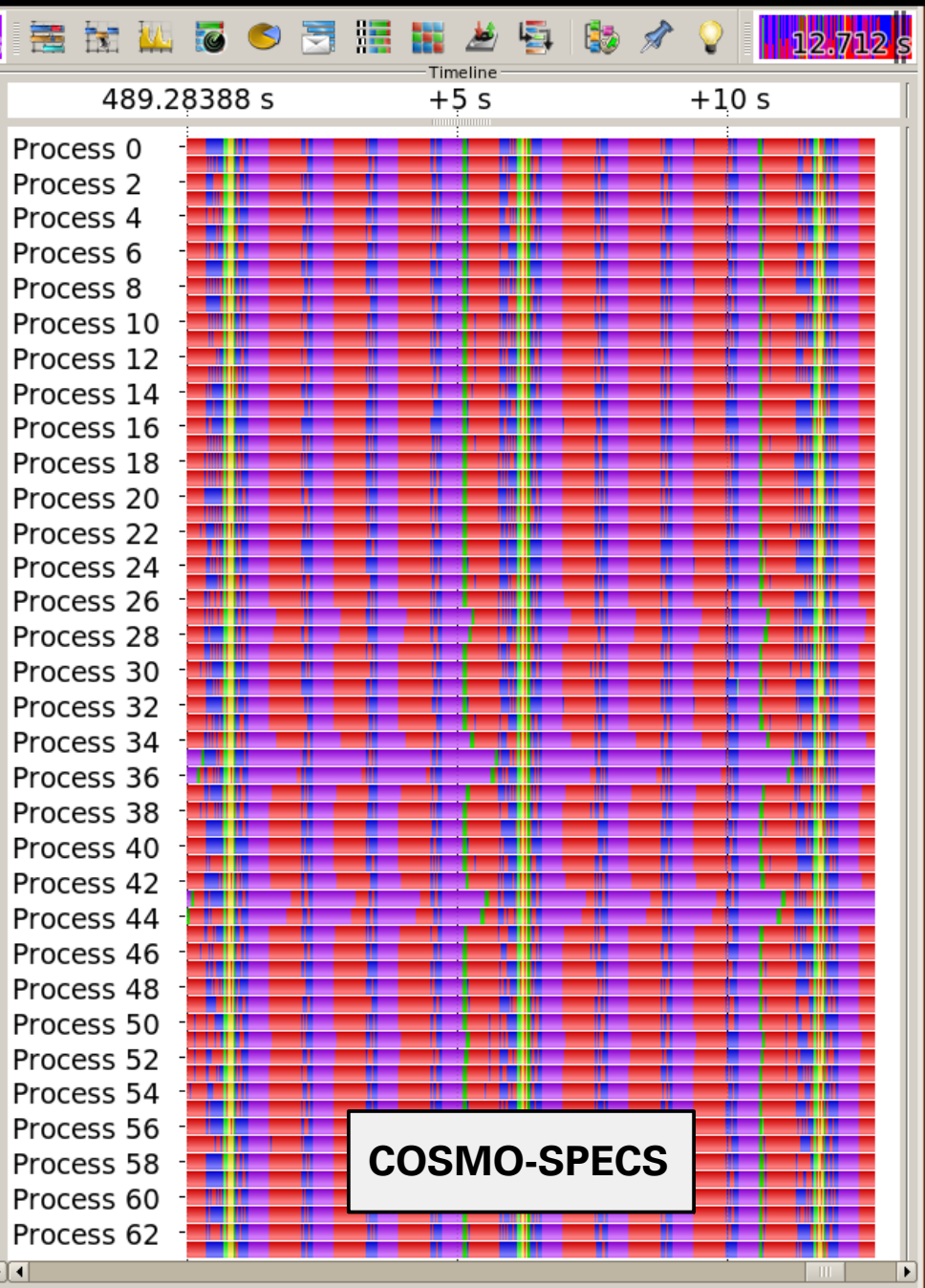
COSMO	**FD4**	**SPECS**	**FD4**
Computes dynamics	Send data to FD4 data structures:	Computes Microphysics	Receive data from FD4 data structures:
Static MxN partitioning	u, v, w, T, p, ρ, q_v	Data dynamically balanced by FD4	ΔT, q_v, q_c, q_i

Analysis: Load Imbalance due to Microphysics



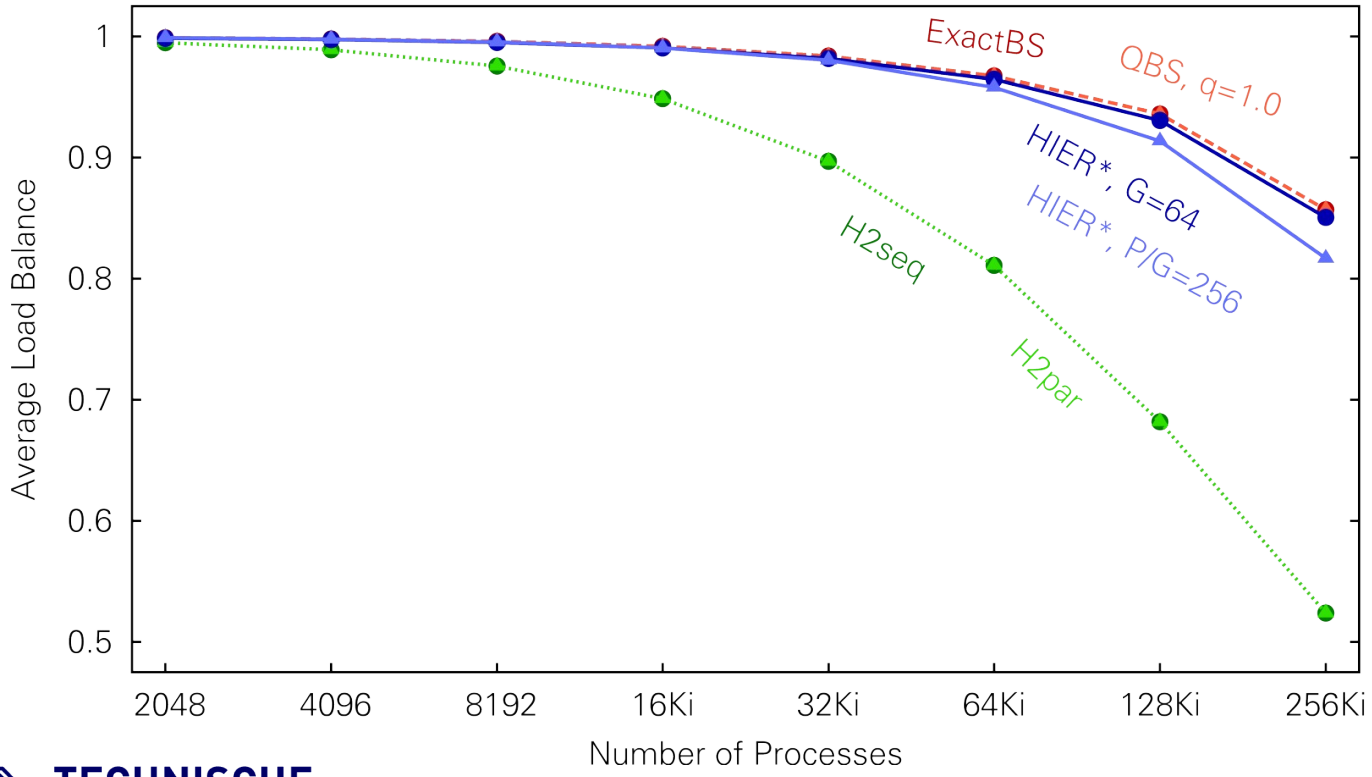
Benchmarks: COSMO-SPECS Performance Comparison





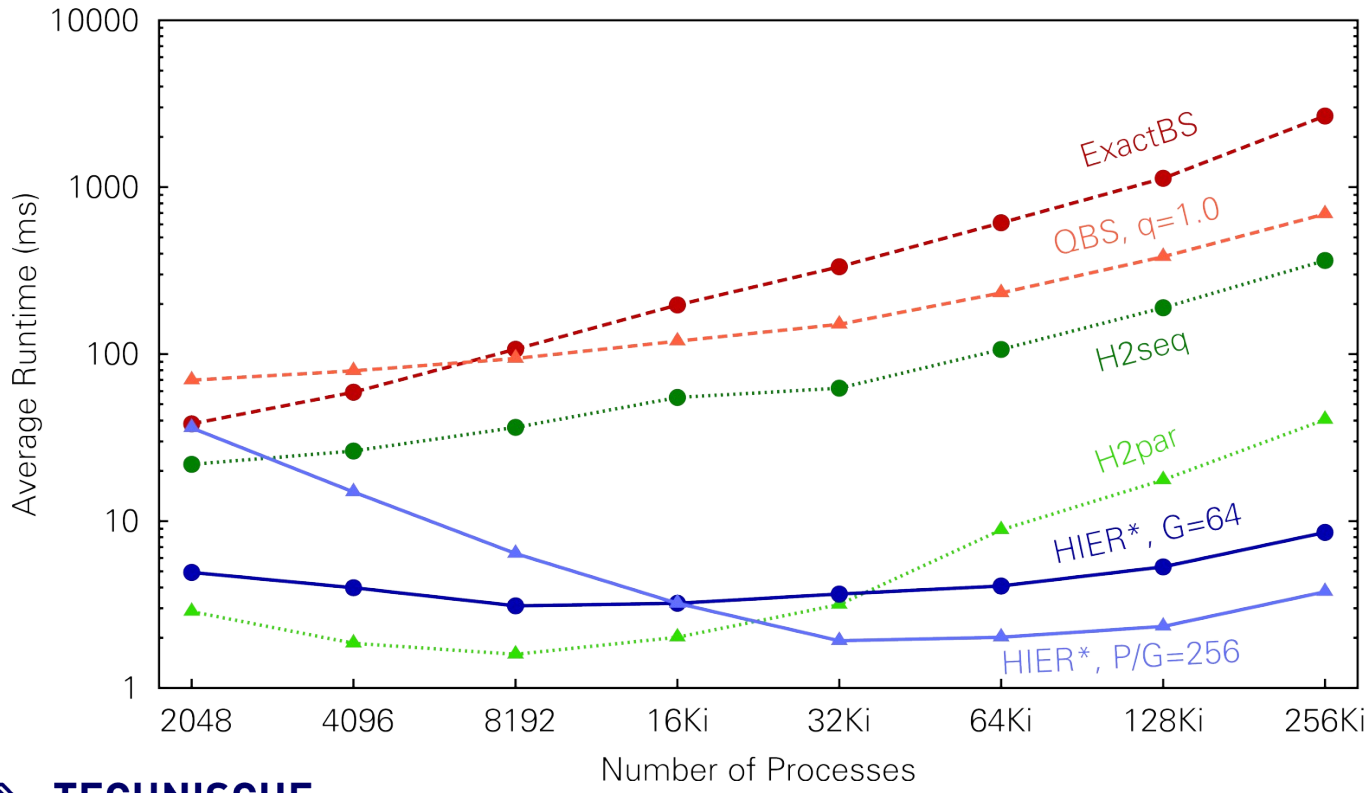
Scalable High-Quality 1D Partitioning: Load Balance

- Cloud simulation, 1 357 824 tasks
- System: JUQUEEN, IBM Blue Gene/Q
- HIER*, G=64 achieves 99.2% of the optimal load balance at 262 144 processes



Scalable High-Quality 1D Partitioning: BG/Q Scalability

- Cloud simulation, 1 357 824 tasks
- System: JUQUEEN, IBM Blue Gene/Q
- HIER*, G=64 runs at 262 144 processes ~300x faster than ExactBS



ExactBS: 2668 ms

QBS: 692 ms

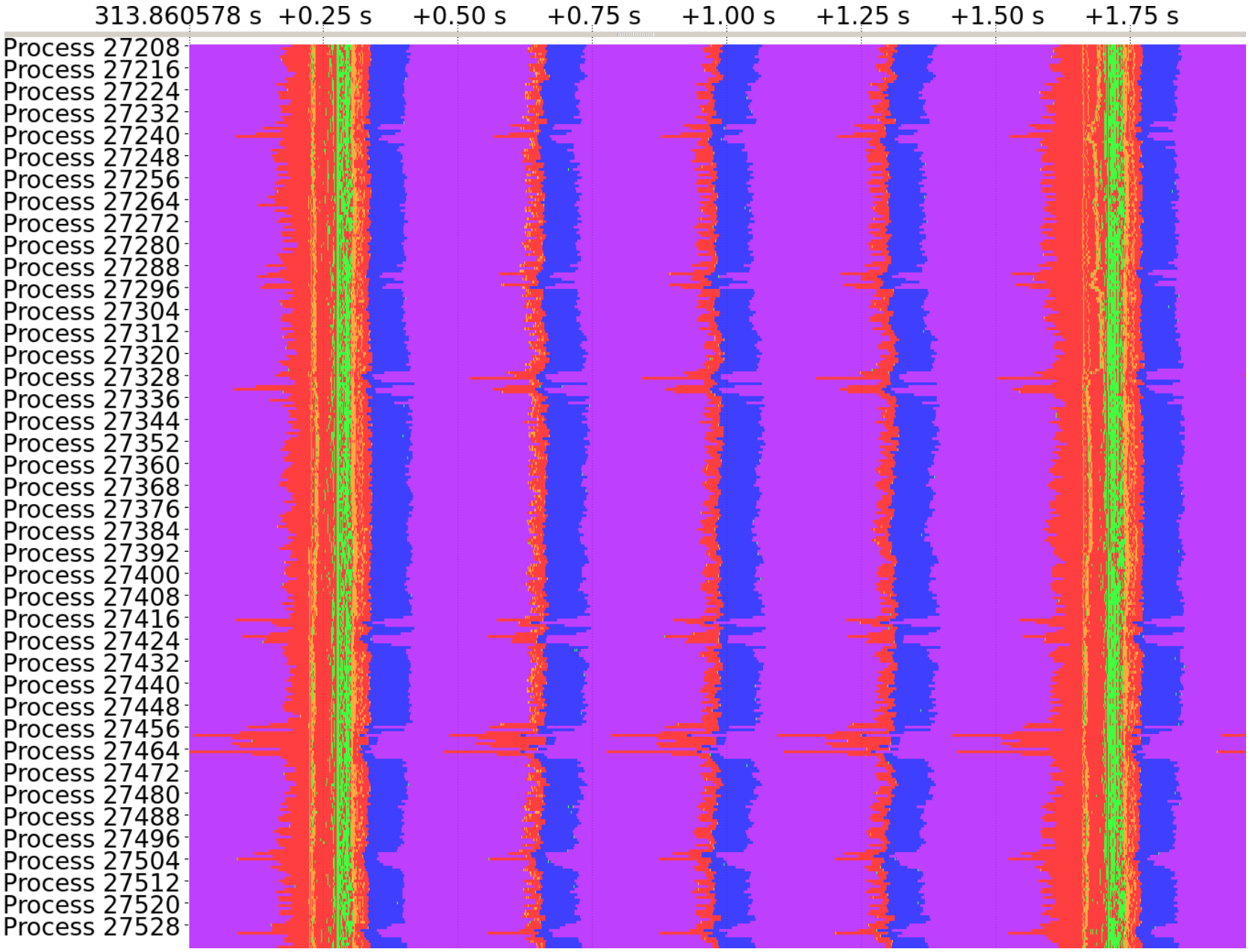
H2seq: 363 ms

H2par: 40.5 ms

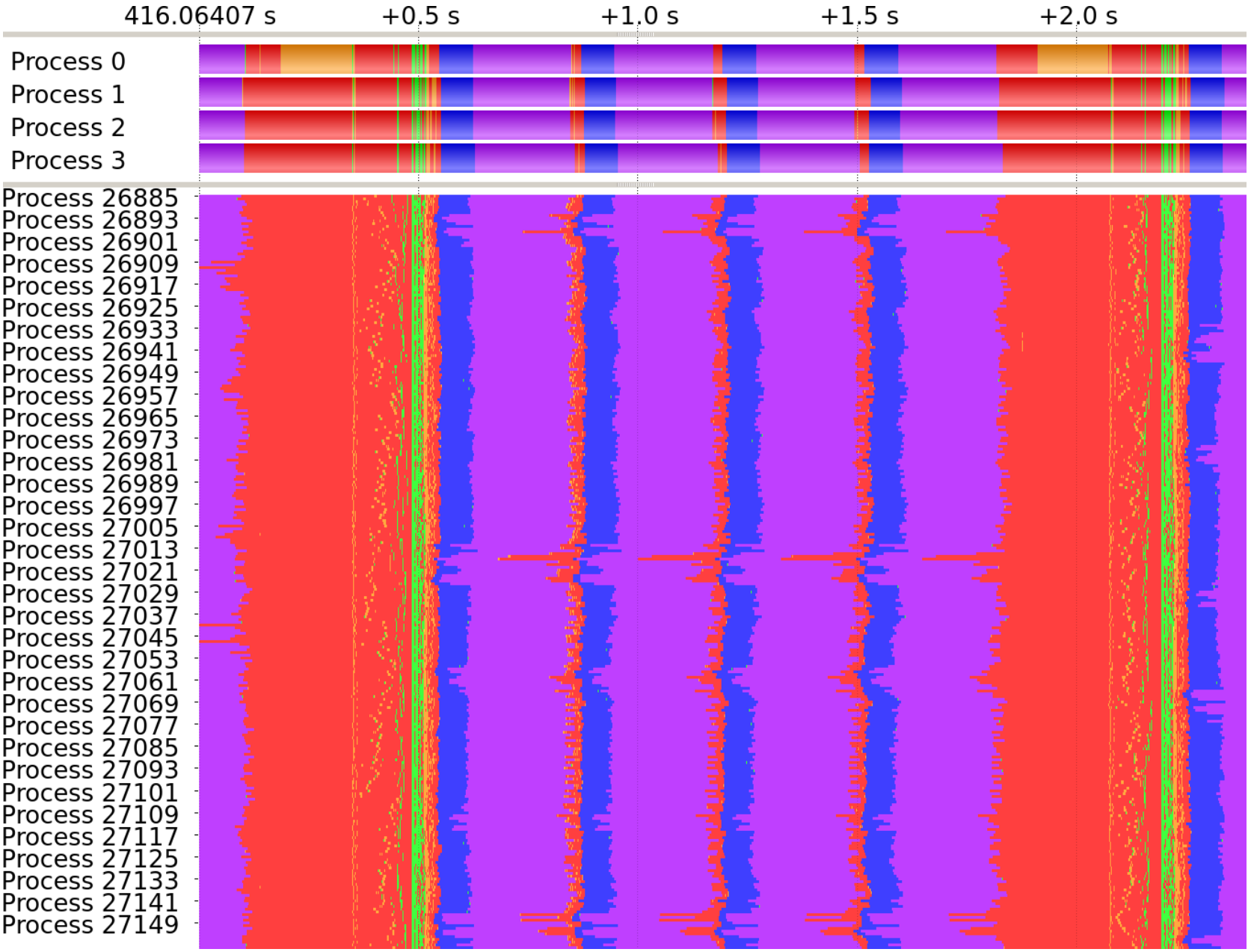
$HIER^*_{G=64}$: 8.55 ms

$HIER^*_{P/G=256}$: 3.77 ms

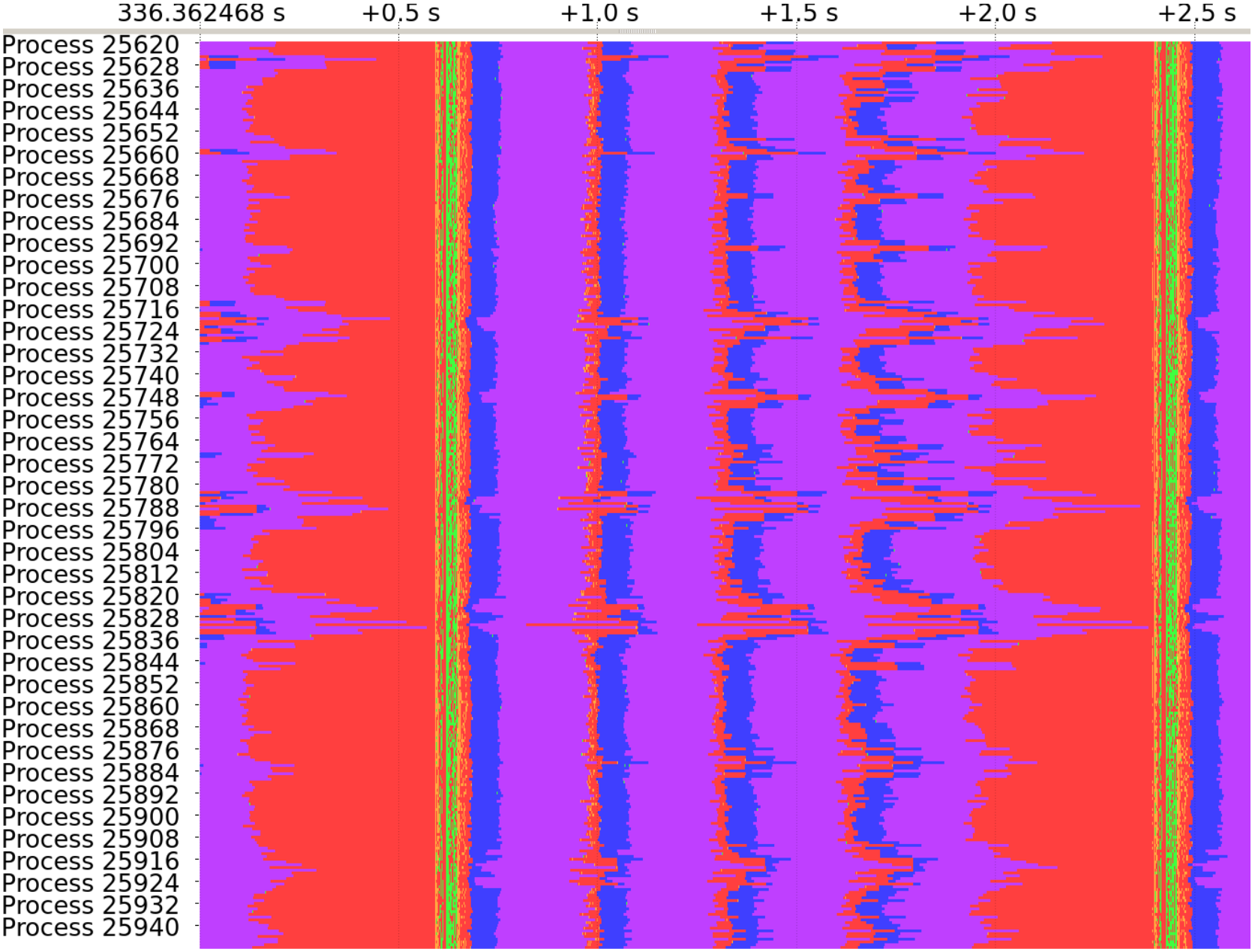
Heuristic H2 in Action (COSMO-SPECS+FD4)



ExactBS in Action (COSMO-SPECS+FD4)

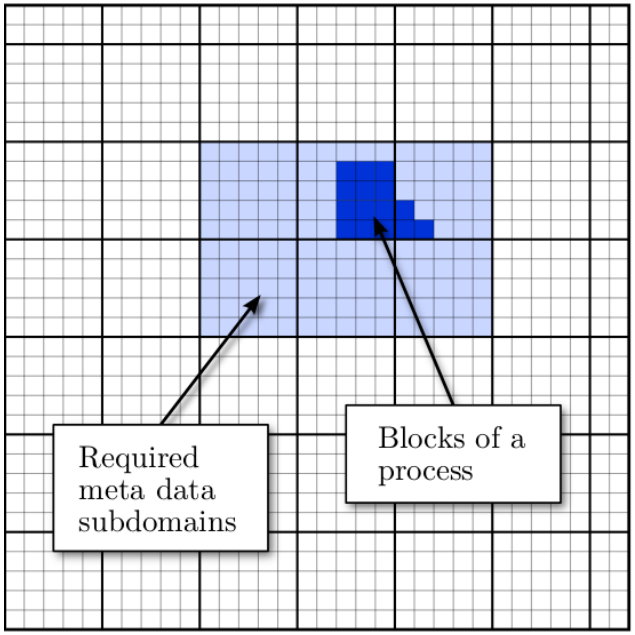


HIER* in Action (COSMO-SPECS+FD4)

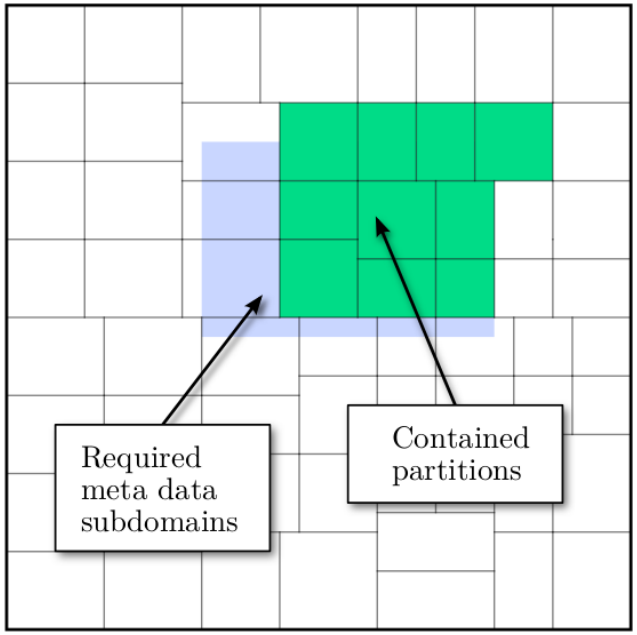


Scalable Coupling: Meta Data Subdomains

- "Handshaking" – Identifying partition overlaps between the coupled models – turned out to be the main scalability bottleneck
- Solved with spatially indexed data structure for coupling meta data in FD4
- Time for locating overlap candidates does not depend on number of ranks



(a) Required meta data subdomains.



(b) Contained coupled partitions.