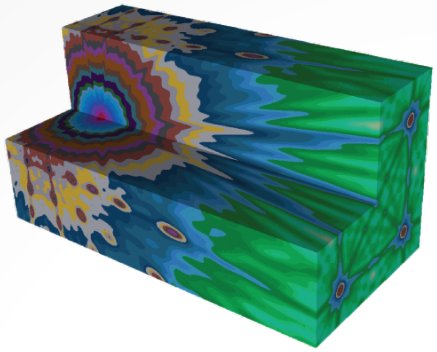


CERT

Center for Exascale Radiation Transport

Center for Exascale Radiation Transport

PDT Overview and Development with STAPL

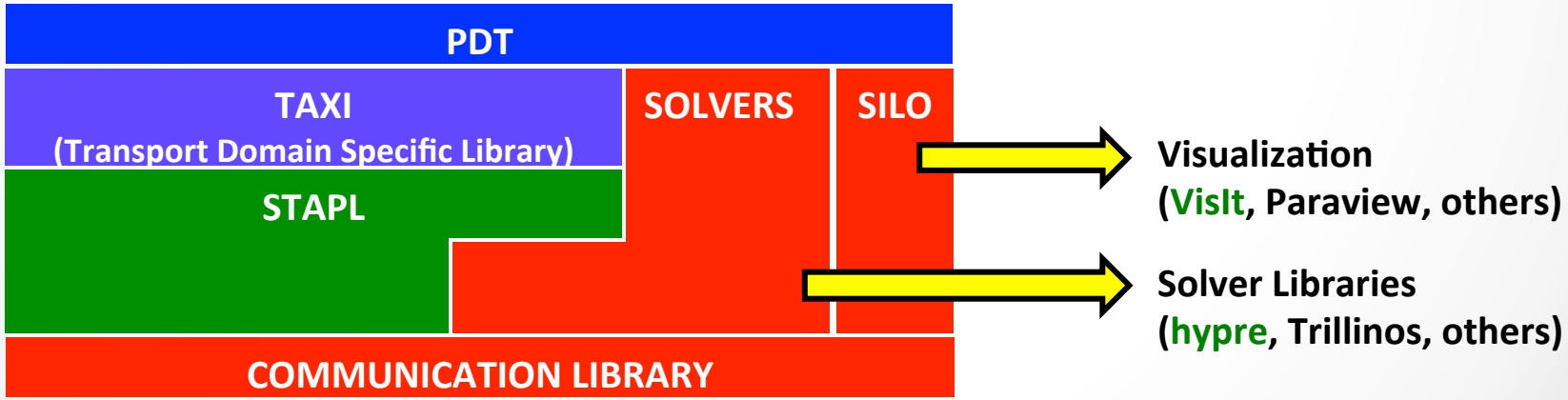
Marvin Adams, Timmie Smith

CERT Programming Model Deep Dive

February 18, 2014

PDT

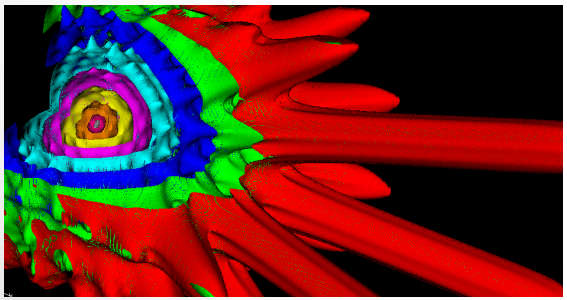
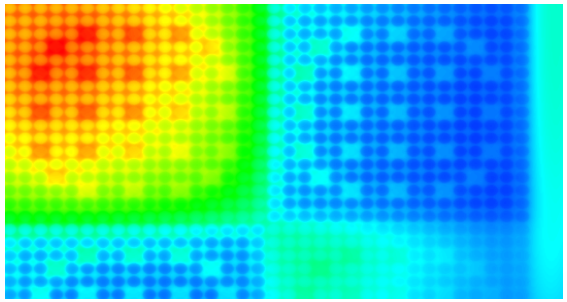
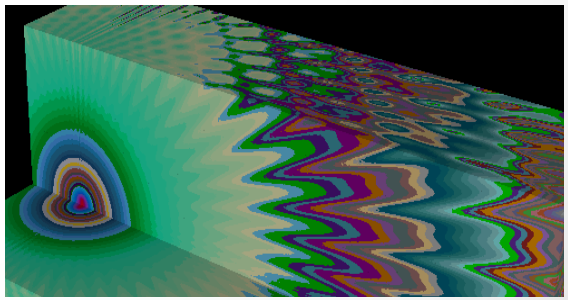
- 2D/3D, massively parallel, deterministic transport code written in C++
- Built on Standard Template Adaptive Parallel Library (STAPL)
- Uses a transport-specific domain library (TAXI)
  - In development (moving code from PDT into TAXI)
  - Specializes STAPL graph/scheduling/partitioning
  - Contains other common code and transport routines
  - Simplifies PDT and allows for rapid prototype of transport algorithms



Capabilities: General

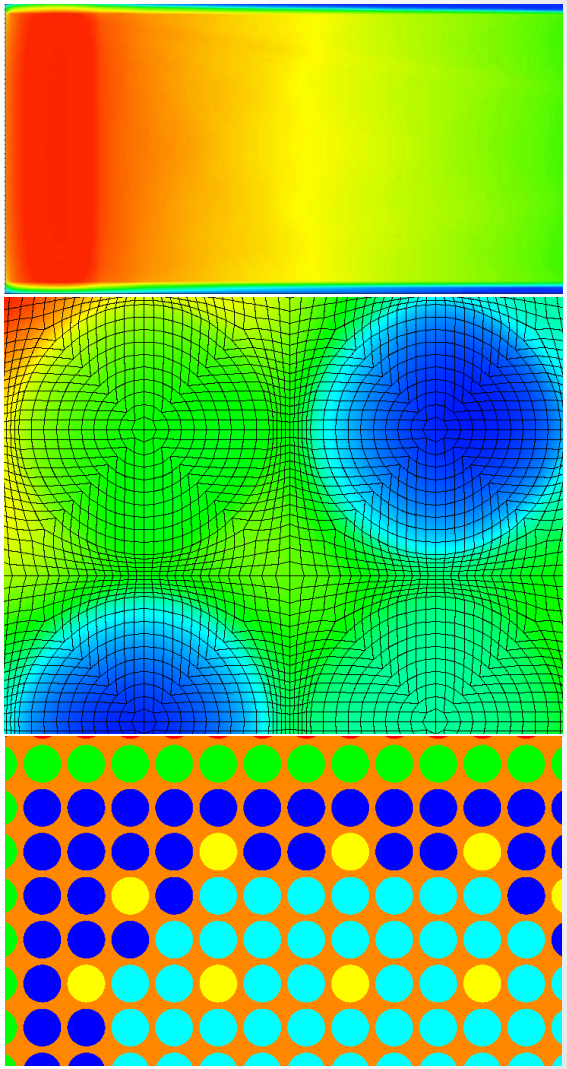
CERT
Center for Exascale Radiation Transport

- **Mixed-mode** parallelism (MPI/threads)
  (*testing and debugging now*)
- **Particle types**: neutron, gamma, coupled neutron-gamma, electron, coupled electron-photon, thermal radiation
- **Problem types**: Steady-state, Time-dependent, Criticality, Depletion
- **Uncollided flux**: Novel sophisticated treatment
- Can be called from external programs (DAKOTA, others)

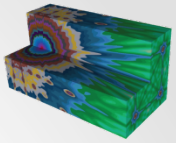




Capabilities: Numerical

- **Discretization:**
  - Energy: Multigroup and FEDS
  - Direction: Discrete ordinates (many quad. types)
  - Spatial: FEM, finite-volume (many options)
  - Time: Implicit (many options, including TBDF2)
  - Anisotropic scattering: arbitrary order
- **Iteration**: sweep-based; many options for accelerating convergence:
  - Diffusion preconditioners (CDFEM and MIP)
  - Gray diffusion for radiative transfer
    - *Can include temperature*
  - Krylov methods
  - Non-linear boundary projection acceleration (*in development*)
  - One-group diffusion for neutron upscattering (*in development)*



Capabilities: Sweeps



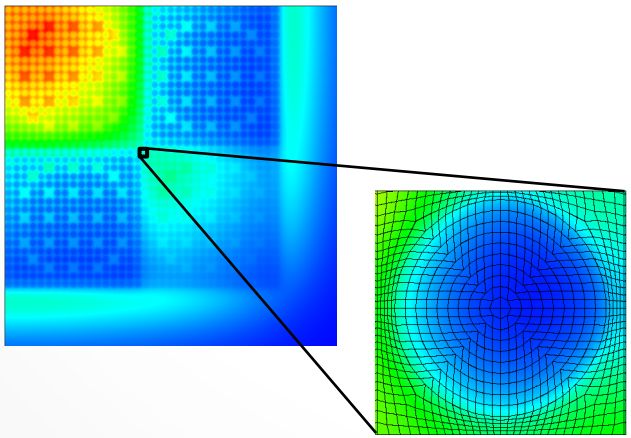
CERT
Center for Exascale Radiation Transport

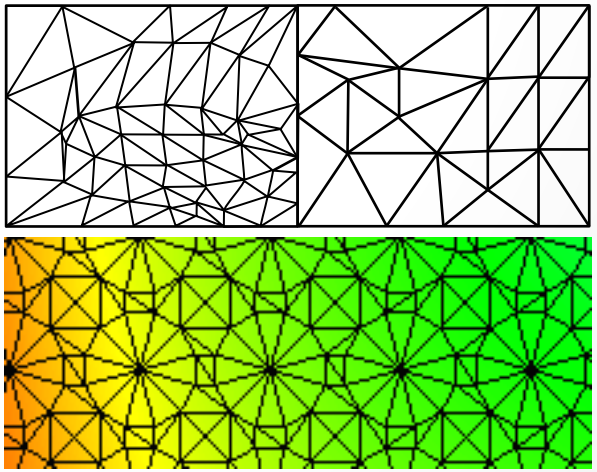
- Scalable parallel transport requires:
  - Scalable parallel algorithm for each iteration
  - Iterations counts that do not increase with resolution (*satisfied by sweep-based methods*)
- PDT employs a provably optimal sweep algorithm for rectangular (parallelepiped) *cellsets*:
  - Optimal scheduler that executes sweeps in the fewest possible stages (M&C 2013, M. P. Adams et al.)
  - Performance model that estimates solve time for given problem, machine, partitioning, and aggregation
  - Algorithm to select partitioning and aggregation parameters that minimize estimated solve time
- TAMU/LLNL collaboration has shown that sweeps can scale well to $O(10^6)$ cores

Capabilities: Grids

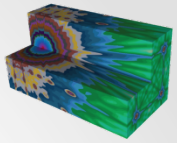
CERT
Center for Exascale Radiation Transport

- PDT has *general* sweep and work functions for arbitrary grids
- PDT has *optimal* sweeps for hierarchical grids: **rectangular** "cellsets" containing **arbitrary** cell shapes & connectivities
  - At cellset level, grid looks brick-like, which permits optimal sweeps
  - Within cellset are many options:
    - Regular rectangular
    - Reactor grids
    - 2D: Triangular (2D), triangular prism (3D), or Tetrahedral (3D) grids built independently and stitched at interfaces (forming polygons / polyhedra)





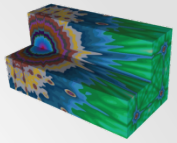
PDT Scaling



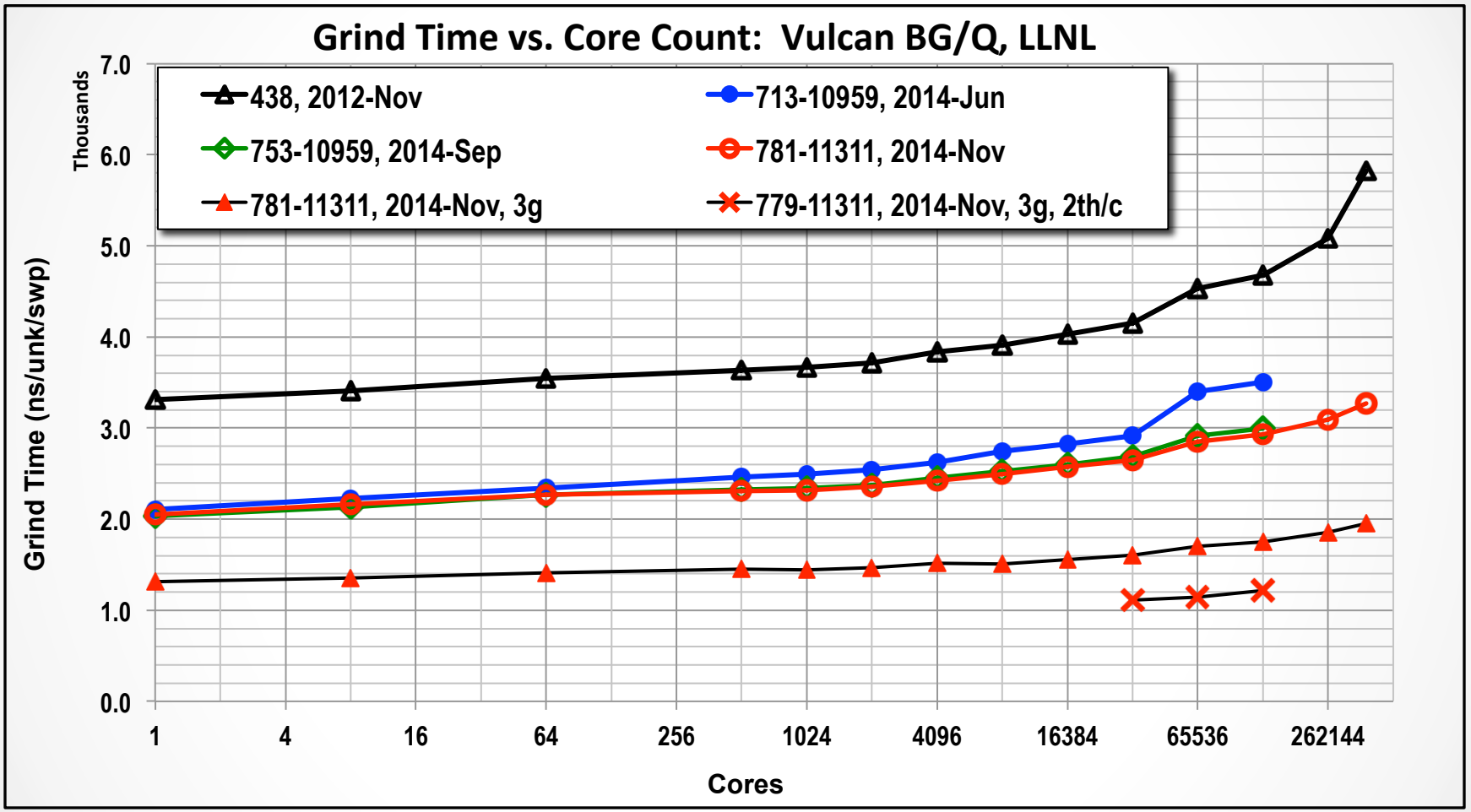
CERT
Center for Exascale Radiation Transport

- Performance modeling is critical
  - Most helpful in finding code issues and reducing overhead
  - Current model will be extended to mixed-mode and nested parallelism
- Current version of PDT runs with 1M MPI processes
  - Not long ago our largest run was 384K MPI processes
  - Recently completed runs with 512K and 1M MPI processes
    - This used vulcan (BG/Q) at LLNL with 16K nodes and 64 MPI processes/node (one per thread)

PDT/STAPL Grind Times

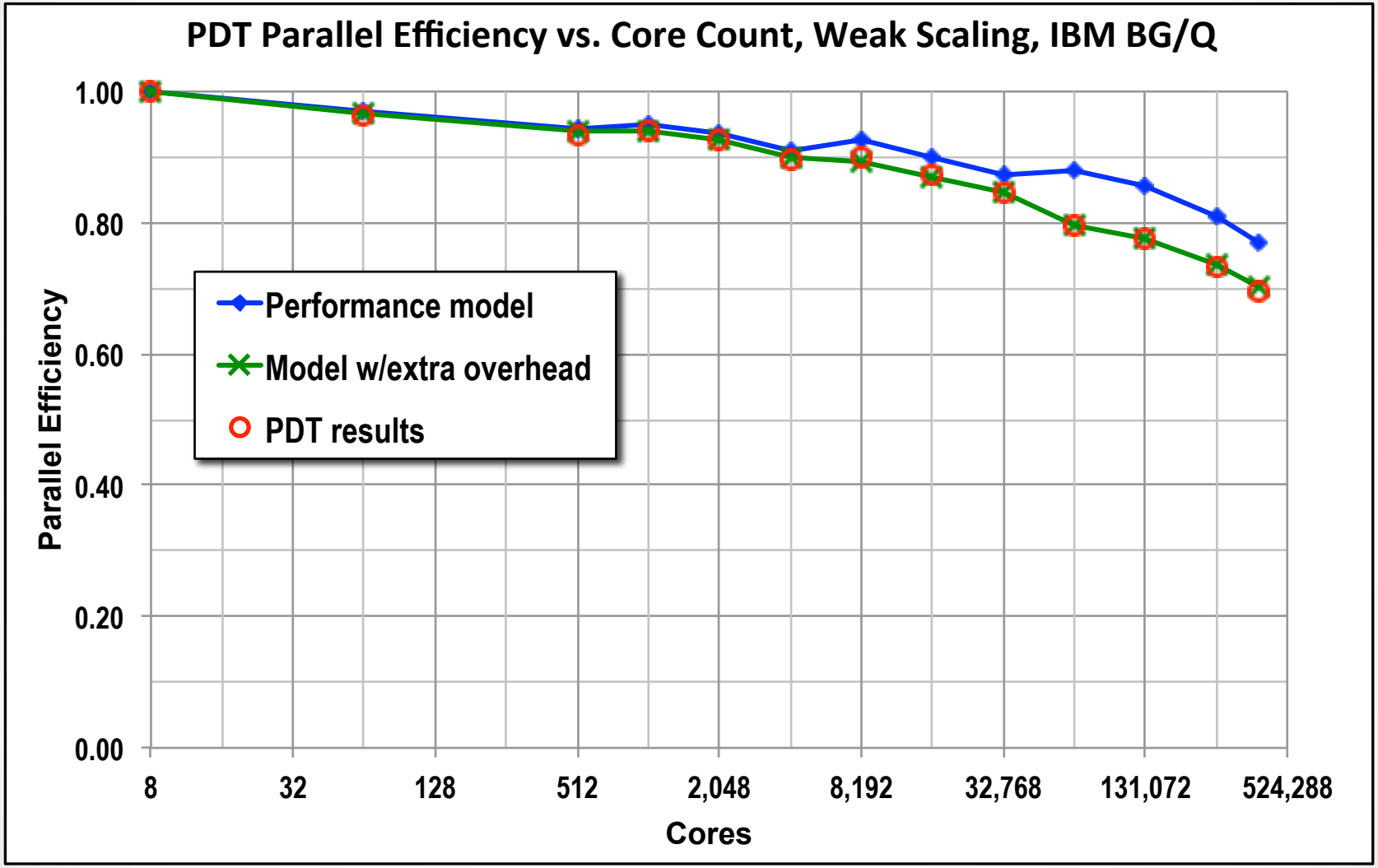


CERT
Center for Exascale Radiation Transport

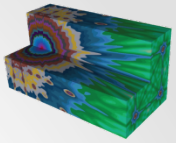


PDT/STAPL Scaling

CERT
Center for Exascale Radiation Transport



PDT-STAPL

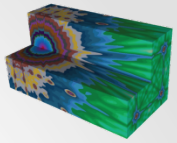


CERT
Center for Exascale Radiation Transport

- Parallelism in PDT is obtained via the underlying STAPL library
- Many of PDT's fundamental data structures are STAPL components

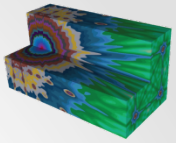
PDT Component	STAPL Component
Grid	graph
Krylov vectors	array
Grid vertices	array
Cellsets	hierarchical_graph_view
Grid cycle detection	strongly_connected_components
Iterative algorithms (inner products, etc.)	inner_product, map, map_reduce
PMPIO functionality	serial_sets skeleton
Task dependencies	explicit paragraphs
Sweep schedule	specialized scheduler

Integrating STAPL in PDT



- Ease of integrating different components varied
- array and array_view were directly usable in Krylov methods
- Grid and array of vertices (corners of discretized spatial unit)
  - Introduction of use straightforward
  - Data distribution support made construction difficult at the time
- Algorithm and Skeleton use has been simple
  - SCC for cycle detection
  - inner product, map, map_reduce for Krylov methods
  - serial_sets for parallel I/O
  - Skeleton composition to allow merging task graphs is still coming
- PARAGRAPH and runtime use
  - PARAGRAPH construction using explicit task factory a motivating example for Skeletons
  - Strict priority scheduling of sweeps influenced scheduler specialization interface

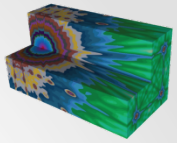
PDT Grids



CERT
Center for Exascale Radiation Transport

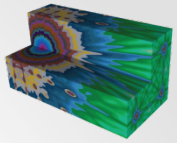
- Grid classes for 2D and 3D
- Inherit from stapl::dynamic_graph
  - vertex in graph represents unit of discretized spatial domain (cell)
  - edges in graph represent shared faces between cells
- Member functions
  - I/O
  - Vertex motion – skew shape of the cells
  - Boundary initialization
- Data members that are not stored per spatial unit
  - vertices that represent coordinates of corners of cells
  - maximum cell width, other geometry information
- Constructors use cell creator classes
  - Correctly initialize cell
  - Distribution of cells explicitly controlled by creator
  - STAPL data distribution specifications will simplify this

PDT Sweep Scheduler



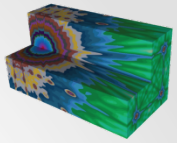
- Customized scheduler implemented for executor that processes PARAGRAPHs on each location
  - Choose from set of PARAGRAPHs that contain tasks ready to execute
- Goal: ensure minimum number of scheduling steps
  - Optimal sweep work guarantees that tasks of different sweep directions didn't delay one another
- PARAGRAPHs assigned a priority on each location
  - Intuitively, a PARAGRAPH with more tasks down stream has higher priority
- On each location
  - Process tasks of highest priority PARAGRAPH
  - If highest priority PARAGRAPH doesn't have ready tasks and isn't complete, wait on it
  - After current highest priority PARAGRAPH finishes all tasks on location, proceed to process next priority
- Strict scheduling isn't the default behavior for STAPL
  - Custom scheduler class that implements policy passed to executor at program startup

PDT a Motivating Example



- No STAPL components were custom built for PDT
  - Hierarchical graph_views used in graph partitioning
  - Containers, Algorithms, Skeletons, and PARAGRAPHs used in every STAPL application
- PDT was often the first application to use features
- PDT emphasized need for components
  - Sweep Pattern
  - Hierarchical graph views
  - Skeletons to replace explicit task factories
  - Support for custom data distributions

PDT and STAPL Co-design



CERT
Center for Exascale Radiation Transport

- PDT has driven overhead reduction in STAPL
  - PARAGRAPH and Runtime overheads
  - PDT performance model a lower bound on overhead

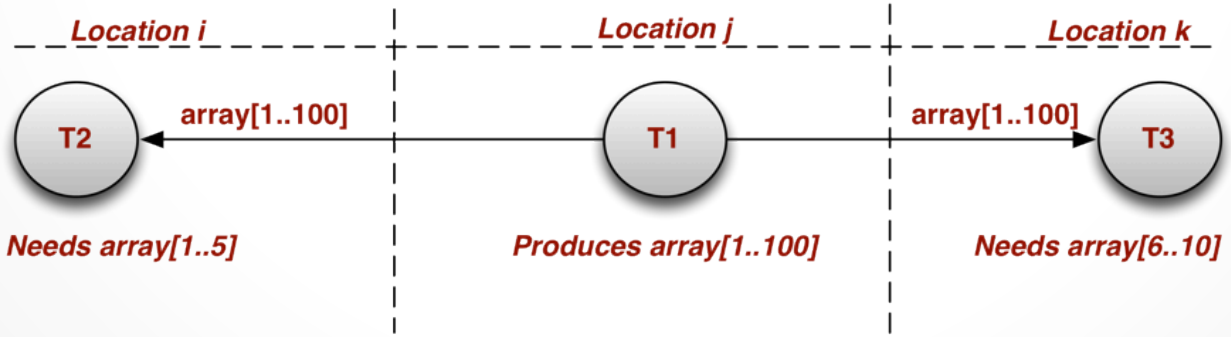
- STAPL has simplified development of transport algorithms
  - Fine-grain algorithm specification
  - run-time independent implementation
    - mixed-mode will be introduced without changing code

Partial PARAGRAPH Edge Consumption

Problem

- Coarser tasks increase likelihood successors will only consume to a piece of (aggregate) produced data.
- Finite bandwidth, wasted if we transmit entire edge value.

```
add_task(T1, wf1, v1, 2);  // produces array[1..100];
add_task(T2, wf2, consume(T1)); // wf2 filters to [1..5]
add_task(T3, wf3, consume(T1)); // wf3 filters to [6..10]
```



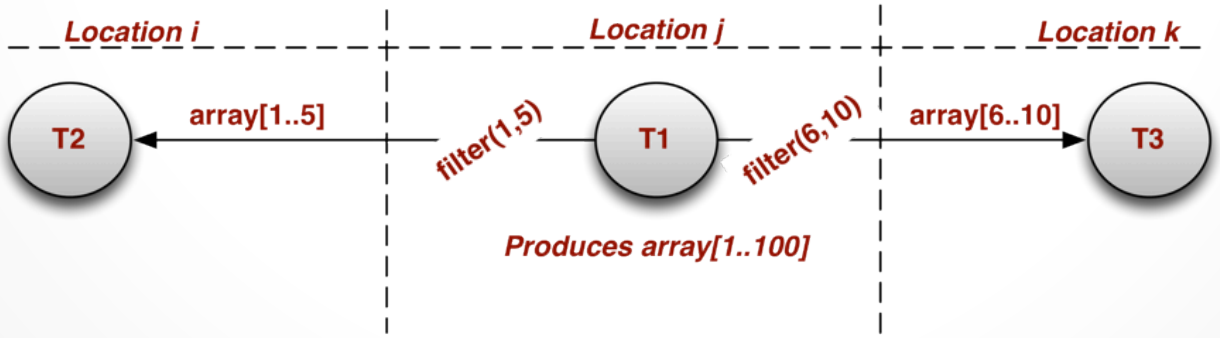
We transport 200 elements, when only 10 are consumed.

Partial PARAGRAPH Edge Consumption

Solution

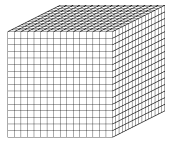
- Filter edge value *prior* to transport.
- Add optional *filtering function* to `consume()` interface.

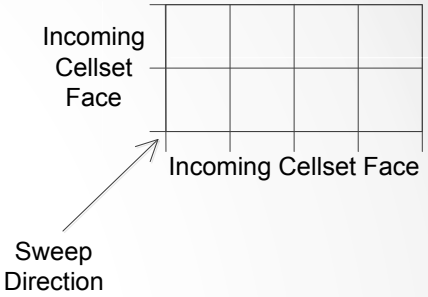
```
add_task(T1, wf1, v1, 2);  // produces array[1..100];
add_task(T2, wf2, consume(T1, filter_func(1,5));
add_task(T3, wf3, consume(T1, filter_func(6,10));
```

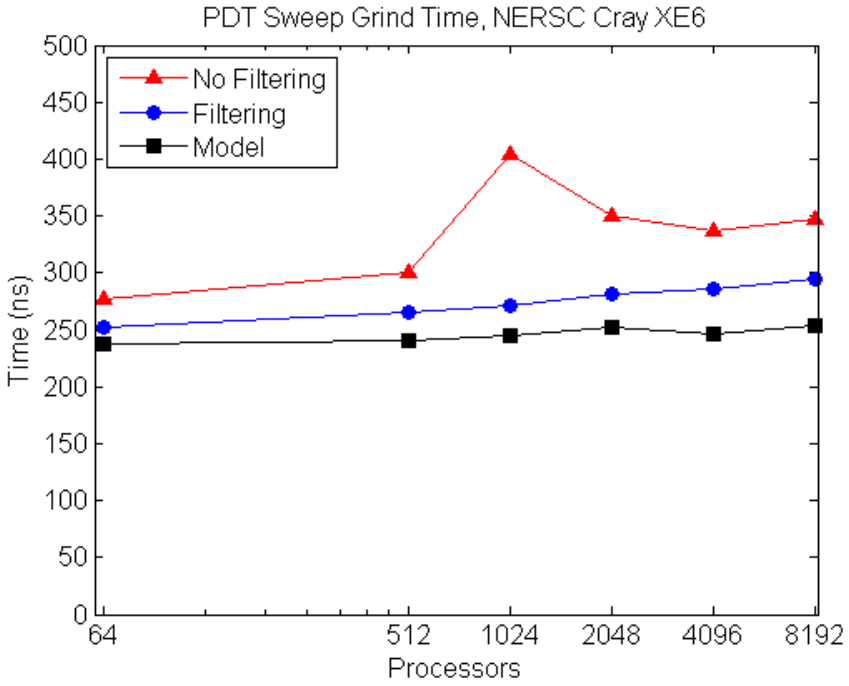


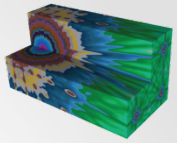
Partial Edge Consumption in PDT

- In sweep, cellsets produce data for multiple faces but consume from one face of each predecessor.
- Sweep pattern uses filtering functions, transparently restrict data prior to calling workfunction.









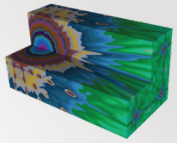
Persistent PARAGRAPHs

Problem

- Portions of global task dependence (e.g., loops) recreate (near) identical **PARAGRAPH**s.
- Even with overlapped creation, redundant & unnecessary computation spent to create **PARAGRAPH**s.

Solution

- Create and execute first **PARAGRAPH** as before, disabling immediate task retirement so that task graph is *persistent*.
- Persistent **PARAGRAPHs** can reused by re-invoking execute().
- Destruction of **PARAGRAPH** object initiates retirement of all tasks and reclamation of resources.

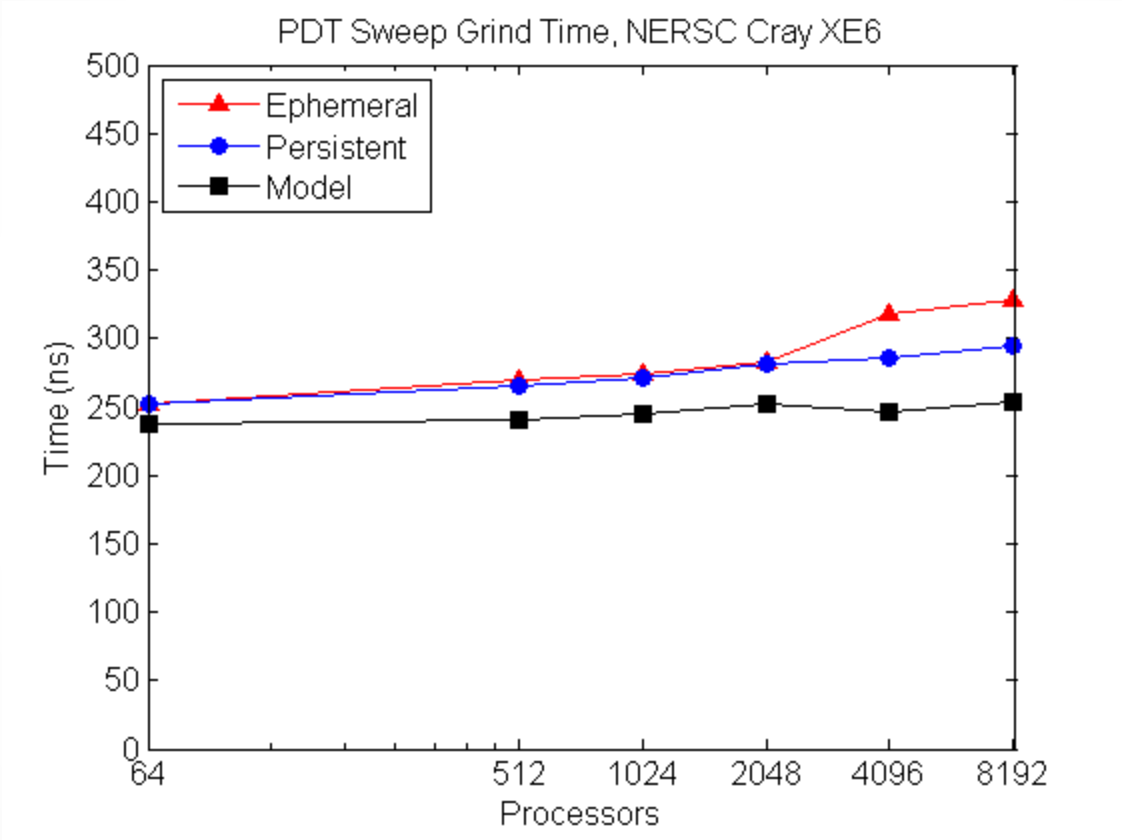


Persistent PARAGRAPHs – Howto

- Enabled by `typedef` flag in the scheduler (per **PARAGRAPH**).
  *User guarantees **PARAGRAPH** structure input insensitive.*

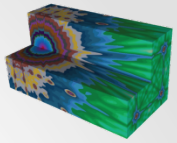
```
struct my_scheduler
{
  typedef void enable_persistence;
  …
};

void foo(v1, v2)
{
pg = make_paragraph(f, v1, v2, my_scheduler());
retval_1 = pg.execute(); // initialize and execute once.
…
retval_2 = pg.execute(); // reuse for another execution();

} // pg destroyed at end of scope. PARAGRAPH dismantled.
```

Persistent PARAGRAPH in PDT Sweep



At 8K, persistency and filtering pushed efficiency from 70% to 78%.

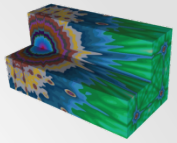
STAPL simplifies development



CERT
Center for Exascale Radiation Transport

- MPI
  - Explicit messaging in user code to express dependencies
  - Manual serialization/deserialization
  - No access to elements on another process
- STAPL
  - Task dependencies enforced by STAPL
  - Serialization only requires method to specify data members
  - Shared object view of containers and views allows simple data access
- Plans to Improve Usability
  - Simplifying or automating generation of define_type method
  - Introduce Skeleton composition in PDT to reduce development of work functions

Current PDT Development



- Introducing mixed-mode
- Planning introduction of nested parallelism
- Kripke will accelerate process
  - Test support for mixed-mode, mapping on to system
  - Develop approach for nested parallelism in Kripke first
    - Smaller code base with fewer Transport Algorithms to keep online
  - Algorithms developed in Kripke will be applied in PDT

Interoperability in STAPL

Ioannis Papadopoulos
ipapadop@cse.tamu.edu

Parasol
Smarter computing.
Texas A&M University

http://parasol.tamu.edu

Why interoperability?

- Existing optimized algorithm implementations
  - STAPL should be able to use them
- Existing applications
  - They should be able to call STAPL effortlessly
- Use views to avoid data copying

STAPL using external libraries

Parasol

- STAPL can use external libraries
  - BLAS, PBLAS as presented in **Design for Interoperability in STAPL : pMatrices and Linear Algebra Algorithms**, A. Buss, T. Smith, G. Tanase, N. Thomas, M. Bianco, N.M. Amato, L. Rauchwerger, *LCPC08*
  - Zoltan redistribution for associative containers (e.g. `stapl::unordered_map`, `stapl::graph`)
  - STAPL Graph500 benchmark implementation uses Graph500 reference implementation generator
- `stapl::external()` call
  - Invoke non-STAPL distributed memory library functions
  - Halts STAPL; ensures quiescence before function call
- Discussions on how to integrate closer with the stapl::skeletons framework

Existing Applications invoking STAPL

Parasol

- STAPL can be initialized and invoked from existing applications
- PDT has a regular main() program entry point

```
int main(int argc, char* argv[]) {
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, …);
  // other MPI code
  auto opt = option{argc, argv} & option{"MPI_Comm", comm};
  stapl::initialize(opts);
  stapl::execute(pdt_entry_point_wf{my_generated_data});
  stapl::finalize();
  // other MPI code
  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

- You can enter and exit STAPL as many times as you want

stapl/benchmarks/kripke

Adam Fidel, Mani Zandifar,
and Nathan Thomas

Parasol Laboratory
Department of Computer Science and Engineering
Texas A&M University

Parasol

Kripke Overview

- Kripke Overview
- Designing the STAPL implementation
  - Data structures
  - Skeleton based algorithm
  - Views
- Current work

Parasol Outline

- An Sn Transport Mini App
- Not a useful calculation - designed to be a representative computational load
- Features all 6 striding orders of directions, groups, and zones
- Specific computational kernel for each “nesting” of domains
- Simplified, no scattering

Parasol
Smarter computing.
Texas A&M University

Kripke Overview

$$\frac{1}{\nu}\frac{\partial \psi}{\partial t} + \Omega \cdot \nabla \psi + \sigma \phi = \sigma_s \psi + \sigma_f \phi + q$$

$$H\Psi^{i+1} = L^{+}SL\Psi^{i} + q$$

$$H\Psi^{i+1} = L^{+}IL\Psi^{i} + 0$$

$$H\Psi^{i+1} = L^{+}L\Psi^{i}$$

- Sweep Kernel (On Core)
  - 3D Diamond Difference
- Parallel Sweep Algo. (MPI)

- LPlusTimes Kernel (moments -> discrete)

- LTimes Kernel (discrete -> moments)

Parasol Kripke Overview

- Explicit MPI + OpenMP
- Process-level sweep across spatial domain in MPI
- Various kernel configurations
  - With OpenMP, process spatial, energy, and direction domains
    - Primary variables psi / rhs (5D) and sigt (4D).
    - Data layout changes with kernel
      - Layout matches loop nesting
- Kernel name indicates order of loop nesting (i.e., ZDG -> Zones, Directions, Groups)

Parasol
Smarter computing.
Texas A&M University

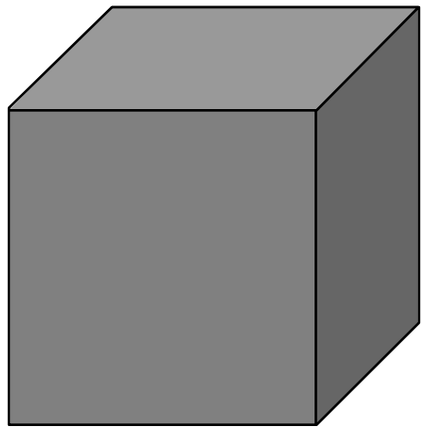
Kripke Computation Structure

```
Sweep_Solver()
   kernel->LTimes()          - compute phi data

   kernel->LPlusTimes()      - compute rhs data

   for (GS)
forall(DS) {
  MPI_IReceive()     - receive from the upstream of this dir
  if (data is received)
    kernel->Sweep()  - sweep the local grid and compute psi
  MPI_ISend()        - send to the downstream of this dir
}
```

Parasol
Smarter computing.
Texas A&M University

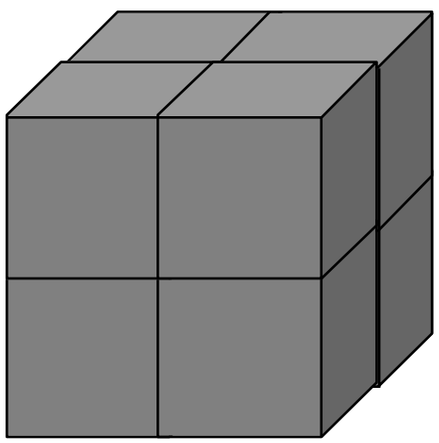
Spatial Decomposition

- Kripke provides spatial decomposition across MPI tasks.
- Each process manages its local grid, which is decided by the number of processors on axis x, y, z.

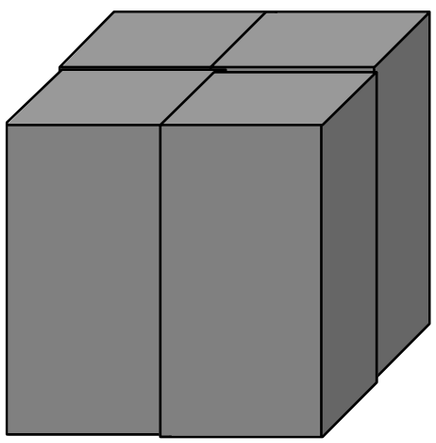


Global space

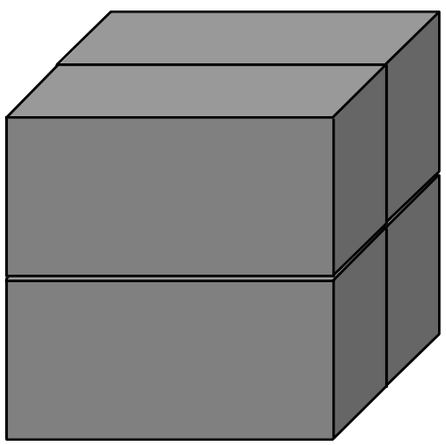
npx : npy : npz
(2 : 2 : 2)



npx : npy : npz
(2 : 1 : 2)



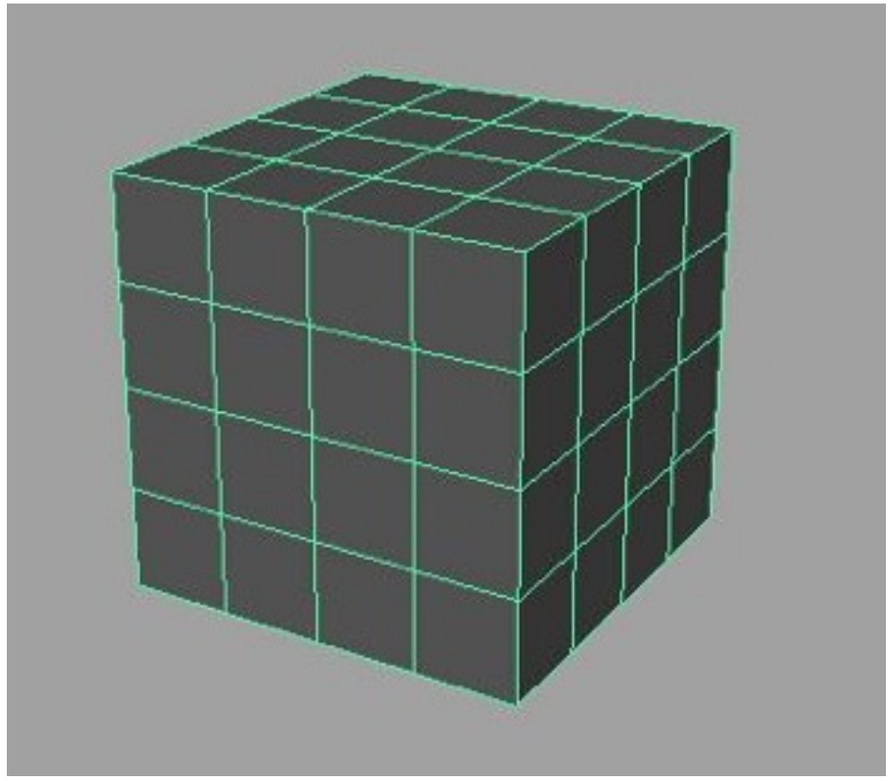
npx : npy : npz
(1 : 2 : 2)



Parasol
Smarter computing.
Texas A&M University

Spatial Decomposition

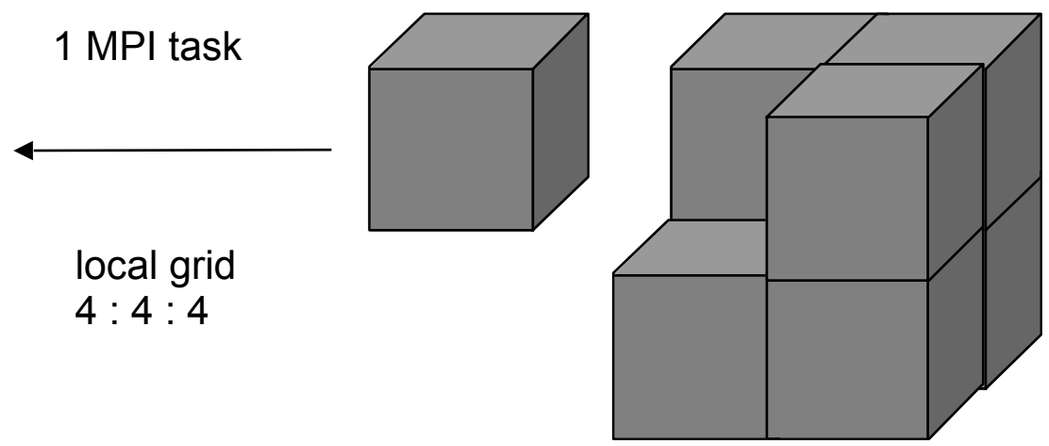
- One MPI task will covers all the Groups and Directions on a subset of zones.
- Data is transferred through the shared surface of two adjacent MPI tasks.



npx : npy : npz
(2 : 2 : 2)
nzones
(8 : 8 : 8)

1 MPI task

local grid
4 : 4 : 4



Parasol
Smarter computing.
Texas A&M University

Parallelism Within Kripke Kernel DGZ

LTimes

```
for each Group Set
  for each Direction Set
    for each entry in L matrix
      for each direction within Direction Set
        forall (groups in set * zones)
```

LPlusTimes

```
for each Group Set
  for each Direction Set
    for each direction within Direction Set
      for each entry in L+ matrix
        forall (groups in set * zones)
```

SweepSolver

```
for each Group Set
```

SweepSolver_GroupSet

```
forall Direction Sets
```

Sweep

```
for each direction within Direction set
  forall groups within Group Set
    for k     |
      for j   | for each zone
        for i |
```

Parallelism Within Kripke Kernel ZGD

Parasol
Smarter computing.
Texas A&M University

LTimes

```
for each Group Set
  for each Direction Set
    forall zones
      for each group within Group Set
        for each direction within Direction Set
          for each entry in L matrix
```

LPlusTimes

```
for each Group Set
  for each Direction Set
    forall zones
      for each group within Group Set
        for each direction within Direction Set
          for each entry in L+ matrix
```

SweepSolver

```
for each Group Set
```

SweepSolver_GroupSet

```
forall Direction Sets
```

Sweep

```
for k      |
  for j    | for each zone
    for i  |
      forall groups within Group set
        for each direction within
            Direction Set
```

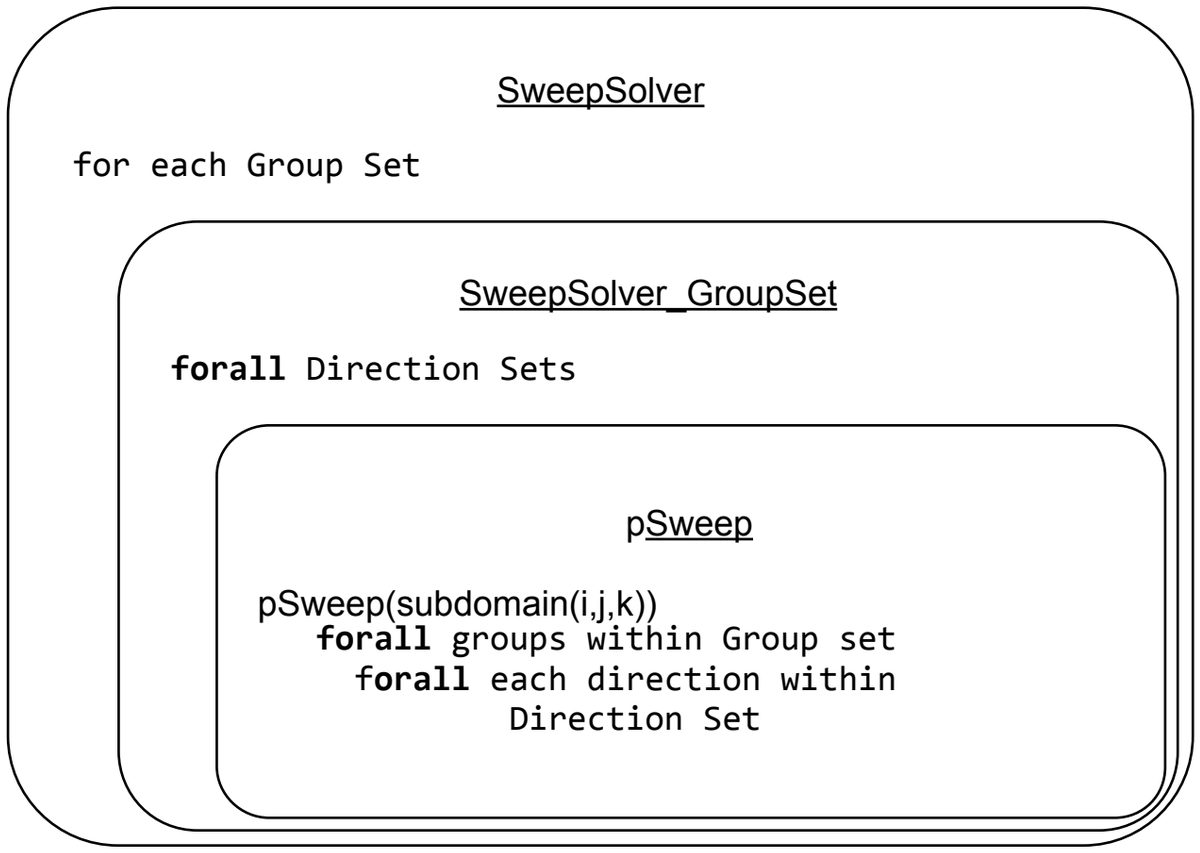
Outline

- Kripke Overview
- Designing the STAPL implementation
  - Data structures
  - Skeleton based algorithm
  - Views
- Current work

Implementation Overview

- Unified model for nested parallelism
  - Reuse of algorithms (i.e., same skeleton implementations at both levels of nesting)
  - Addition of support for sweep within sweep
    - Uses wavefront skeleton
- Abstract machine model, map onto the machine
- Nested containers used to define zonesets.

STAPL Kripke, ZGD Kernel



Implementation Overview

- Containers
  - Uses multiarray<3, multiarray<5, double>>
  - Mimics setup of Kripke reference code
  - View-based, Composed distribution specifies distribution of both levels

- Algorithms
  - Use compositions of wavefront and zip skeletons
  - Fine grain solver workfunction
    Unit of work:
    DD Sweep over One Direction, One Group, and One Zone

- Views
  - Conform data structure to kernel composition
  - New STAPL views - slices_view & extended_view

Parasol Fine Grain Solver WorkFunction

Unit of work: DD Sweep over One Direction, One Group, and One Zone

```
struct diamond_difference_wf {
 …
 operator()(psi, rhs, sigt, lf_bndry, fr_bndry, bo_bndry)
 {
    …

    double zcos_dzk = 2.0 * m_directions[d].zcos / dzk;
    … // ycos_dyj, xcos_dxi

    // Calculate new zonal flux
    double new_psi = (rhs + lf_bndry * xcos_dxi
                          + fr_bndry * ycos_dyj
                          + bo_bndry * zcos_dzk) /
                         ((xcos_dxi + ycos_dyj + zcos_dzk + sigt));
    psi = new_psi;

    // Apply diamond-difference relationships
    new_psi *= 2.0;
    lf_bndry = new_psi - lf_bndry;
    …
  }
};
```

Outline

- Kripke Overview
- Designing the STAPL implementation
  - Data structures
  - Skeleton based algorithm
  - Views
- Current work

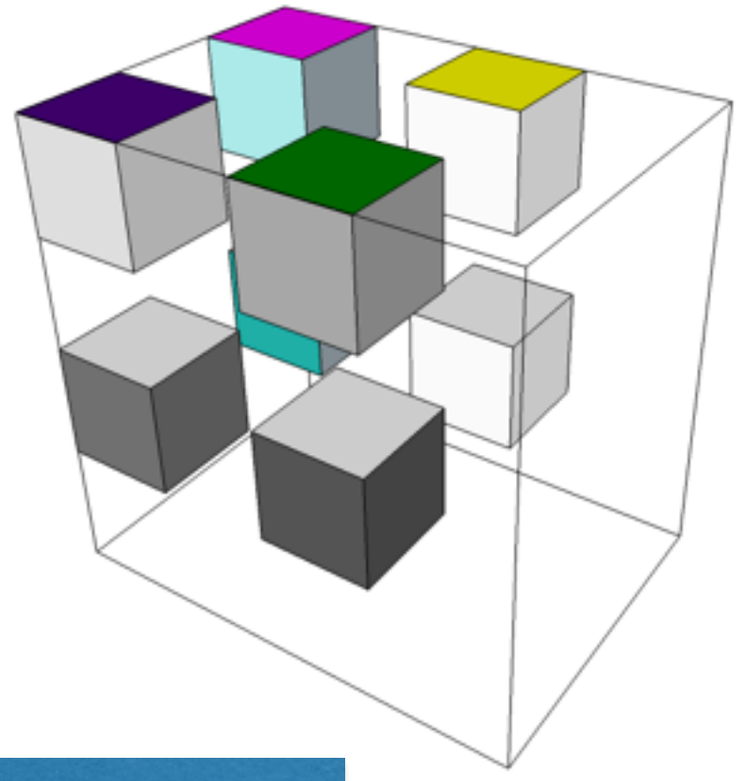
Containers

- Structured spatial decomposition.
  - Use `stapl::multiarray` instead of more general `stapl::graph` (used in PDT).
- Two Level container composition (rhs, psi)
  `stapl::multiarray<3, stapl::multiarray<5, double>`
- In contrast to Kripke reference, inner container has an explicit data distribution.
  - Both Distribution and layout changes with kernel
  - Regardless of indexing remains (i, j, k, D, G)

Parasol Zoneset Container

- Outer multiarray has one element per zoneset.
- Uniform distribution in 3D location layout.

```
multiarray<3,…>(
 stapl::uniform_nd({2, 2, 2}, {4, 4, 4})
);
```



2x2x2 Zoneset Layout

4x4x4 Location Layout

Kernel Container

- Explicit data distribution as compared to shared memory, OpenMP reference
  - Balanced distribution of Spatial, Direction, or Group Domain (whichever is on the outside).
  - Constant indexing (i, j, k, D, G)
  - Changing data layout (traversal)

DGZ

```
typedef ints_<3>              part_dims;
typedef ints_<3,4,0,1,2>      traversal_t;
multiarray<5>(
  partial_balanced<part_dims, traversal_t>({8,8,8,4,2})
);
```

Kernel Container Example

ZDG

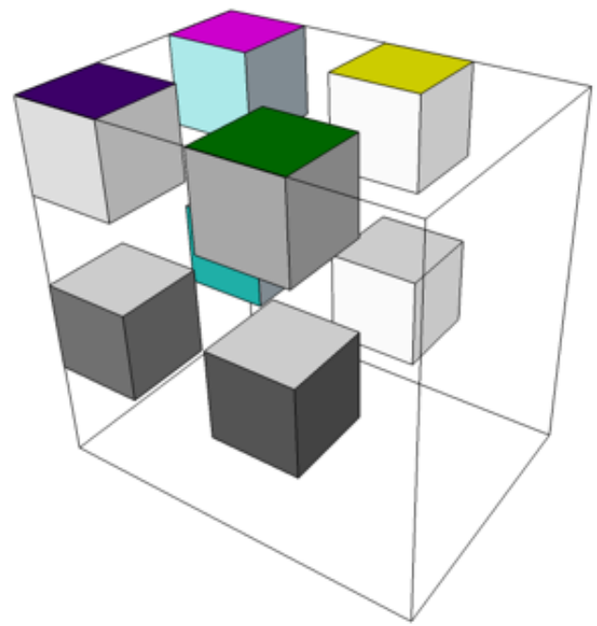
```
typedef ints_<0,1,2>        part_dims;
typedef ints_<0,1,2,3,4>    traversal_t;
multiarray<5, …> ma(
  partial_balanced<part_dims, traversal_t>({8,8,8,4,2})
);
```

Parasol Composing the Distributions

- Previous examples showed inner and outer containers made in isolation
- Actually composed container instantiation made in single statement:

```
multiarray<multiarray<...>> psi(...);
```

- Need to combine distribution specifications into single, composed distribution (view-based)
- Address mapping of each inner container to subset of locations



Parasol Composing the Distributions

```
composed_spec vw(…);
multiarray<multiarray<…>> ma(vw);
```

```
composed_spec_wf {
    // Specify Outer Container Distribution
    operator()(){
     return uniform_nd(…);
    }

    // Specify Inner Container Distribution
    operator()(typename Spec1::gid_type gid) {
       location_range = f(gid);
       return partial_balanced<…>(size, location_range);
    }
};
```

Parasol Kripke

- Kripke Overview
- Designing the STAPL implementation
  - Data structures
  - Skeleton based algorithms
  - Views
- Current work

Kripke Algorithm

Reference Implementation

```
void SweepSolver (User_Data *user_data)
{
  Kernel *kernel = user_data->kernel;
  Grid_Data *grid_data = user_data->grid_data;

  for(int iter = 0;iter < user_data->niter;++ iter){
    // Discrete to Moments transformation
    kernel->LTimes(grid_data);

    // Moments to Discrete transformation
    kernel->LPlusTimes(grid_data);

    for (int group_set = 0;group_set < num_group_sets;++ group_set) {
      doall(int direction_set = 0; ds < num_direction_sets; direction_set++) {
          ...
          kernel->sweep(grid_data,
                        dir_sets[direction_set],
                        i_plane_data[direction_set],
                        j_plane_data[direction_set],
                        k_plane_data[direction_set]);
          ...
      }
    }
  }
}
```

Parasol Sweep Kernels

- Sweep kernel is specialized over the loop orders
  - Zone-Group-Direction (ZGD)
  - Zone-Direction-Group (ZDG)
  - Group-Zone-Direction (GZD)
  - Group-Direction-Zone (GDZ)
  - Direction-Zone-Group (DZG)
  - Direction-Group-Zone (DGZ)
- The loop orders are interchanged

Reference Implementation

ZDG vs. ZGD

```
Kernel_ZGD::sweep(args...)
{
  for (int k = extent.start_k; k != extent.end_k; k += extent.inc_k)      // #1
    for (int j = extent.start_j; j != extent.end_j; j += extent.inc_j)    // .
      for (int i = extent.start_i; i != extent.end_i; i += extent.inc_i) // .
        for (int group = 0; group < num_groups; ++group)                  // #2
          for (int d = 0; d < num_directions; ++d)                        // #3
            diamond_difference_wf(…);
}
```

```
Kernel_ZDG::sweep(args...)
{
  for (int k = extent.start_k; k != extent.end_k; k += extent.inc_k)      // #1
    for (int j = extent.start_j; j != extent.end_j; j += extent.inc_j)    // .
      for (int i = extent.start_i; i != extent.end_i; i += extent.inc_i) // .
        for (int d = 0; d < num_directions; ++d)                        // #3
          for (int group = 0; group < num_groups; ++group)              // #2
            diamond_difference_wf(...);
}
```

Parasol Zip Skeletons

- We use the zip skeleton for loops #2 and #3

```
Kernel_ZGD::sweep(args...)
{
  for (int k = extent.start_k; k != extent.end_k; k += extent.inc_k)      // #1
    for (int j = extent.start_j; j != extent.end_j; j += extent.inc_j)    // .
      for (int i = extent.start_i; i != extent.end_i; i += extent.inc_i) // .
        for (int group = 0; group < num_groups; ++group)                  // #2
          for (int d = 0; d < num_directions; ++d)                        // #3
            diamond_difference_wf(...);
}
```

```
Kernel_ZGD::sweep(args...)
{
  for (int k = extent.start_k; k != extent.end_k; k += extent.inc_k)      // #1
    for (int j = extent.start_j; j != extent.end_j; j += extent.inc_j)    // .
      for (int i = extent.start_i; i != extent.end_i; i += extent.inc_i) // .
        exec.execute(zip(zip(diamond_difference_wf(...))),                // #2, #3
                     input_views...
}
```

Parasol Wavefront Skeleton

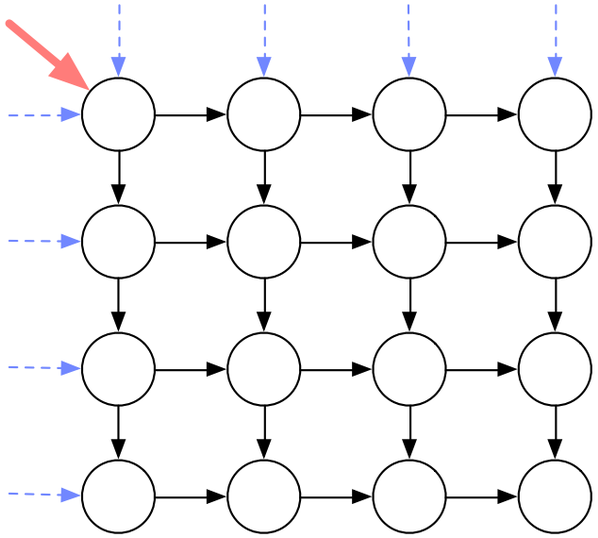
- We use the wavefront skeleton for loop #1

```
Kernel_ZGD::sweep(args...)
{
  for (int k = extent.start_k; k != extent.end_k; k += extent.inc_k)      // #1
    for (int j = extent.start_j; j != extent.end_j; j += extent.inc_j)    // .
      for (int i = extent.start_i; i != extent.end_i; i += extent.inc_i) // .
        for (int group = 0; group < num_groups; ++group)                  // #2
          for (int d = 0; d < num_directions; ++d)                        // #3
            diamond_difference_wf(...);
}
```

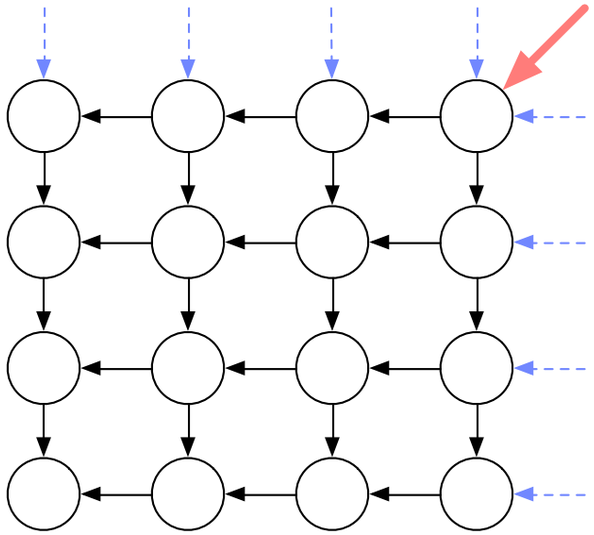
- A topologically ordered traversal of a data structure
- Wavefront on regular grids in STAPL
  - 2D wavefront skeleton
  - 3D wavefront skeleton

Parasol 2D Wavefront Skeleton

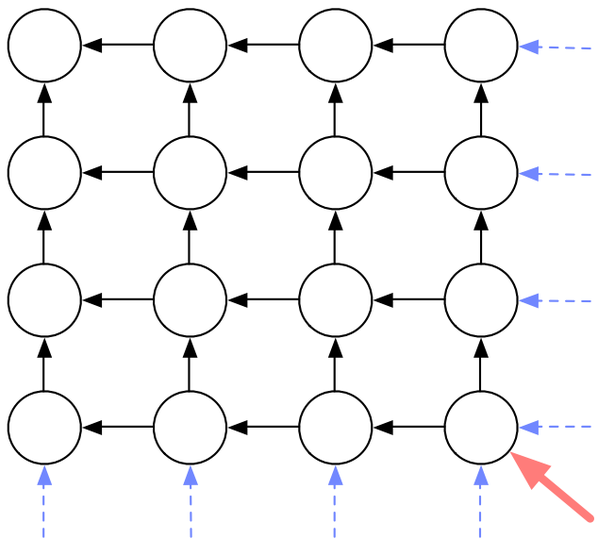
- Parameterized over the size and corner to start from



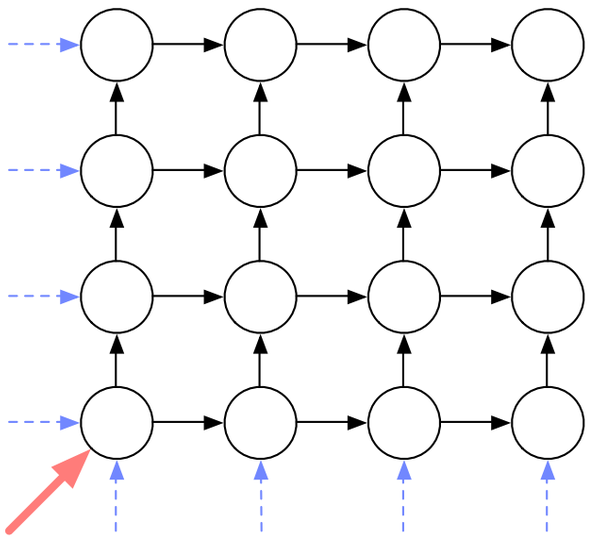
Starting from top left ({first, first})



Starting from top right ({first, last})



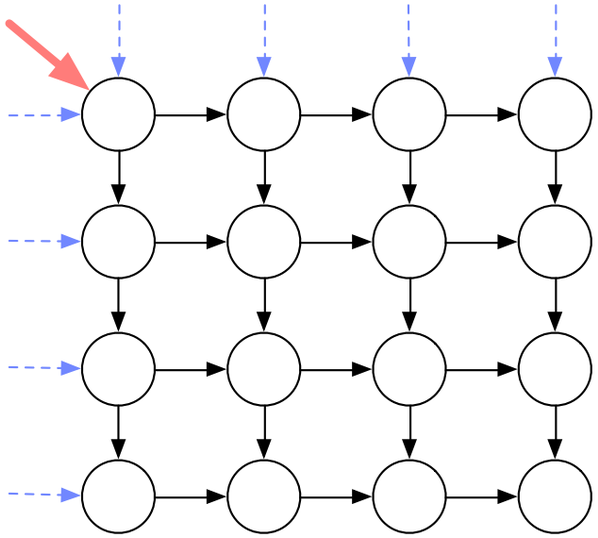
Starting from bottom right ({last, last})



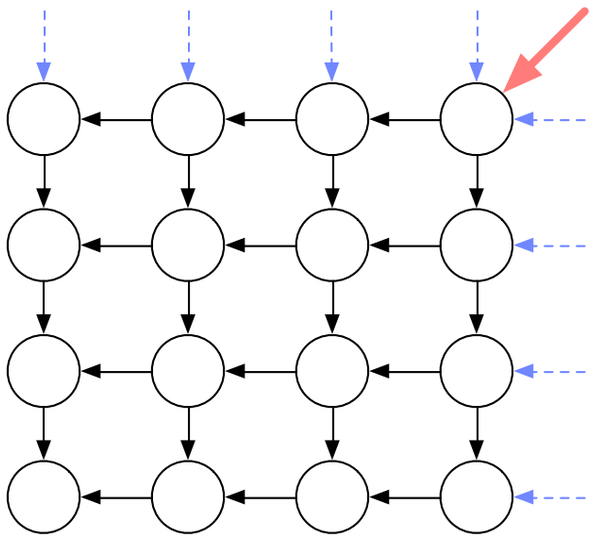
Starting from bottom left ({last, first})

Parasol 2D Wavefront Skeleton

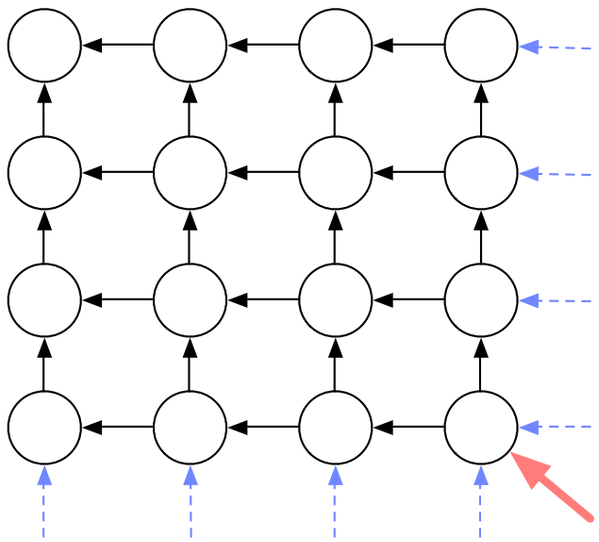
- In STAPL they are defined as:



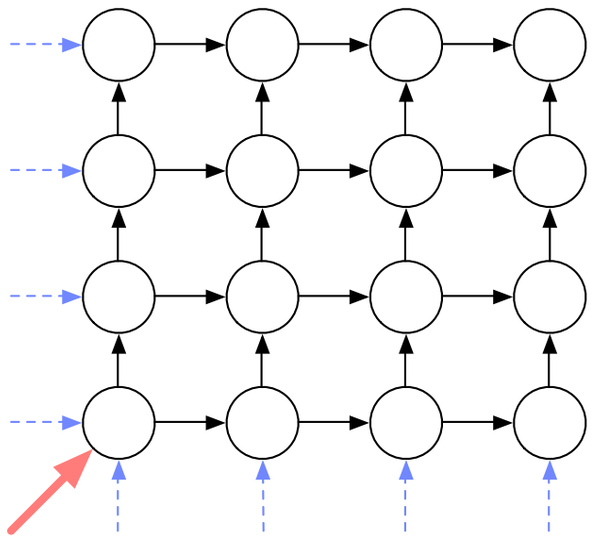
skeletons::wavefront(op, {{first, first}})



skeletons::wavefront(op, {{first, last}})



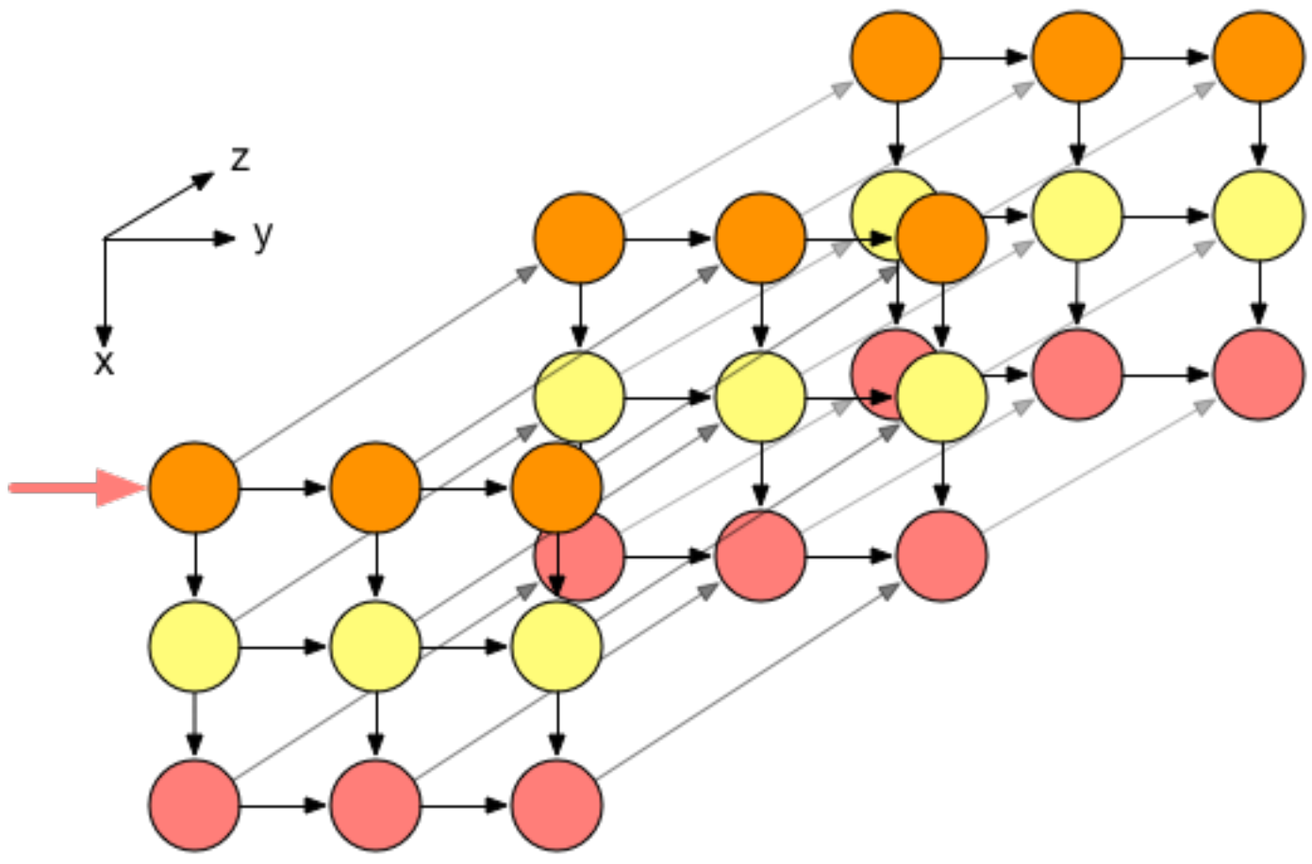
skeletons::wavefront(op, {{last, last}})



skeletons::wavefront(op, {{last, first}})

Parasol 3D Wavefront Skeleton

- Defined similarly
- Parameterized based on the corner to start from



wavefront(op, {{first, first, first}})

Parasol 3D Wavefront Skeleton Example

- Implementation in STAPL

```
// Definition in the STAPL Skeleton Framework
template <typename Op, int i>
auto wavefront(Op op, std::array<wavefront_direction, i> corners) {
  return elem<input_span, doacross>(wavefront_pd(op, corners));
}

// Assuming the input is provided in the input_view, you need to
// create your boundary views
auto size = input_view.size();
auto boundary_view1 = functor_view(size, your_generator(direction0));
auto boundary_view2 = functor_view(size, your_generator(direction1));
auto boundary_view3 = functor_view(size, your_generator(direction2));

// Finally, execute the wavefront skeleton
algorithm_executor exec;
exec.execute(wavefront(my_wavefront_operation(),
                       {{first, first, first}}),
             boundary_view1, boundary_view2, boundary_view3,
             input_view);
```

Parasol Kernel ZGD

- Reference Implementation

```
Kernel_ZGD::sweep(args...)
{
  for (int k = extent.start_k; k != extent.end_k; k += extent.inc_k)      // #1
    for (int j = extent.start_j; j != extent.end_j; j += extent.inc_j)    // .
      for (int i = extent.start_i; i != extent.end_i; i += extent.inc_i) // .
        for (int group = 0; group < num_groups; ++group)                  // #2
          for (int d = 0; d < num_directions; ++d)                        // #3
            diamond_difference_wf(...);
}
```

- STAPL Implementation

```
Kernel_ZGD::sweep(args...)
{
  auto kernel_ZGD_skeleton = wavefront(zip(zip(diamond_difference_wf(...)),
                                           find_corners<ZGD>(...));
  exec.execute(kernel_skeleton, input_views...);
}
```

Parasol All Kernels

```
auto kernel_ZGD_skeleton = wavefront(zip(zip(diamond_difference_wf(...)),
                                         find_corners<ZGD>(...)));

auto kernel_ZDG_skeleton = wavefront(zip(zip(diamond_difference_wf(...)),
                                         find_corners<ZGD>(...)));

auto kernel_DZG_skeleton = zip(wavefront(zip(diamond_difference_wf(...)),
                                           find_corners<ZGD>(...)));

auto kernel_DGZ_skeleton = zip(zip(wavefront(diamond_difference_wf(...),
                                             find_corners<ZGD>(...))));

auto kernel_GZD_skeleton = zip(wavefront(zip(diamond_difference_wf(...)),
                                           find_corners<ZGD>(...)));

auto kernel_GDZ_skeleton = zip(zip(wavefront(diamond_difference_wf(...),
                                             find_corners<ZGD>(...))));
```

Parasol Kripke Algorithm

- The full algorithms is defined as

```
// kernel computation
auto kernel_ZGD_skeleton = wavefront(zip(zip(diamond_difference_wf(...)),
                                         find_corners<ZGD>(...)));
for (int iteration = 0; iteration < num_iterations; ++iteration) {
  exec.execute(compose(zip(LTimes(...)), zip(LPlusTimes(...))),
               input_view...);

  for (int group_set = 0; group_set < num_group_sets; ++group_set) {
    exec.execute(doall(kernel_ZGD_skeleton,
                       [&](size_t size){return num_direction_sets}),
                 input_view...)
  }
}
```

Parasol

Kripke

- Kripke Overview
- Designing the STAPL implementation
  - Data structures
  - Skeleton based algorithms
  - Views
- Current work

Parasol

Algorithms and Views

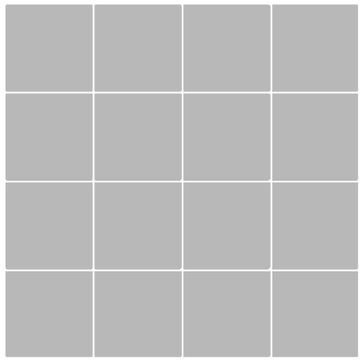
- Data structure inside zoneset is 5-dimensional
  - Separate 3 dimensions for wavefront
  - Separate others into 1D zips
- Modify how a container looks to conform with what the algorithm expects
  - View the 5D container in a manner conformant to the kernel nest
  - Auxiliary 4D structure (sigt) where direction dimension is aggregated

Multiarray View

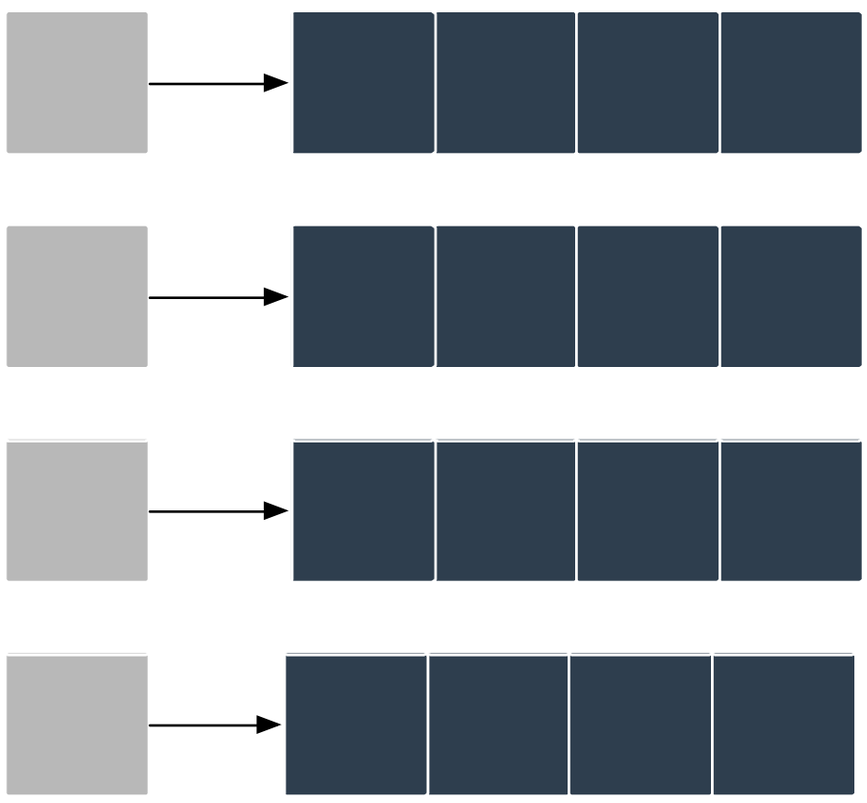
- The main container in Kripke is stapl::multiarray
- multiarray_view is a view over the multiarray container
- Forms the basis of the following complex views
  - Slices view
  - Extended view
  - Functor view

Rows and columns

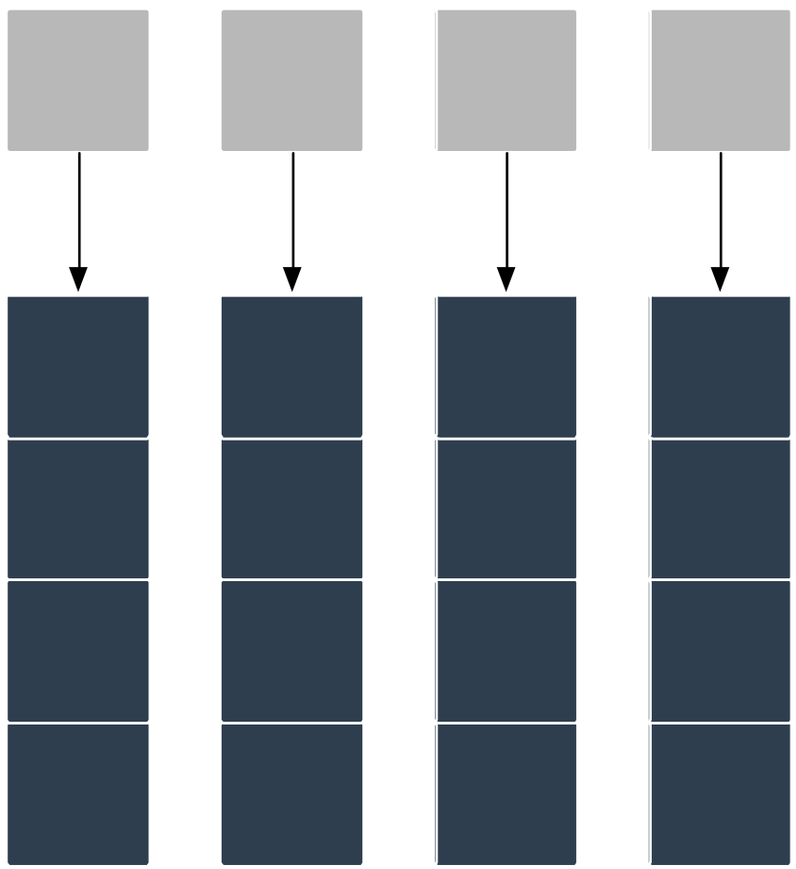
- For a 2D matrix, produce a collection whose elements are rows or columns



`rows(A)`

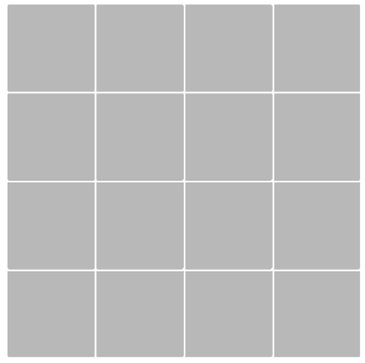


`columns(A)`

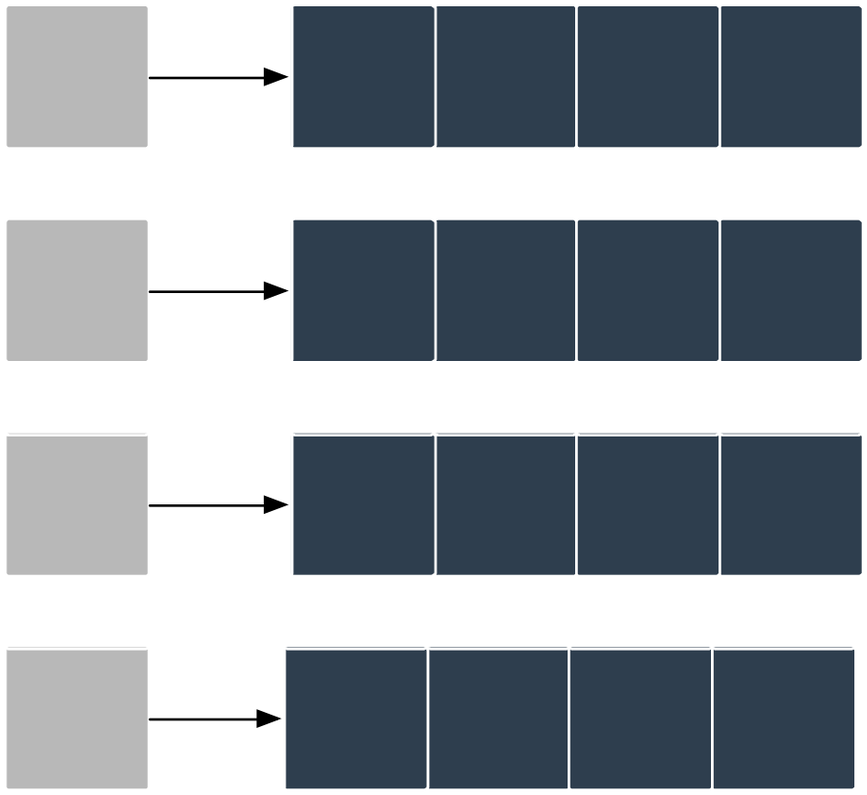


Slices View

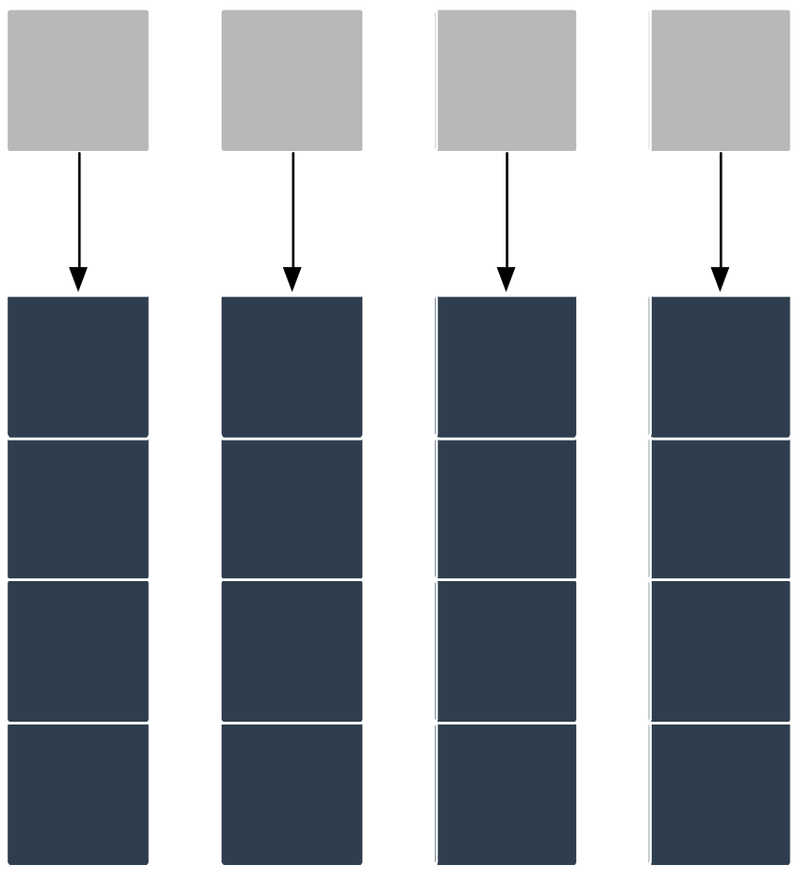
- View that produce subviews
  - Each subview is a single slice along given dimension



`make_slices_view<0>(vw)`



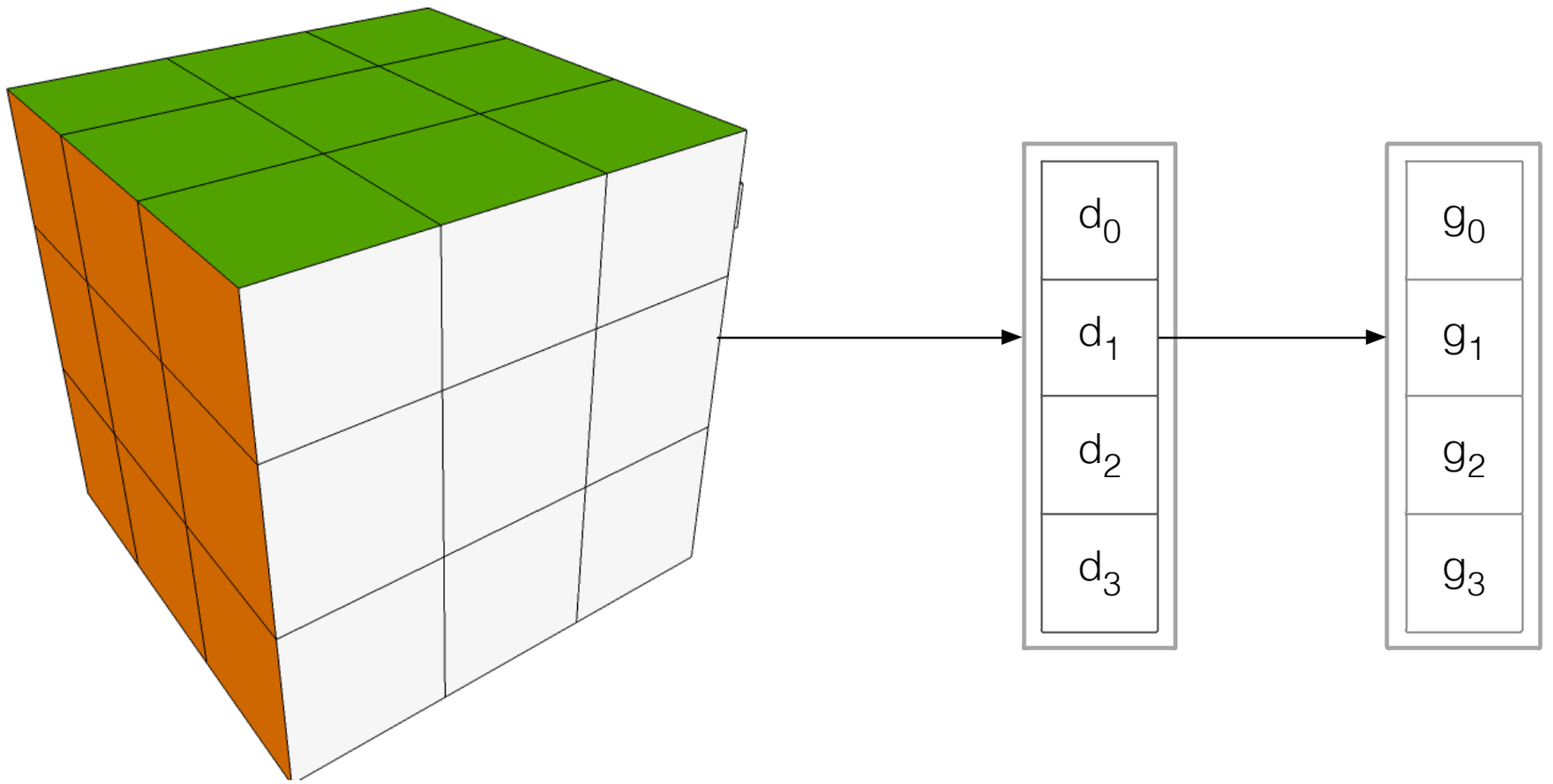
`make_slices_view<1>(vw)`



Parasol

Slices in Kripke (ZGD)

(i, j, k) g d



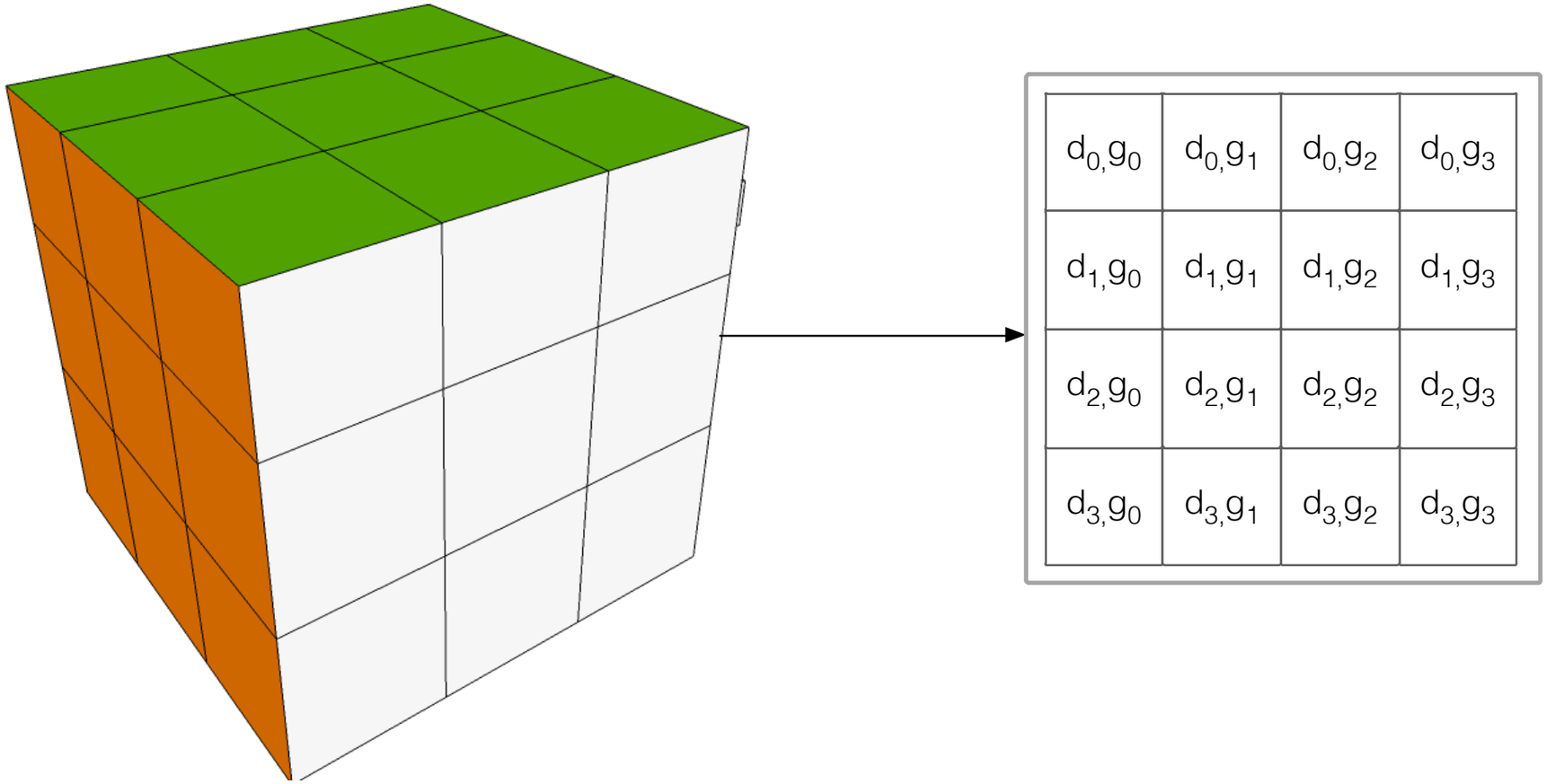
wavefront zip zip

Slices in Kripke (ZGD)

canonical order is i,j,k,d,g

```
make_slices_view<0,1,2>(
  rhs
)
```

(i, j, k) (g,d)

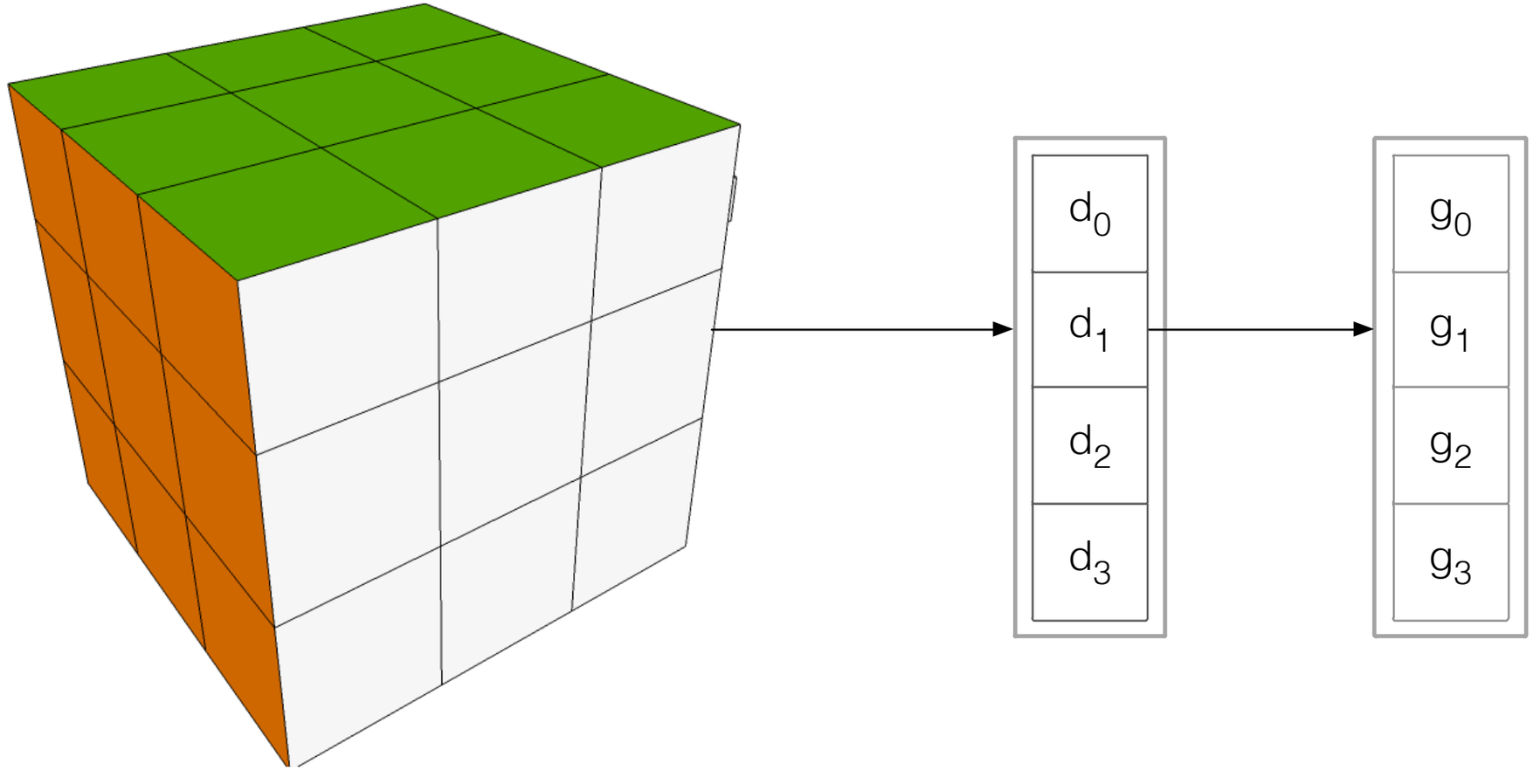


Slices in Kripke (ZGD)

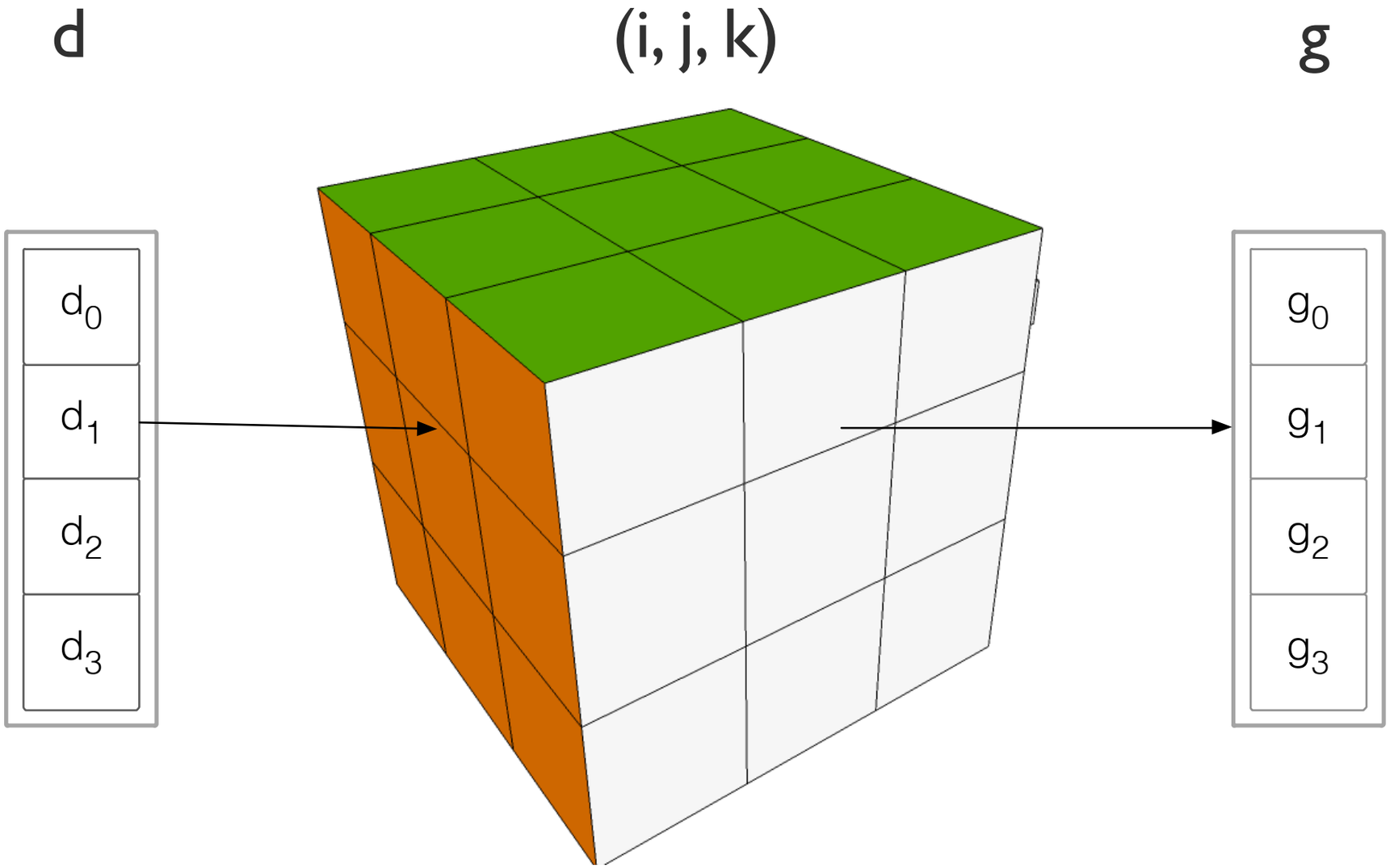
canonical order is i,j,k,d,g

```
make_slices_view<0,1,2>(
  make_slices_view<0,1,2,4>(rhs)
)
```

(i, j, k) g d



Slices in Kripke (DZG)

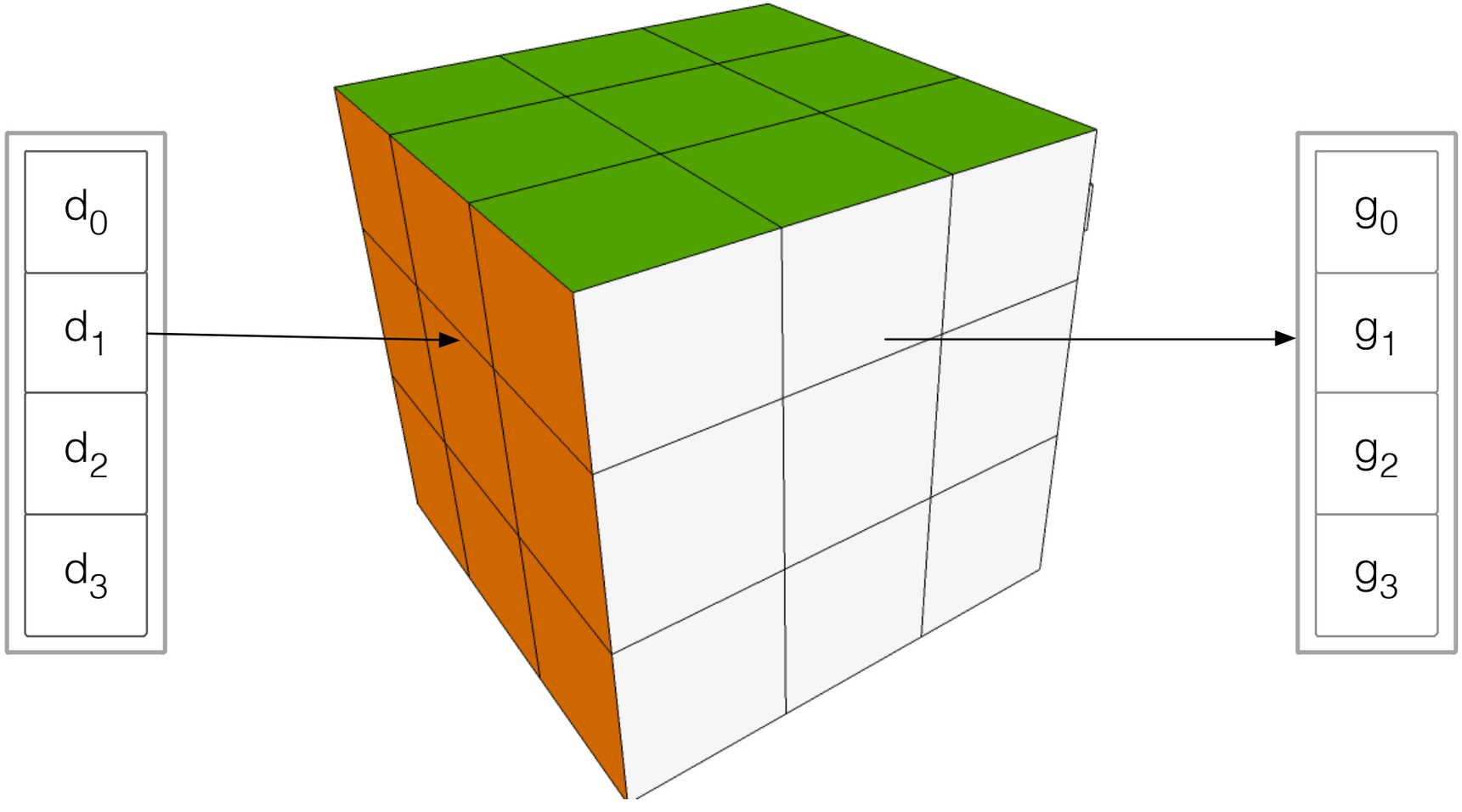


Slices in Kripke (DZG)

canonical order is i,j,k,d,g

```
make_slices_view<3>(
  make_slices_view<0,1,2,3>(rhs)
)
```

d (i, j, k) g

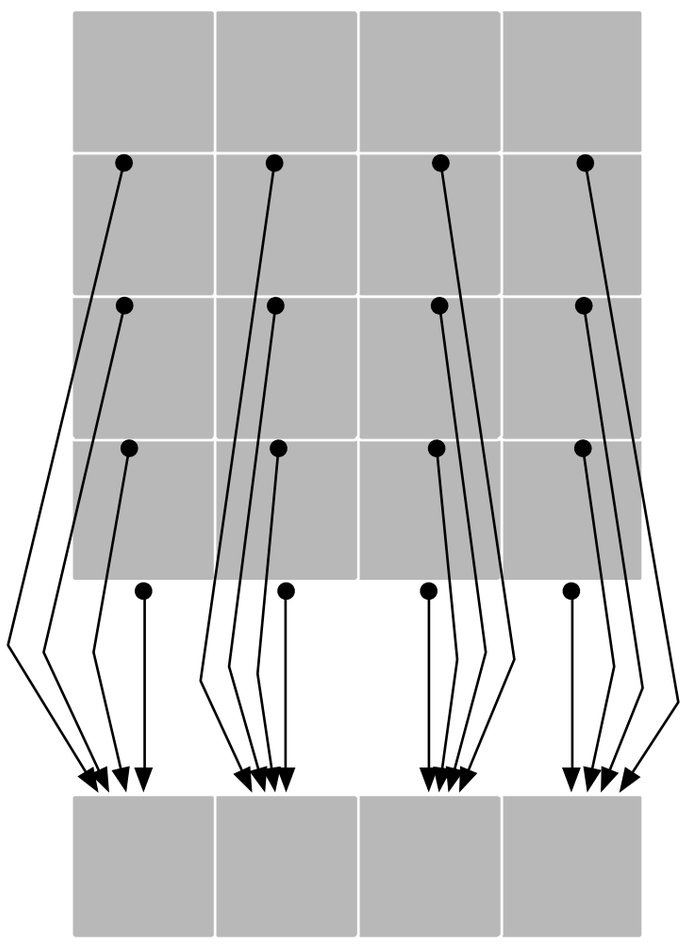


Extended View

- A view that extends a single dimension of a view to a given size
- Scalar expansion

```
make_extended_view<0>(vw, 4)
```

multiarray<1>(4)



Extended View in Kripke

- sigt is a container where the flux is aggregated for each direction
- The dimensions are (i,j,k,g), but we want to extend the direction dimension d
  - Map 4D container to 5D view using expansion
- All views are then conformable to each other

```
ext_sigt = make_extended_view<3>(
  sigt, num_directions
);

skeleton(rhs, psi, ext_sigt)
```

Functor View

- A view that evaluates a function when read
  - Collection of elements is closed-form solution

```
make_functor_view(tuple(4,4), f)
```

$f_{0,0}$	$f_{0,1}$	$f_{0,2}$	$f_{0,3}$
$f_{1,0}$	$f_{1,1}$	$f_{1,2}$	$f_{1,3}$
$f_{2,0}$	$f_{2,1}$	$f_{2,2}$	$f_{2,3}$
$f_{3,0}$	$f_{3,1}$	$f_{3,2}$	$f_{3,3}$

ZGD Kernel

```
skeleton = …
containers = …
```

```
execute(skeleton,

)
```

ZGD Kernel

```
skeleton = …
containers = …

ext_sigt = make_extended_view<3>(sigt, num_directions)
```

```
execute(skeleton,

)
```

ZGD Kernel

```
skeleton = …
containers = …

ext_sigt = make_extended_view<3>(sigt, num_directions)

psi_slices = make_slices_view<0,1,2>(
  make_slices_view<0,1,2,4>(psi)
)

rhs_slices = // same as above with rhs
ext_sigt_slices = // same as above with ext_sigt



execute(skeleton,

  psi_slices, rhs_slices, ext_sigt_slices, output
)
```

ZGD Kernel

```
skeleton = …
containers = …

ext_sigt = make_extended_view<3>(sigt, num_directions)

psi_slices = make_slices_view<0,1,2>(
  make_slices_view<0,1,2,4>(psi)
)

rhs_slices = // same as above with rhs
ext_sigt_slices = // same as above with ext_sigt

boundary_view0 = make_functor_view(dims, boundary_cond)

execute(skeleton,
  boundary_view0, boundary_view1, boundary_view2,
  psi_slices, rhs_slices, ext_sigt_slices, output
)
```

Implementation of views

- Create new views and modify components
- Views are defined as
  - C: partitioned collection of elements
  - D: domain of the view
  - F: function to map from the view to the container
  - O: operations

Implementation of views

- Slices view
  - Modify domain to include dimensions that are being sliced
  - Mapping function to project from n-d to m-d
- Extended view
  - Modify domain to include new dimension
  - Mapping function ignores indices in that dimension

Parasol Kripke

- Kripke Overview
- Designing the STAPL implementation
  - Data structures
  - Skeleton based algorithms
  - Views
- Current work

Parasol Status

- Integration of multiarray<multiarray<>> for data structures.
  - Uniform, spatial decomposition for outer container of zonesets.
  - Varying composition of subset of dimensions for inner container, based on specified kernel.

- Integrated View composition for kernels
  - slices_view and extended_view to conform data to kernel nesting.

- Wavefront skeleton written and integrated both across zonesets and within a zoneset.

Parasol Current Work

- Skeleton Composition
- Performance Tuning
- Investigate further extensions, configurations.

Questions?