

## DIY Parallel Data Analysis



Image courtesy pigtimes.com

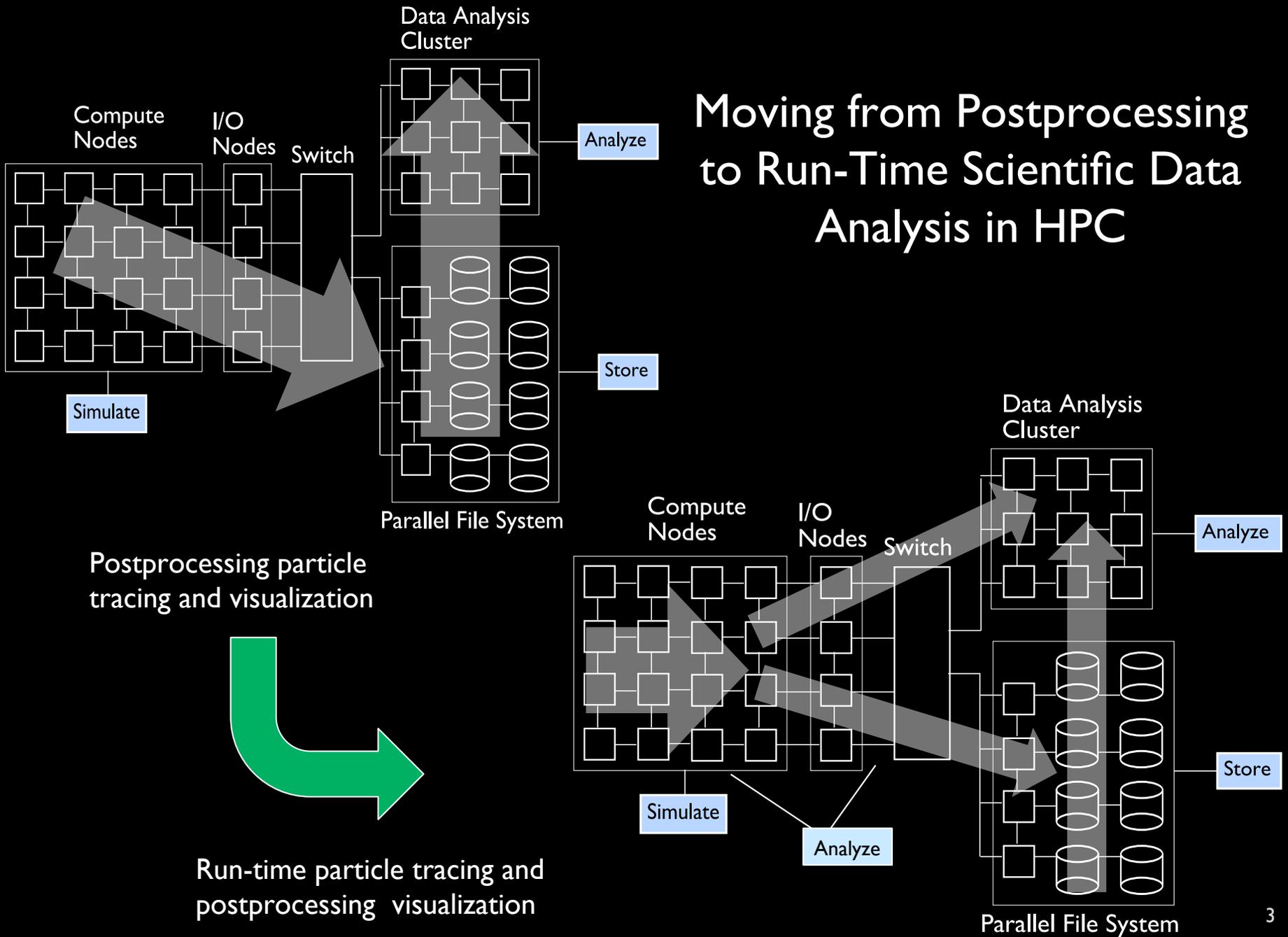
Tom Peterka

[tpeterka@mcs.anl.gov](mailto:tpeterka@mcs.anl.gov)

Mathematics and Computer Science Division

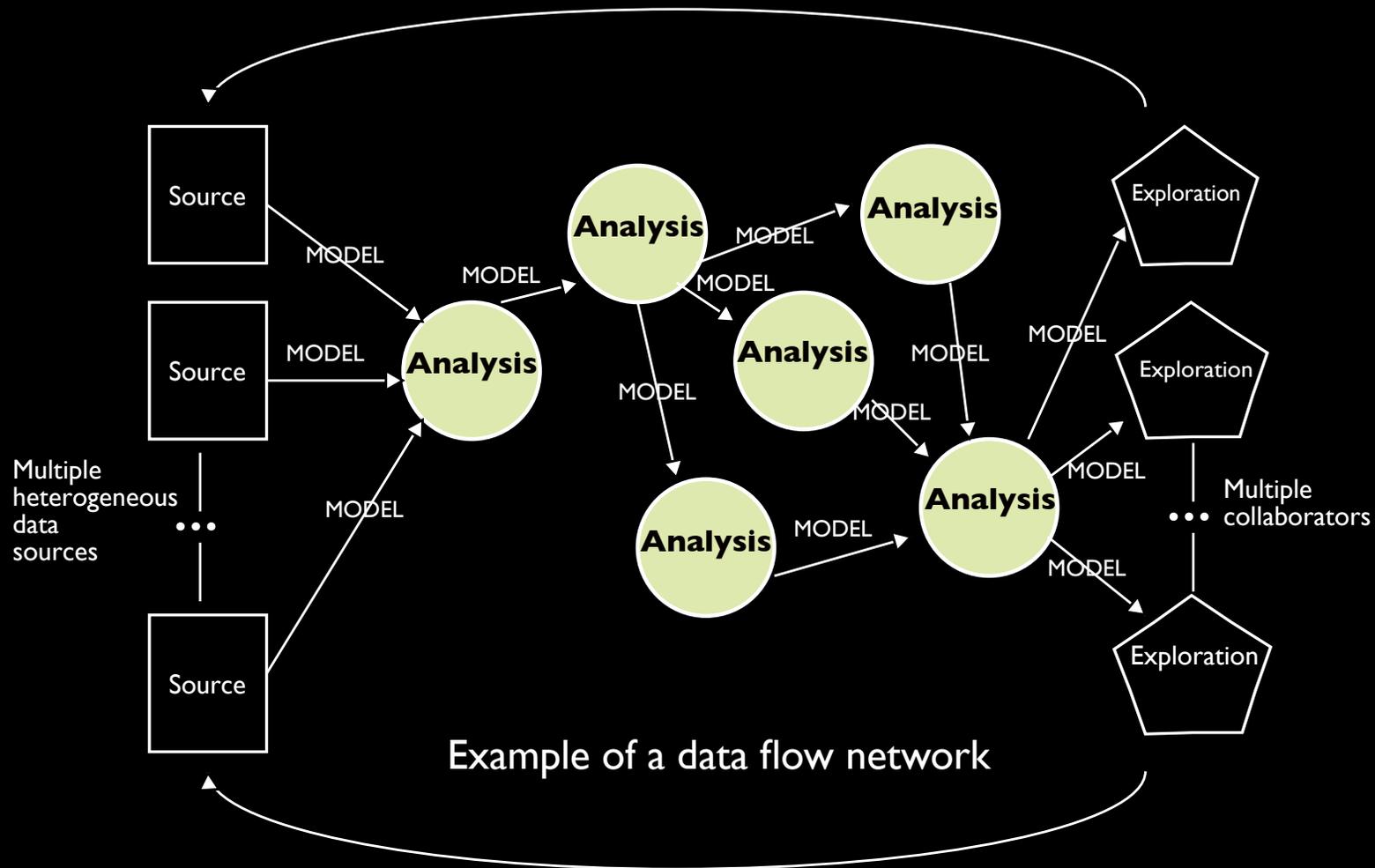
# Preliminaries

# Moving from Postprocessing to Run-Time Scientific Data Analysis in HPC

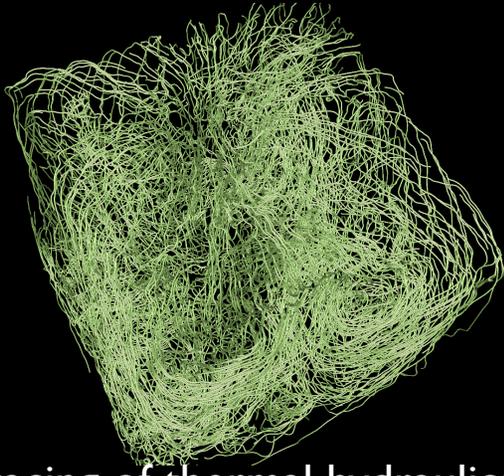


# Definition of Data Analysis

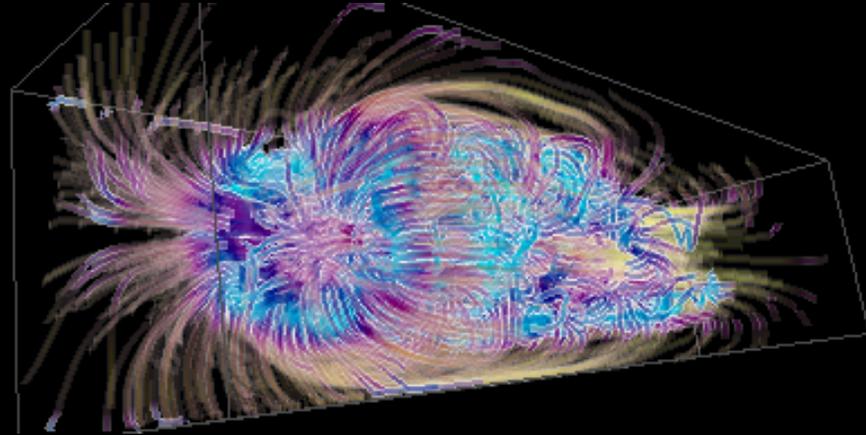
- Any data transformation, or a network or transformations.
- Anything done to original data beyond its original generation.
- Can be visual, analytical, statistical, or data management.



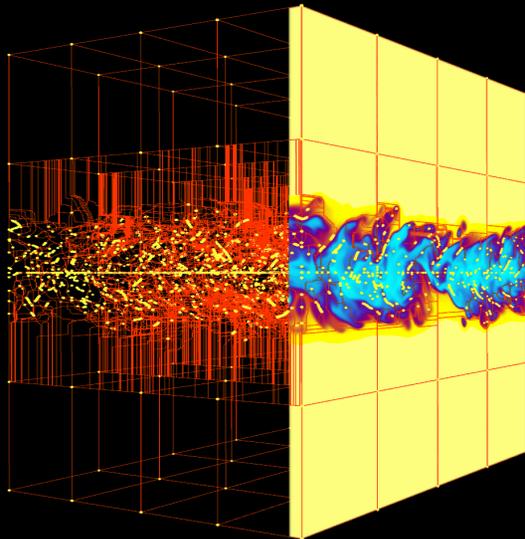
# Examples of Data Analysis



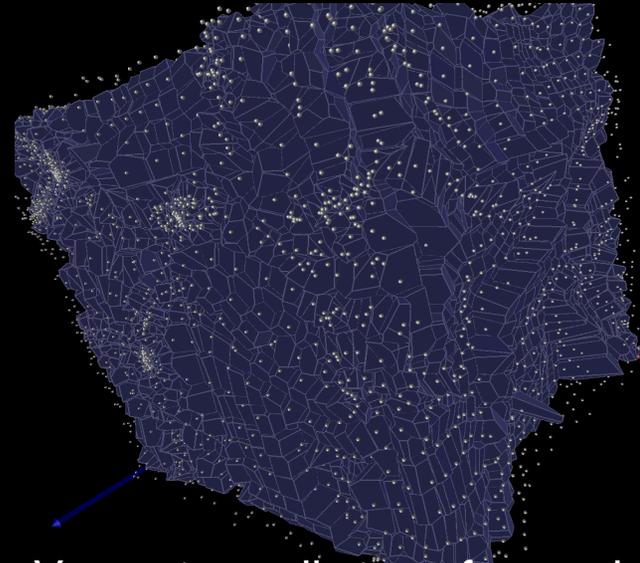
Particle tracing of thermal hydraulics flow



Information entropy analysis of astrophysics



Morse-Smale complex of combustion



Voronoi tessellation of cosmology

... and infinitely many more

# Scientific Data Analysis Today

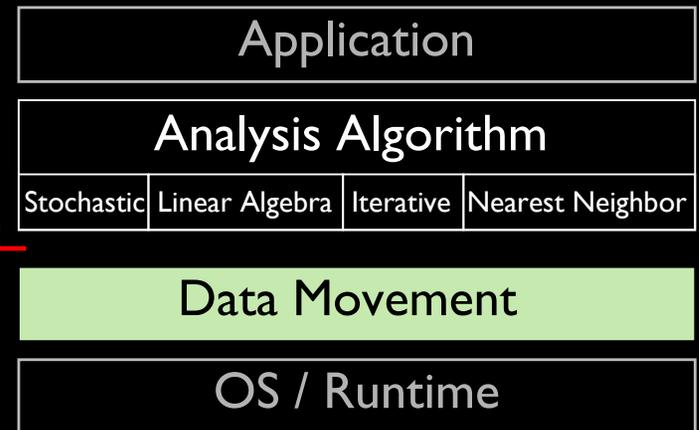
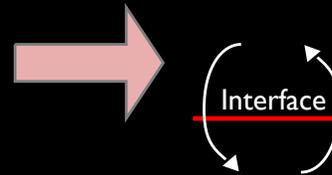
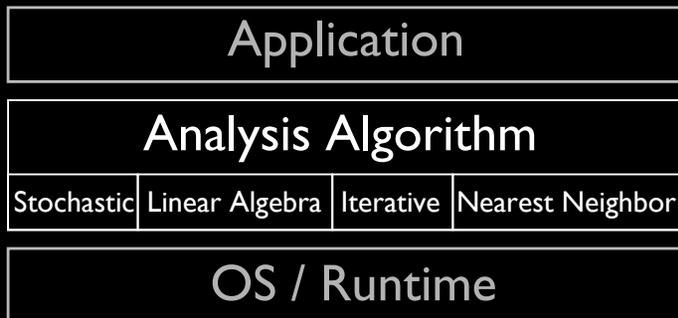
- Big science = big data, and
  - Big data analysis => big science resources
- Data analysis is data intensive.
  - Data intensity = data movement.
- Parallel = data parallel (for us)
  - Big data => data decomposition
  - Task parallelism, thread parallelism, while important, are not part of this work
- Most analysis algorithms are not up to the challenge
  - Either serial, or
  - Communication and I/O are scalability killers

# You Have Two Choices to Parallelize Data Analysis

By hand

or

With tools



```
void ParallelAlgorithm() {  
    ...  
    MPI_Send();  
    ...  
    MPI_Recv();  
    ...  
    MPI_Barrier();  
    ...  
    MPI_File_write();  
}
```

```
void ParallelAlgorithm() {  
    ...  
    LocalAlgorithm();  
    ...  
    DIY_Merge_blocks();  
    ...  
    DIY_File_write()  
}
```

# Executive Summary

DIY helps the user write data-parallel analysis algorithms.

## Main ideas and Objectives

- Large-scale parallel analysis for HPC
- Scientists, visualization researchers, tool builders
- In situ, coprocessing, postprocessing
- Data-parallel problem decomposition
- Scalable data movement algorithms

## Benefits

- Researchers can focus on their own work, not on parallel infrastructure
- Analysis applications can be custom
- Reuse core components and algorithms for performance and productivity

# Thirteen things you need for parallel data analysis

# #1: Separate Analysis Ops from Data Ops

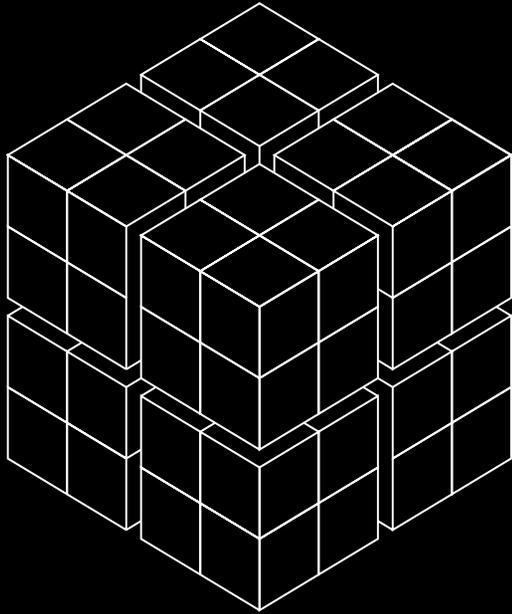
Analysis	Application	Application Data Model	Analysis Data Model	Analysis Algorithm	Communication	Additional
Particle Tracing	CFD	Unstructured Mesh	Particles	Numerical Integration	Nearest neighbor	File I/O, Domain decomposition, process assignment, utilities
Information Entropy	Astrophysics	AMR	Histograms	Convolution	Global reduction, nearest neighbor	
Morse-Smale Complex	Combustion	Structured Grid	Complexes	Graph Simplification	Global reduction	
Computational Geometry	Cosmology	Particles	Tessellations	Voronoi	Nearest neighbor	

You do this yourself

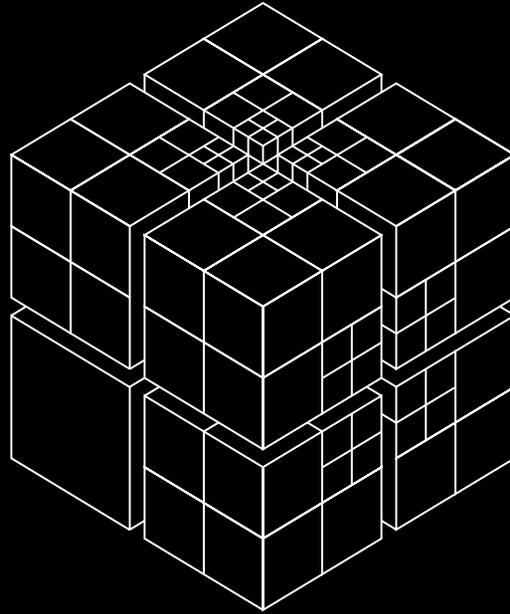
Can use serial libraries such as OSUFlow, Qhull, VTK  
(don't have to start from scratch)

DIY handles this

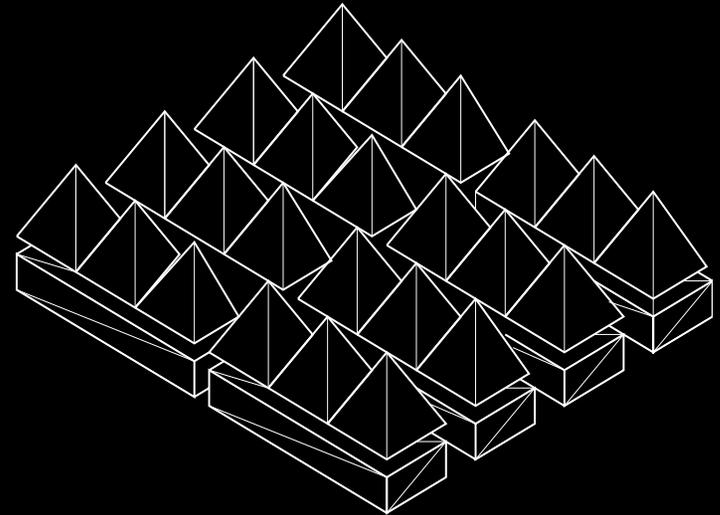
## #2: Group Data Items Into Blocks



Structured Grid



AMR Grid



Unstructured Mesh

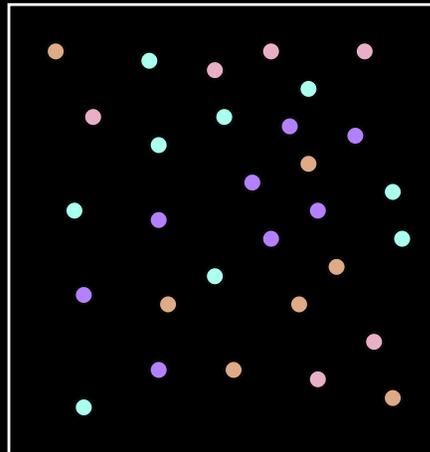
The block is DIY's basic unit of data decomposition. Original dataset is decomposed into generic subsets called blocks, and associated analysis items live in the same blocks. Blocks don't have to be "blocky." Any subdivision of data (eg., a set of graph nodes, a group of particles, etc.) is a block in DIY.

# #3: Support Multiple Domains

Uses:

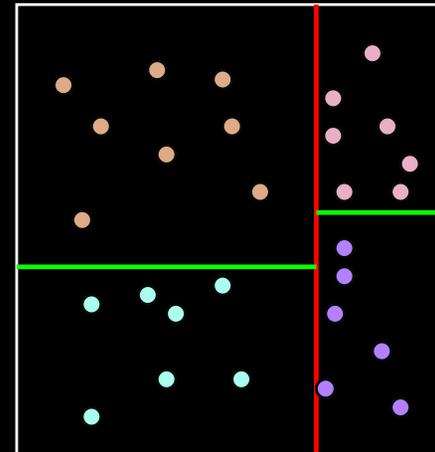
1. Organize input (upper right)
2. Second decomposition suited for particular analysis (lower right)
3. Comparing multiple unrelated data domains (not shown)

Original data  
Arbitrary decomposition



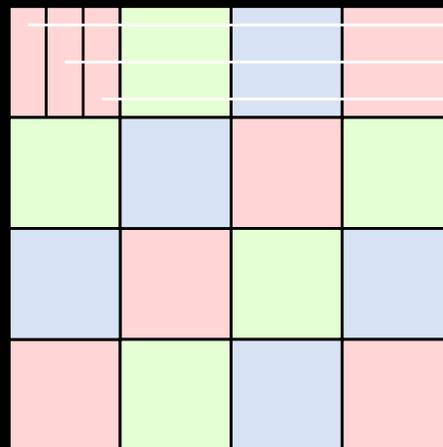
4 blocks indicated by color.  
No spatial locality assumed.

Kd-tree decomposition



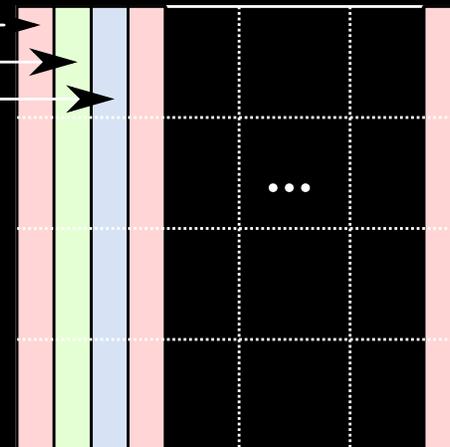
4 new blocks spatially contiguous  
and load balanced by number of  
objects in each.

Original block decomposition



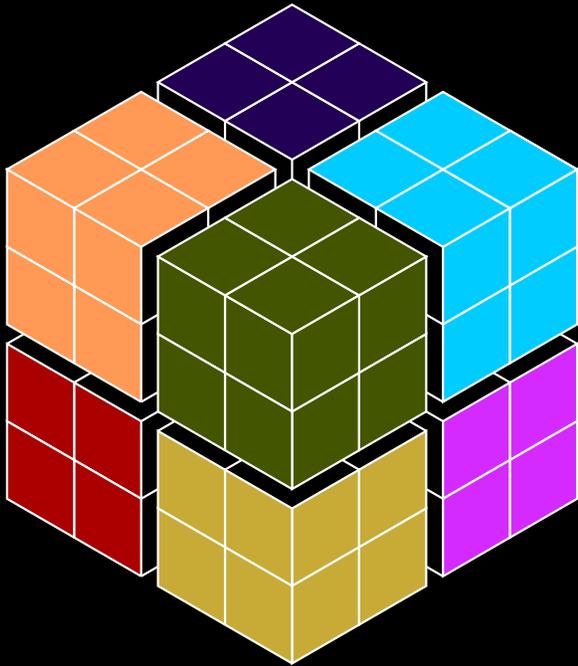
16 blocks, 3 procs indicated by color

Slab or pencil decomposition for FFT

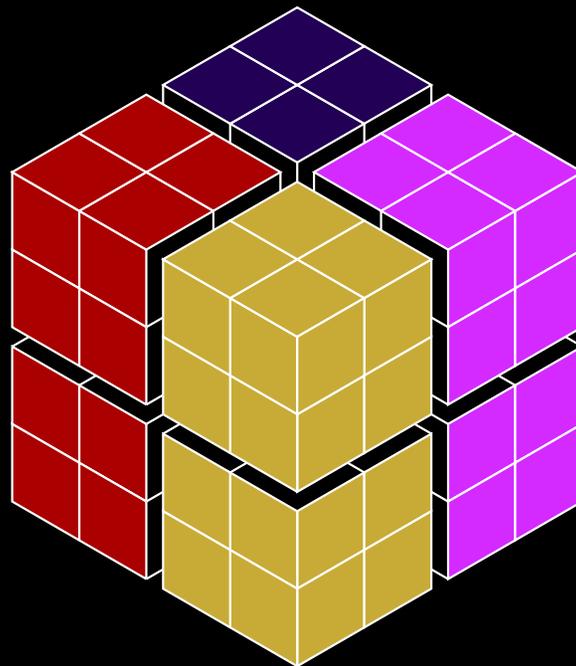


Need not be same number of blocks

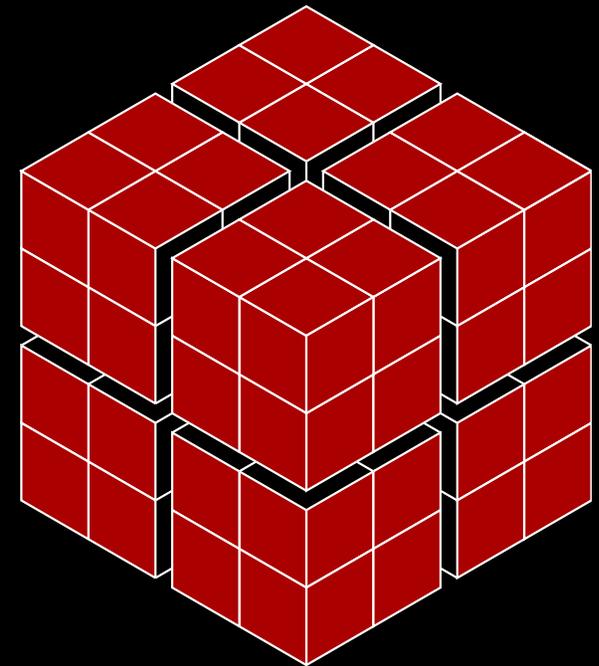
## #4: Distinguish Between Blocks and Processes



8 processes



4 processes

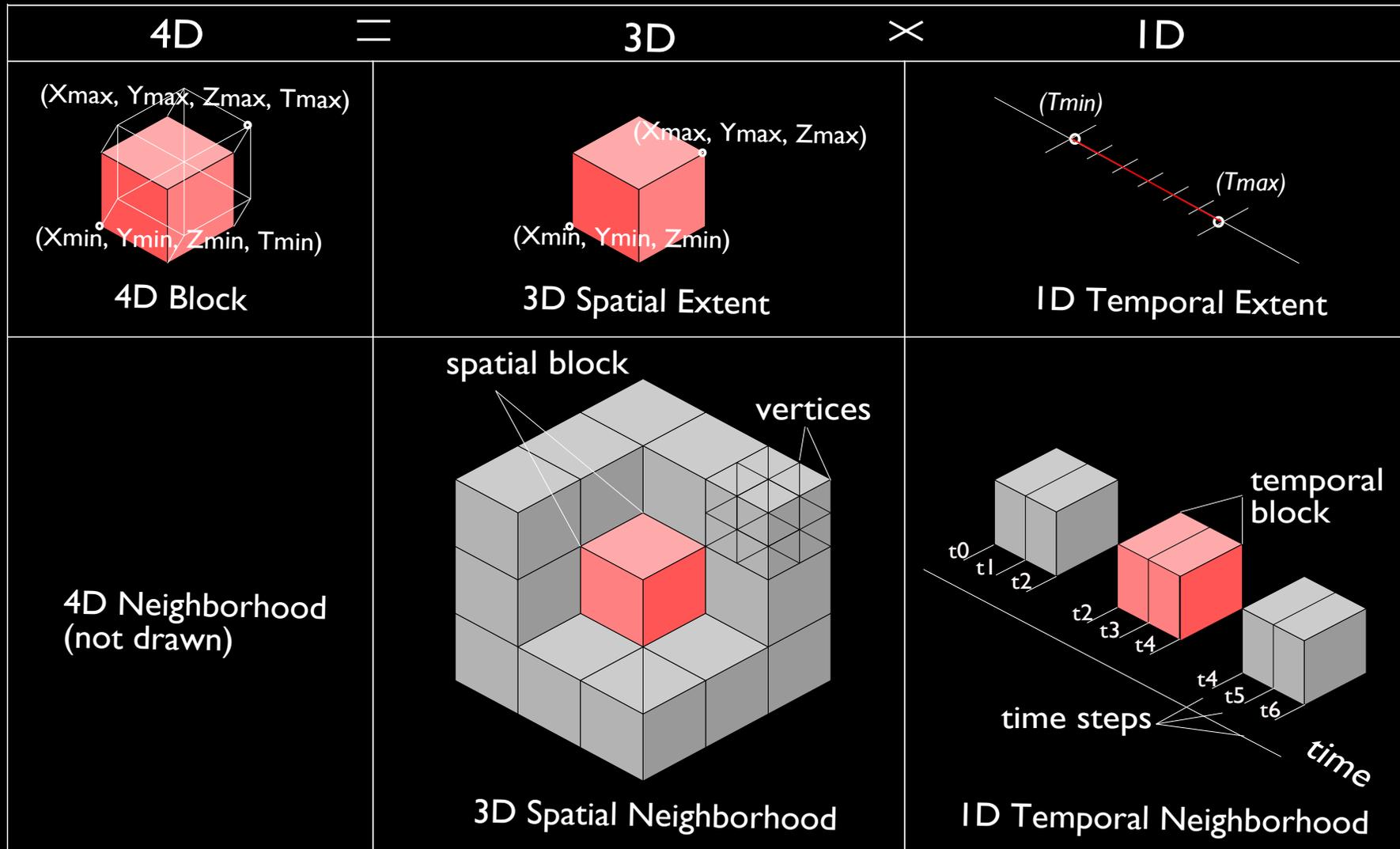


1 process

All data movement operations are per **block**; blocks exchange information with each other using DIY's communication algorithms. DIY manages and optimizes exchange between processes based on the process assignment. This allows for flexible process assignment as well as easy debugging.

# #5: Handle Time

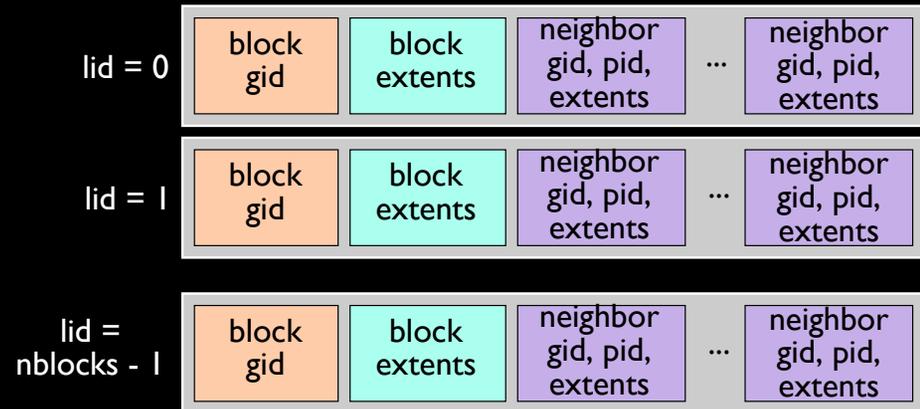
- Time often goes forward only
- Usually do not need all time steps at once



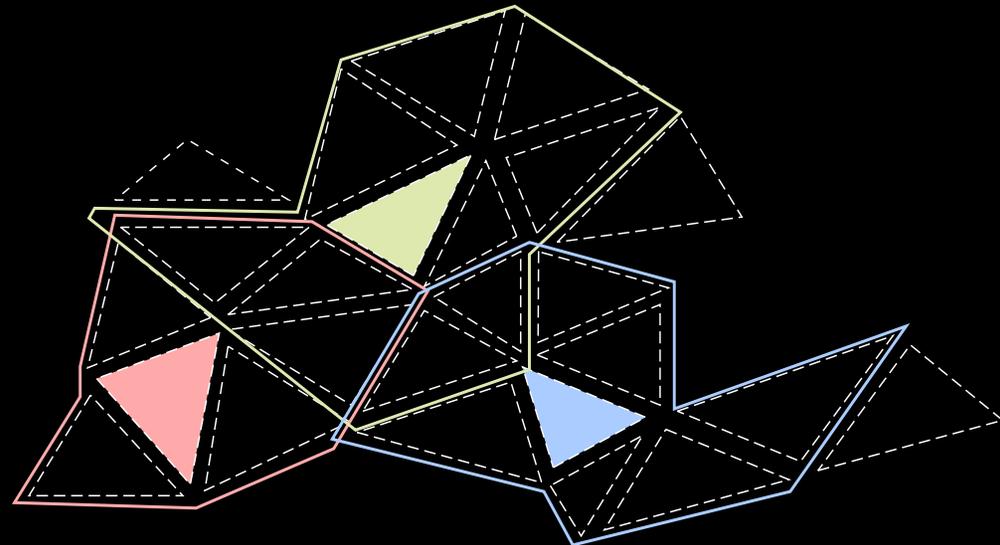
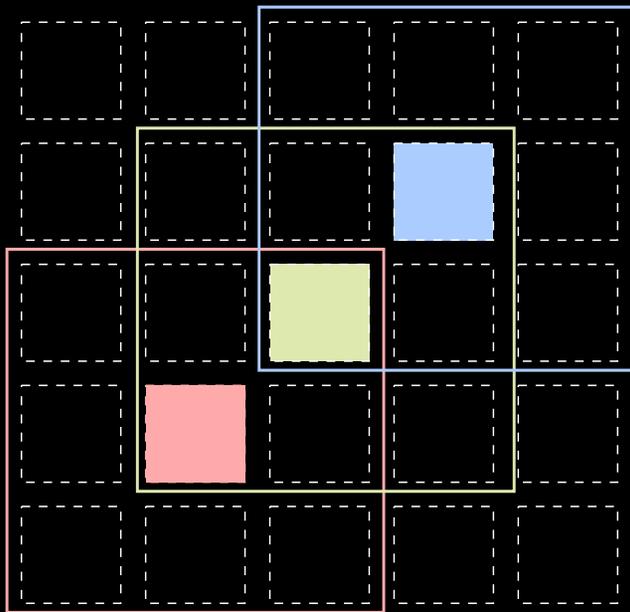
Hybrid 3D/4D time-space decomposition. Time-space is represented by 4D blocks that can also be decomposed such that time blocking is handled separately.

# #6: Group Blocks into Neighborhoods

- Limited-range communication
- Allow arbitrary groupings
- Distributed, local data structure and knowledge of other blocks (not master-slave global knowledge)



gid = global block identification  
 lid = local block identification  
 pid = process identification



Two examples of 3 out of a total of 25 neighborhoods

# #7 Make Communication Fun

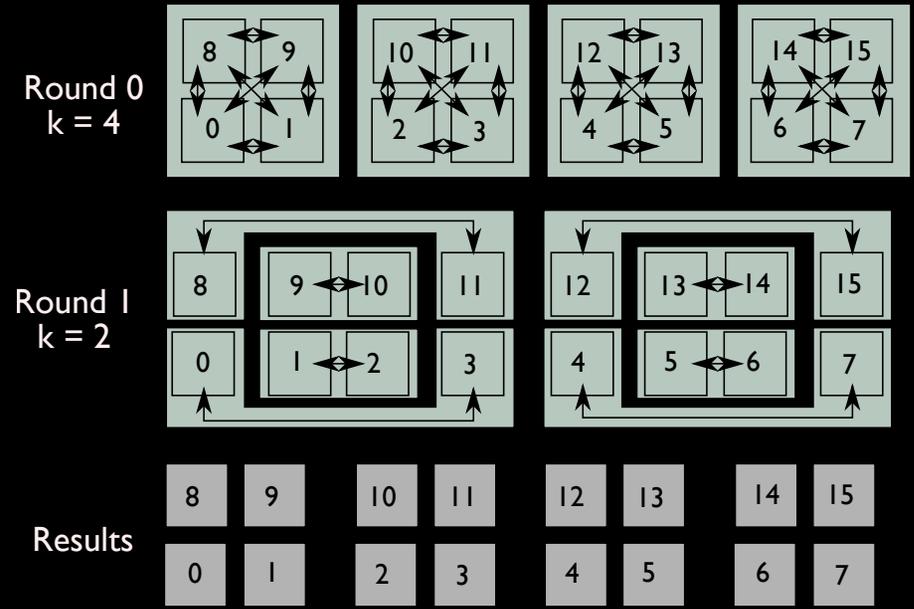
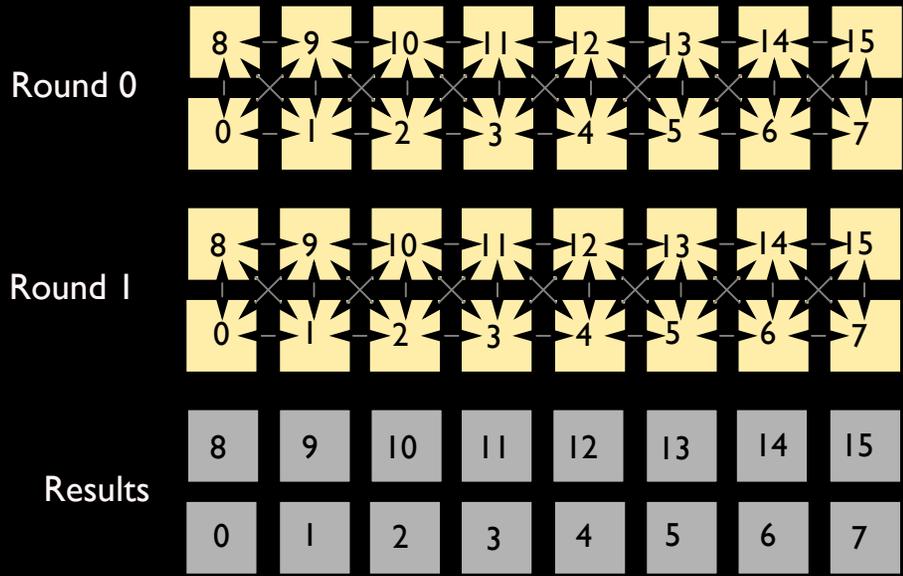
	Analysis	Communication	
Regular	Sort-Last Rendering	Swap-Based Reduction	Homogeneous Data
	Morse-Smale Complex	Merge-Based Reduction	
	Information Entropy	Merge-Based Reduction	Heterogeneous Data
Semi Regular	Particle Tracing	Neighborhood Exchange	
	Voronoi Tessellation	Neighborhood Exchange	
Irregular	Graph layout	Send-Receive	

Many different analysis operations share a small set of communication patterns. These communication kernels together with supporting utilities for decomposition and I/O can be encapsulated, optimized, and reused. DIY provides 3 efficient scalable communication algorithms on top of MPI. May be used in any combination.

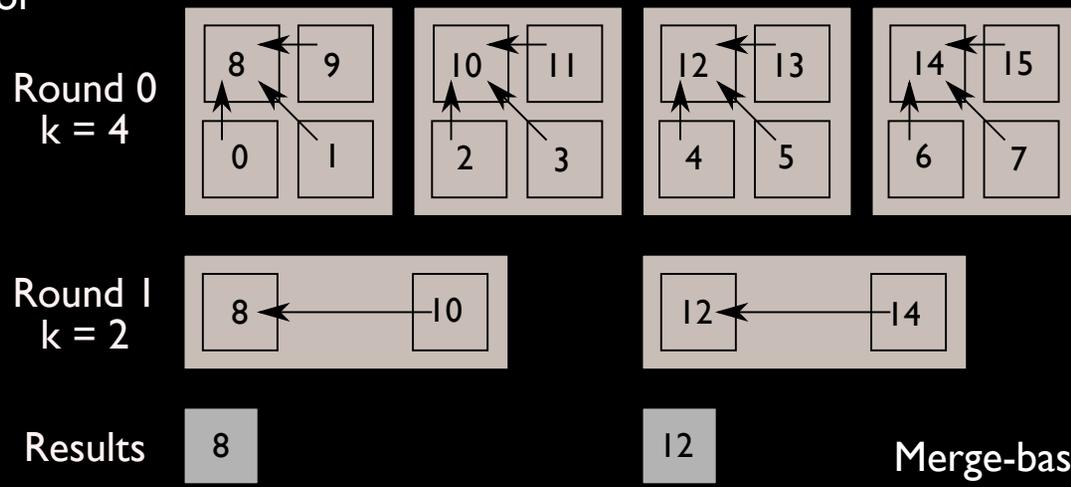
Factors for selecting communication algorithm:

- associativity
- number of iterations
- data size vs. memory size

# 3 Communication Patterns



## Nearest neighbor

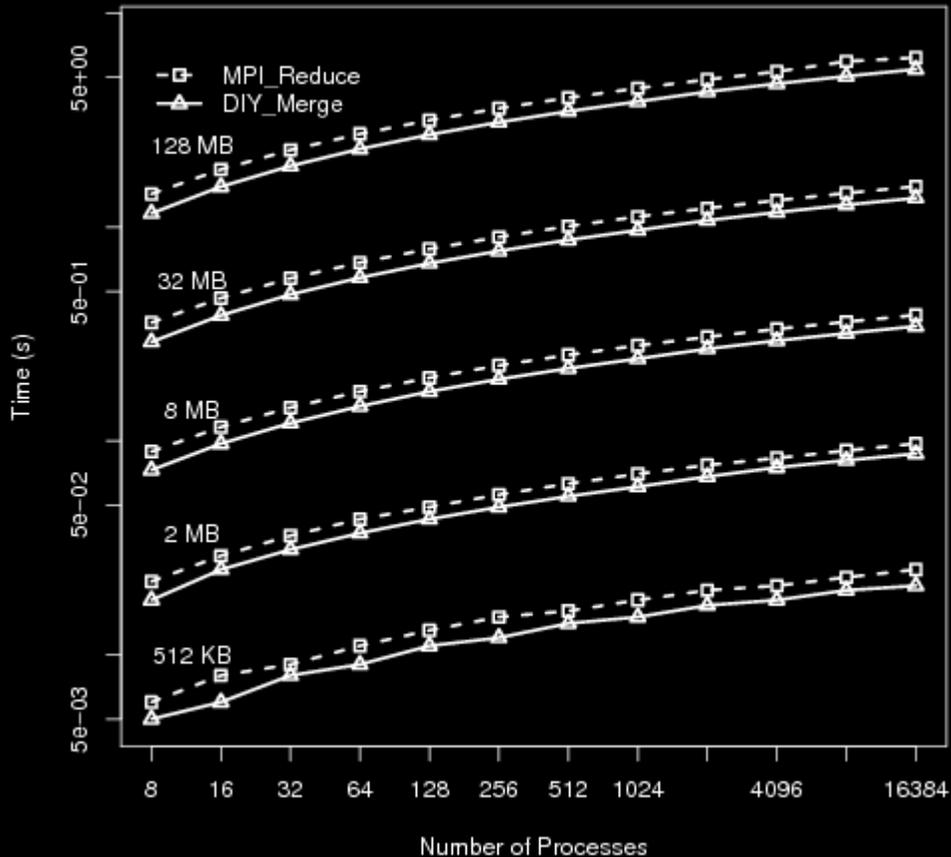


## Swap-based reduction

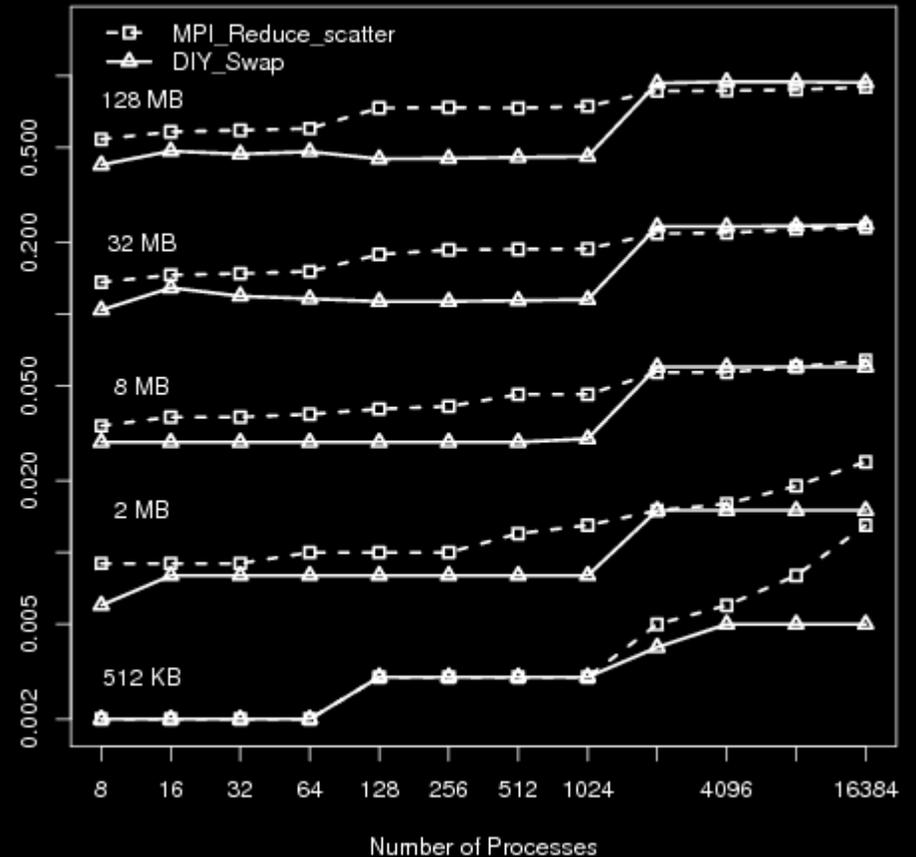
## Merge-based reduction

# Communication Performance Benchmarks

## Merge No-op Performance



## Swap No-op Performance



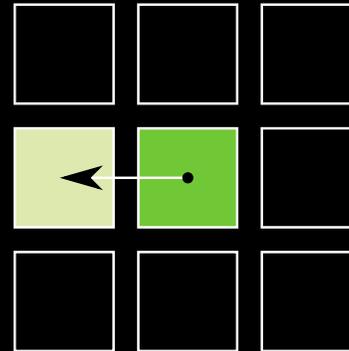
Communication time only for our merge algorithm compared with MPI's reduction algorithm (left) and our swap algorithm compared with MPI's reduce-scatter algorithm (right).

# Different Neighborhood Communication Patterns

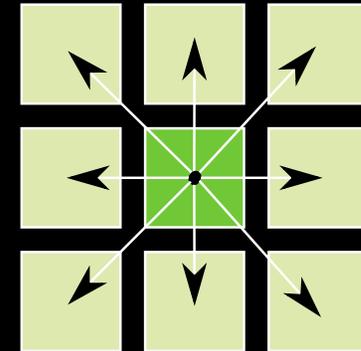
DIY provides point to point and different varieties of collectives within a neighborhood via its enqueue\_item mechanism. Items are enqueued and subsequently exchanged (2 steps).

## How to enqueue items for neighbor exchange

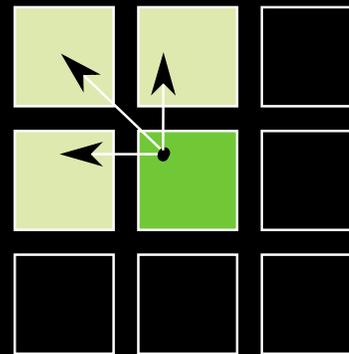
- DIY offers several options
- Send to a particular neighbor or neighbors, send to all nearby neighbors, send to all neighbors
- Support for periodic boundary conditions involves tagging which neighbors are periodic and calling user-defined transform on objects being sent to them



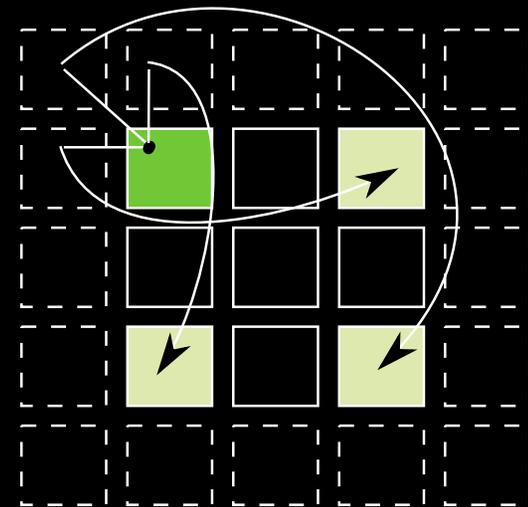
Send to only specific neighbors, indicated in various ways



Send to all neighbors



Send to all neighbors near enough to a target point



Support for wraparound neighbors (periodic boundary conditions)

# Adjustable Synchronization Communication Algorithm

**Wait factor:** the fraction of items for which to wait is adjustable. Typically we use 0.1 (wait for 10% of pending items to arrive in each round).

```
for (blocks in my neighborhood) {  
  
    pack and send messages of block IDs and  
    particle counts  
    pack and send messages of particles  
  
}  
wait for enough IDs and counts to arrive  
for (IDs and counts that arrived) {  
    receive particles  
  
}
```

# Stress Test: Number of Items Exchanged

Particle tracing usually exchanges few particles between blocks; eg., previous results were between 8 and 256 particles per block. We also benchmarked our neighbor exchange algorithm for much greater number of items exchanged.

Small item counts at various process counts

# Items	Bytes/Item	Total Bytes	# Procs	Exchange Time (s)
64	20	1 K	32	0.018
			128	0.028
			512	0.028
256	20	5 K	32	0.064
			128	0.097
			512	0.098
1 K	20	20 K	32	0.235
			128	0.354
			512	0.357

Large item counts at 512 processes

# Items	Bytes/Item	Total Bytes	# Procs	Exchange Time (s)
4 K	20	80 K	512	1.358
16 K		320 K	512	5.507
64 K		1 M	512	22.083
256 K	20	5 M	512	90.238
1 M		20 M	512	351.068

One aggregated item at 512 processes

# Items	Bytes/Item	Total Bytes	# Procs	Exchange Time (s)
1	20 M	20 M	512	0.223

Platform: IBM Blue Gene/Q

Conclusion: Exchanging up to a few thousand small items performs well. Beyond that number, the user should aggregate small items into a larger item prior to exchanging.

## #8: Define Custom Data Models

### HACC (cosmology) Data Model

```
int num_particles;  
float *xx, *yy, *zz;  
float *vx, *vy, *vz;  
float *phi;  
int64_t pid;  
uint16_t mask;
```

Corollary: analysis X data  
model  $\neq$  analysis Y data  
model

### Tess (voronoi tessellation) Data Model

```
float mins[3];  
float maxs[3];  
int num_verts;  
int num_cells;  
double *verts;  
int *num_cell_verts;  
int tot_num_cell_verts;  
int *cells  
double *sites;  
int num_complete_cells;  
int *complete_cells;  
double *areas;  
double *vols;  
int tot_num_cell_faces;  
int *num_cell_faces;  
int *num_face_verts;  
int tot_num_face_verts;  
int *face_verts;
```

# Compact DIY Datatypes

## C data structure

```
float mins[3];
float maxs[3];
double *verts;
double *sites;
int *complete_cells;
double *areas;
double *vols;
int *num_cell_faces;
int *num_face_verts;
int *face_verts;
```

DIY\_Datatype type;

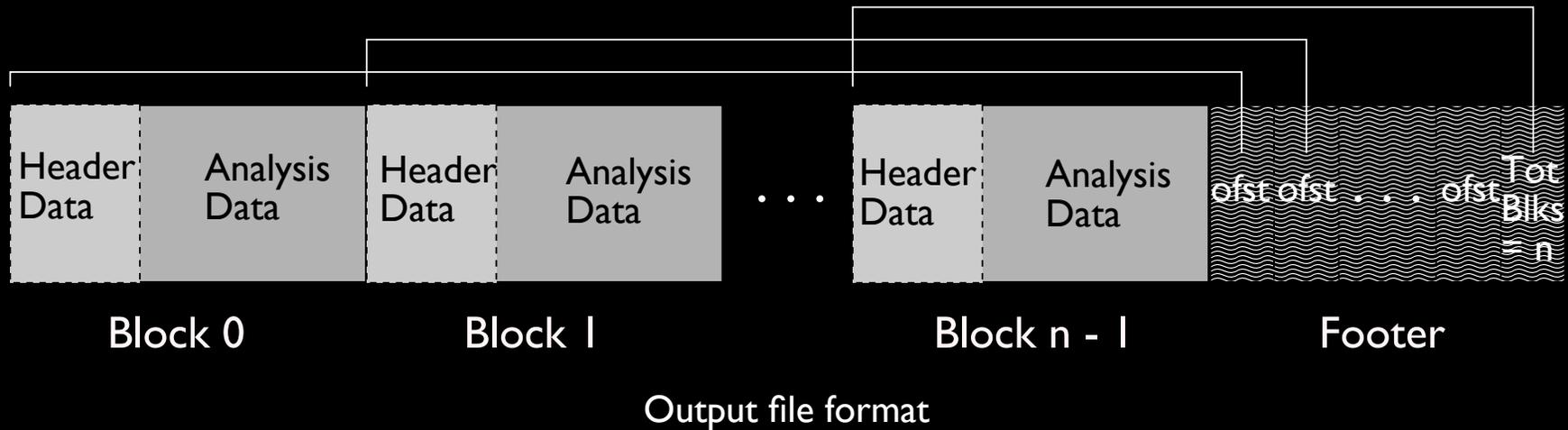
```
struct map_block_t map[] = {
    { DIY_FLOAT,  OFST, 3, offsetof(struct vblock_t, mins)           },
    { DIY_DOUBLE, ADDR, v->num_verts * 3, DIY_Addr(v->verts)       },
    { DIY_DOUBLE, ADDR, v->num_cells * 3, DIY_Addr(v->sites)        },
    { DIY_INT,     ADDR, v->num_complete_cells, DIY_Addr(v->complete_cells) },
    { DIY_DOUBLE, ADDR, v->num_complete_cells, DIY_Addr(v->areas)     },
    { DIY_DOUBLE, ADDR, v->num_complete_cells, DIY_Addr(v->vols)      },
    { DIY_INT,     ADDR, v->num_complete_cells, DIY_Addr(v->num_cell_faces) },
    { DIY_INT,     ADDR, v->tot_num_cell_faces, DIY_Addr(v->num_face_verts) },
    { DIY_INT,     ADDR, v->tot_num_face_verts, DIY_Addr(v->face_verts)   },
    { DIY_FLOAT,  OFST, 3, offsetof(struct vblock_t, maxs)           },
};
```

## DIY data type

```
DIY_Create_struct_datatype(DIY_Addr(vblock), 10, map, dtype);
```

- Any C/C++/Fortran data structure can be represented as an DIY (MPI) data type
- DIY uses data type to fetch data directly from memory or storage
- User does not pack / unpack (serialize / deserialize) data
- Zero copy at application level saves time and space
- DIY helps make data type creation easier

# #9 Output and Input Results



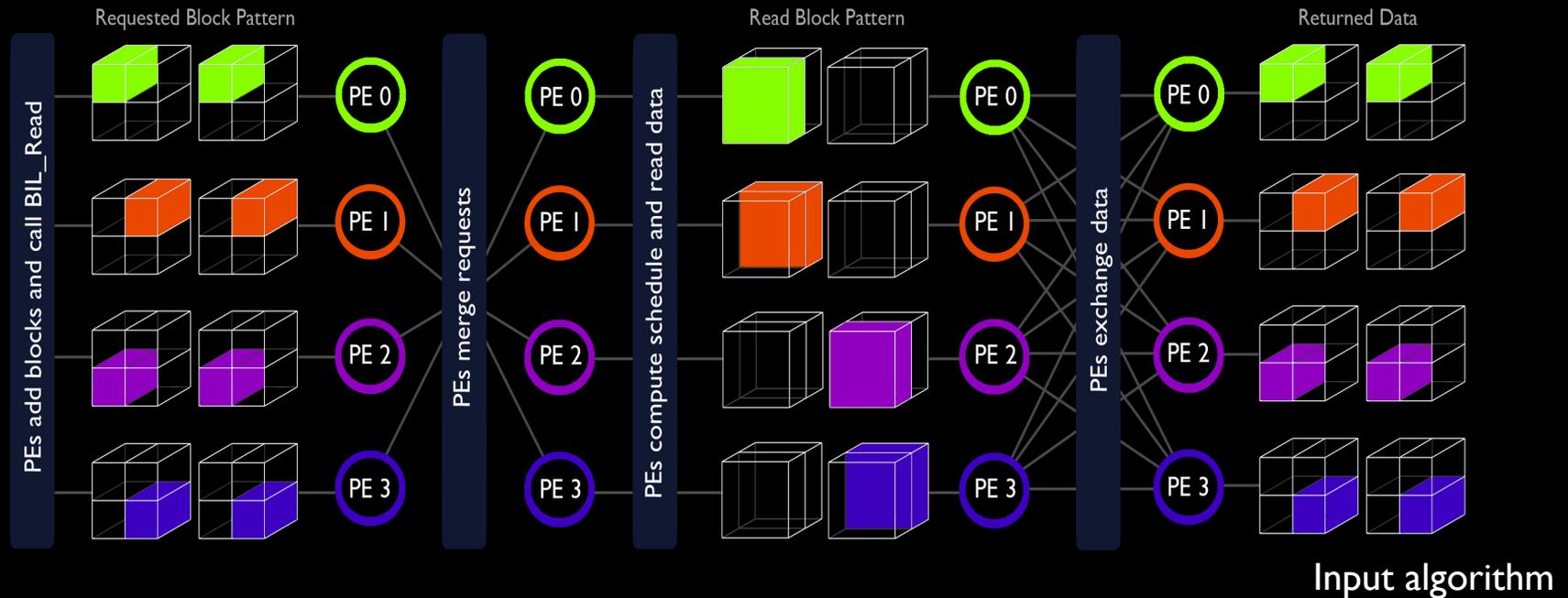
## Features

- Binary
- General header/data blocks
- Footer with indices
- Application assigns semantic value to DIY blocks
- Written efficiently in parallel
- Parallel block-wise compression

# Data Input

## Multiblock and Multifile I/O

- Application-level two-phase I/O
- Reads raw, netCDF, HDF5 (future)
- Read requests sorted and aggregated into large contiguous accesses
- Data redistributed to processes after reading
- Single and multi block/file domains
- 75% of IOR benchmark on actual scientific data

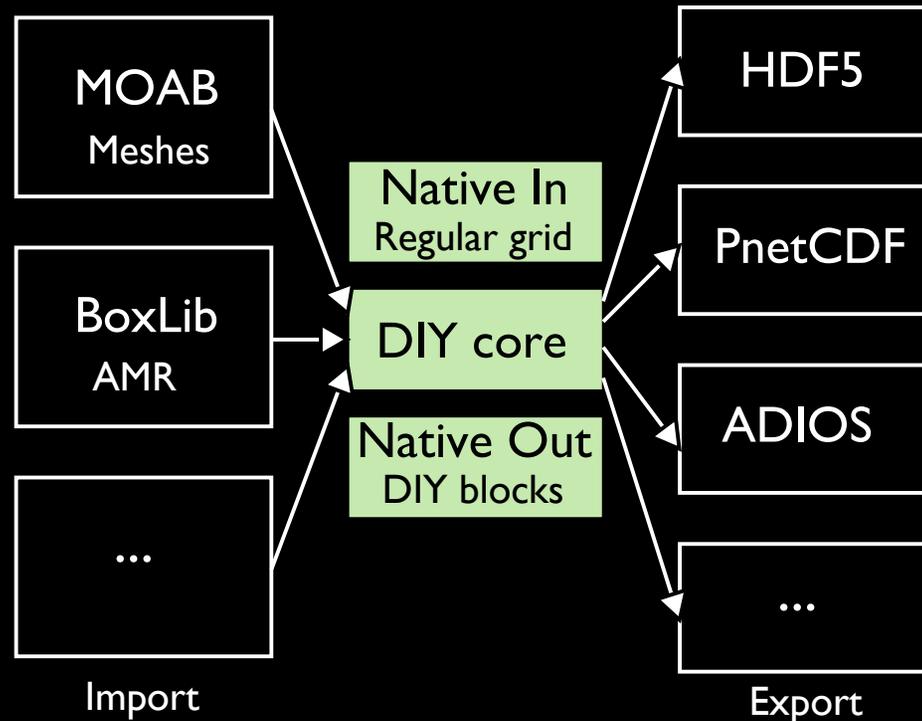


# #10: Play Nicely with Others

DIY by design doesn't include input or output data models. Rather than re-inventing them, it can import and export those models.

Import: Replicate model using `DIY_Decomposed()`, explicitly providing blocks and neighbors to DIY

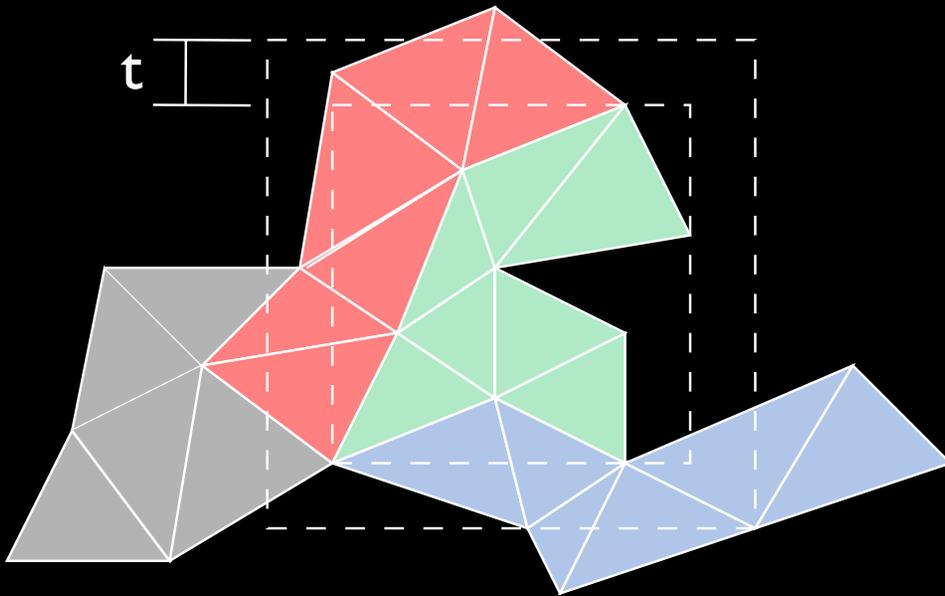
Export: Just use the other model API. DIY does not prevent you from making other library calls.



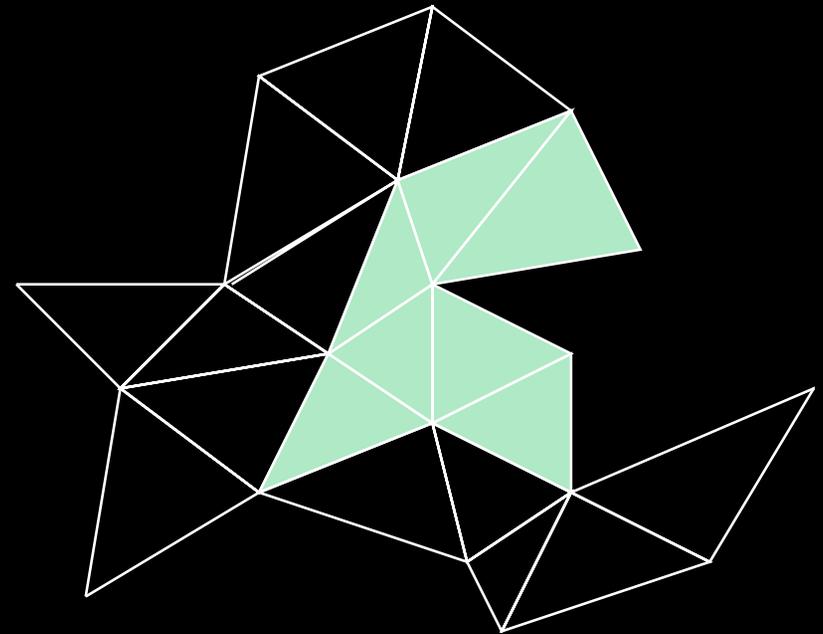
# Support Applications

## In Situ Unstructured Spectral Meshes With Help from MOAB

- Decomposition assigned by the application, not DIY
- DIY needs to get the decomposition from the app
- Call on MOAB for help with connectivity



Given the above mesh, assume the green block wants ghost cells in a given ghost radius of size  $t$ .

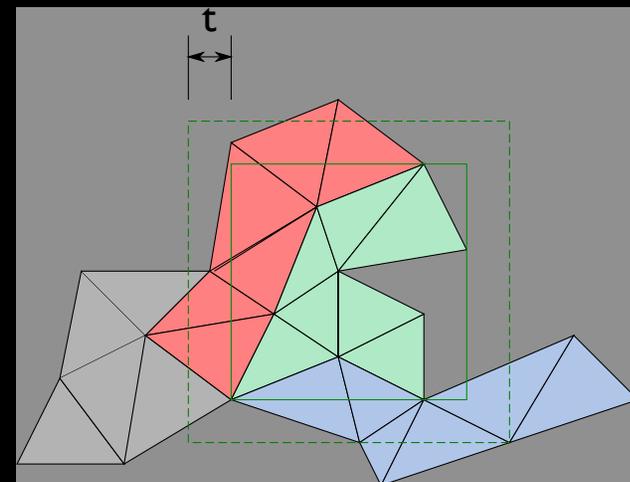


Result: the green block will have these cells (original green cells plus transparent cells)

# MOAB Example

```
void foo(imesh *mesh) { // MOAB mesh
  DIY_Init(num_blocks);
  for (num_blocks) {
    // query MOAB for verts in block
    get_adjacencies(hex, adj_verts);
    BlockBounds(bounds); // find min/max of verts
    // query MOAB for local neighbors of vertices
    get_adjacencies(adj_verts, adj_hexes);
    store adj_hexes in neighbors, num_neighbors
    // query MOAB for remote neighbors
    get_sharing_data(adj_verts, remote_handles,
      remote_procs);
    remote_data = remote_handles, remote_procs;
    // query MOAB for local vertex ids
    loc_vids[block] =
      id_from handle(shared_adj_verts);
  }
  DIY_Decomposed(blocks, bounds, remote_data,
    num_remote_data, loc_vids, neighbors,
    num_neighbors);
}
```

```
while (!done) {
  for (cells) {
    for (neighbors) {
      if (cell intersects neighbor extents + t &&
        cell was not sent already &&
        cell did not come from neighbor)
        post cell to neighbor;
    }
  }
  num_recvd = DIY_Exchange_neighbors();
  done = DIY_Check_done_all(!num_recvd);
}
```



# #11: Be Lightweight

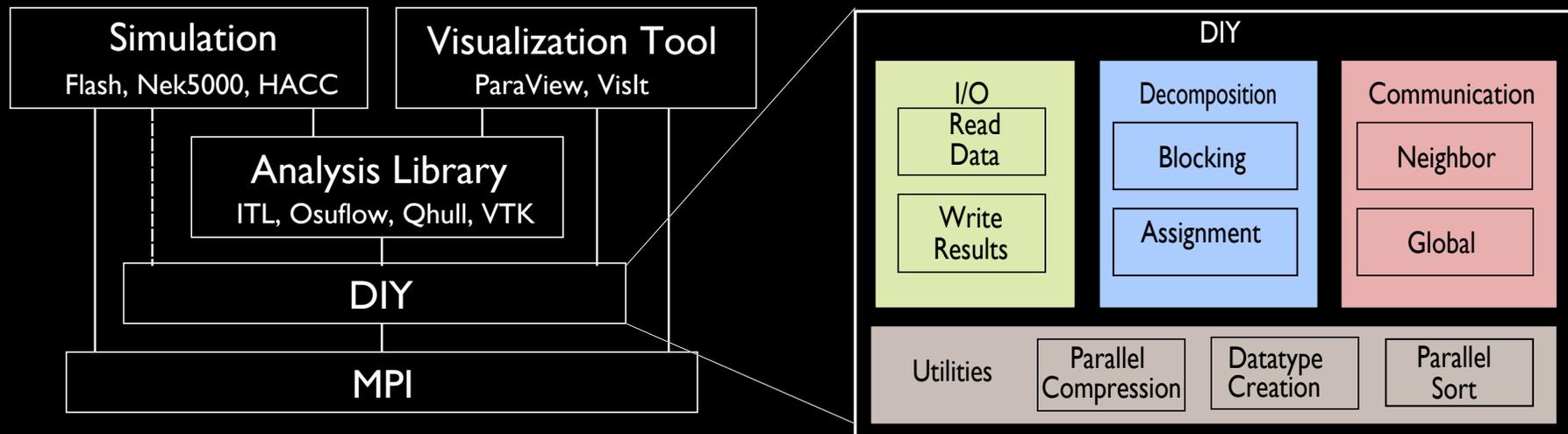
A library with a small  $\ell$

## Features

Parallel I/O to/from storage  
Domain decomposition  
Network communication  
Utilities

## Library

Written in C++ with C bindings  
Autoconf build system (configure, make, make install)  
Lightweight: libdiy.a 800KB  
Maintainable: ~15K lines of code, including examples



DIY usage and library organization



## Example Usage

### // initialize

```
int dim = 3; // number of dimensions in the problem
int tot_blocks = 8; // total number of blocks
int data_size[3] = {10, 10, 10}; // data size
MPI_Init(&argc, &argv); // init MPI before DIY
DIY_Init(dim, ROUND_ROBIN_ORDER, tot_blocks, &nblocks,
        data_size, MPI_COMM_WORLD);
```

### // decompose domain

```
int share_face = 0; // whether adjoining blocks share the same face
int ghost = 0; // additional layers of ghost cells
int ghost_dir = 0; // ghost cells apply to all or some sides of a block
int given[3] = {0, 0, 0}; // constraints on blocking (none)
DIY-Decompose(share_face, ghost, ghost_dir, given);
```

### // read data

```
for (int i = 0; i < nblocks; i++) {
    DIY_Block_starts_sizes(i, min, size);
    DIY_Read_add_block_raw(min, size, infile, MPI_INT, (void**)&(data[i]));
}
DIY_Read_blocks_all();
```

## Example API Continued

```
// your own local analysis
```

```
// merge results, in this example
```

```
// could be any combination / repetition of the three communication patterns
```

```
int rounds = 2; // two rounds of merging
```

```
int kvalues[2] = {4, 2}; // k-way merging, eg 4-way followed by 2-way merge
```

```
int nb_merged; // number of output merged blocks
```

```
DIY_Merge_blocks(in_blocks, hdrs, num_in_blocks, out_blocks, num_rounds, k_values,  
&MergeFunc, &CreateItemFunc, &DeleteItemFunc, &CreateTypeFunc, &num_out_blocks);
```

```
// write results
```

```
DIY_Write_open_all(outfile);
```

```
DIY_Write_blocks_all(out_blocks, num_out_blocks, datatype);
```

```
DIY_Write_close_all();
```

```
// terminate
```

```
DIY_Finalize(); // finalize DIY before MPI
```

```
MPI_Finalize();
```

# #13: Deliver Performance and Scalability

## DIY

- Peterka, T., Ross, R., Kendall, W., Gyulassy, A., Pascucci, V., Shen, H.-W., Lee, T.-Y., Chaudhuri, A.: Scalable Parallel Building Blocks for Custom Data Analysis. Proceedings of Large Data Analysis and Visualization Symposium (LDAV'11), IEEE Visualization Conference, Providence RI, 2011.
- Peterka, T., Ross, R.: Versatile Communication Algorithms for Data Analysis. 2012 EuroMPI Special Session on Improving MPI User and Developer Interaction IMUDI'12, Vienna, AT.

## DIY applications

- Peterka, T., Ross, R., Nouanesengsey, B., Lee, T.-Y., Shen, H.-W., Kendall, W., Huang, J.: A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. Proceedings IPDPS'11, Anchorage AK, May 2011.
- Gyulassy, A., Peterka, T., Pascucci, V., Ross, R.: The Parallel Computation of Morse-Smale Complexes. Proceedings of IPDPS'12, Shanghai, China, 2012.
- Nouanesengsy, B., Lee, T.-Y., Lu, K., Shen, H.-W., Peterka, T.: Parallel Particle Advection and FTLE Computation for Time-Varying Flow Fields. Proceedings of SCI2, Salt Lake, UT.
- Peterka, T., Kwan, J., Pope, A., Finkel, H., Heitmann, K., Habib, S., Wang, J., Zagaris, G.: Meshing the Universe: Integrating Analysis in Cosmological Simulations. Proceedings of the SCI2 Ultrascale Visualization Workshop, Salt Lake City, UT.
- Chaudhuri, A., Lee-T.-Y., Zhou, B., Wang, C., Xu, T., Shen, H.-W., Peterka, T., Chiang, Y.-J.: Scalable Computation of Distributions from Large Scale Data Sets. Proceedings of 2012 Symposium on Large Data Analysis and Visualization, LDAV'12, Seattle, WA.

# Parallel Time-Varying Flow Analysis

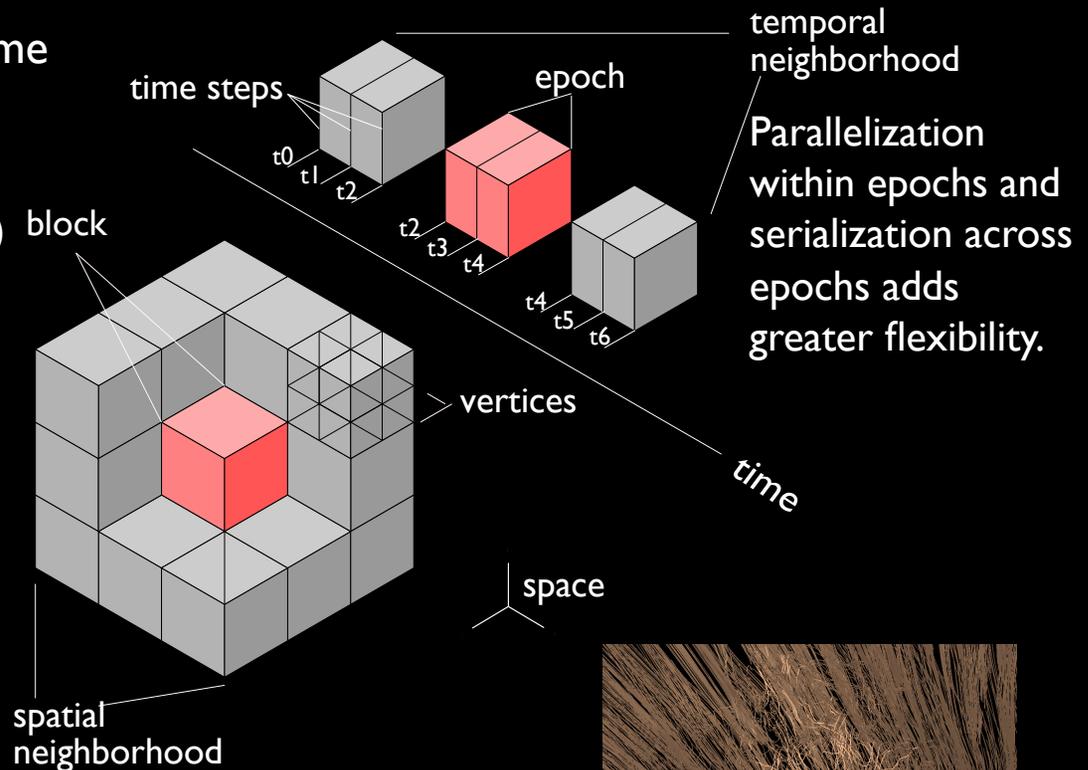
Collaboration with the Ohio State University and University of Tennessee Knoxville

## Approach

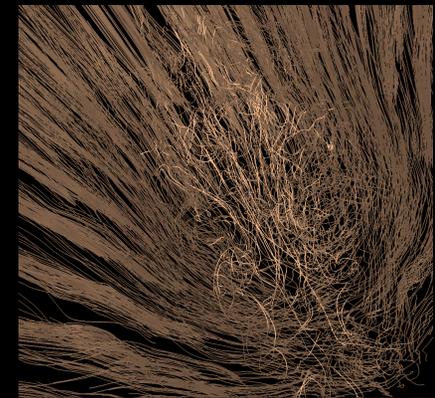
- In core / out of core processing of time steps
- Simple load balancing (multiblock assignment, early particle termination)
- Adjustable synchronization communication

## Algorithm

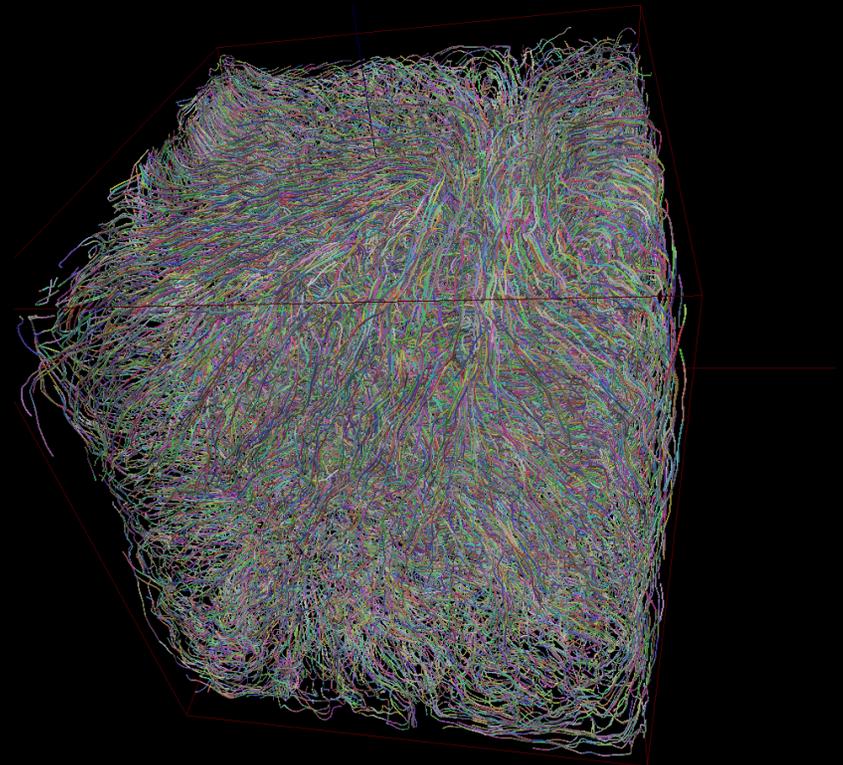
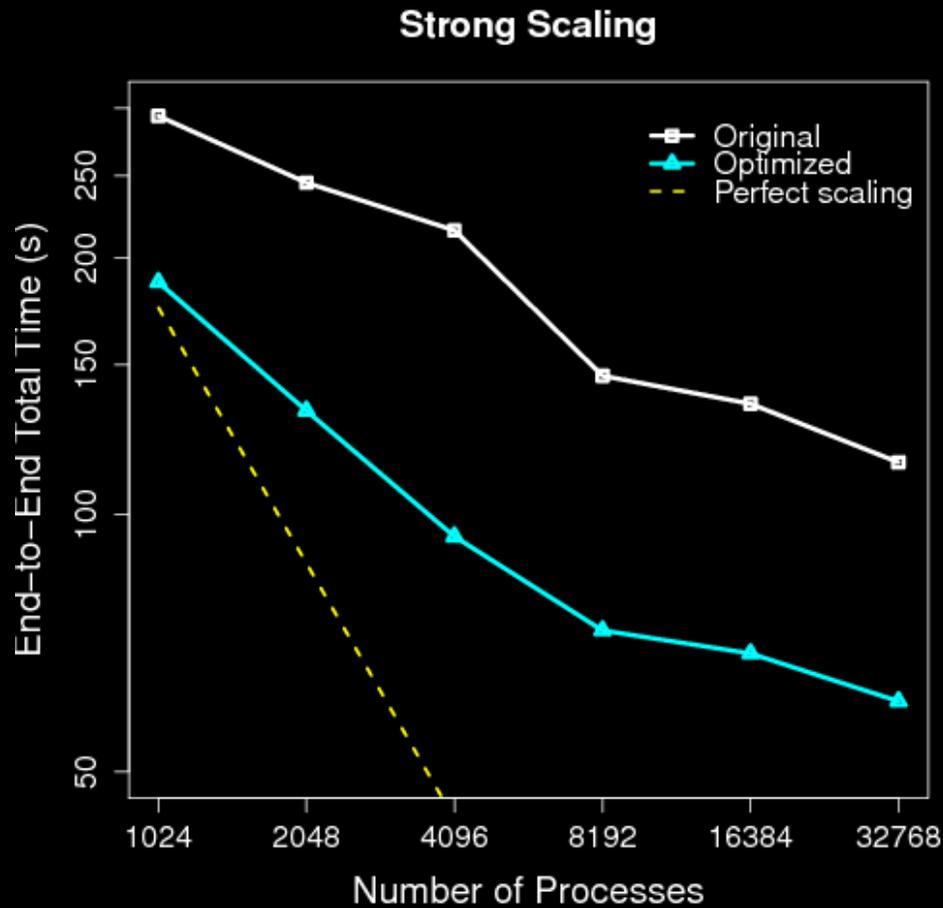
```
for (epochs) {  
  read my process' data blocks  
  for (rounds) {  
    for (my blocks) {  
      advect particles  
    }  
    exchange particles  
  }  
}
```



Pathline tracing of 32 time-steps of combustion in the presence of a cross-flow



# Particle Tracing



Particle tracing of  $\frac{1}{4}$  million particles in a  $2048^3$  thermal hydraulics dataset results in strong scaling to 32K processes and an overall improvement of 2X over earlier algorithms

# Parallel Information-Theoretic Analysis

Collaboration with the Ohio State University and New York University Polytechnic Institute

## Objective

- Decide what data are the most essential for analysis
- Minimize the information losses and maximize the quality of analysis
- Steer the analysis of data based on information saliency

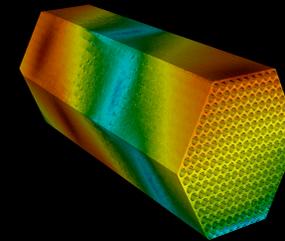
## Information-theoretic approach

- Quantify Information content based on Shannon's entropy
- Use this model to design new analysis data structures and algorithms

## Shannon's Entropy

The average amount of information expressed by the random variable is

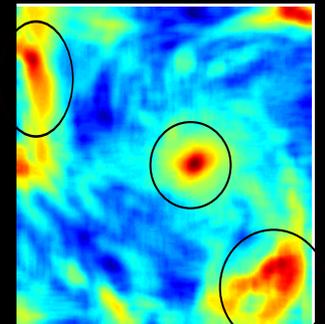
$$H(x) = - \sum_{i=1} p_i \log p_i$$



Nek5000  
CFD model

Information-  
theoretic  
algorithms

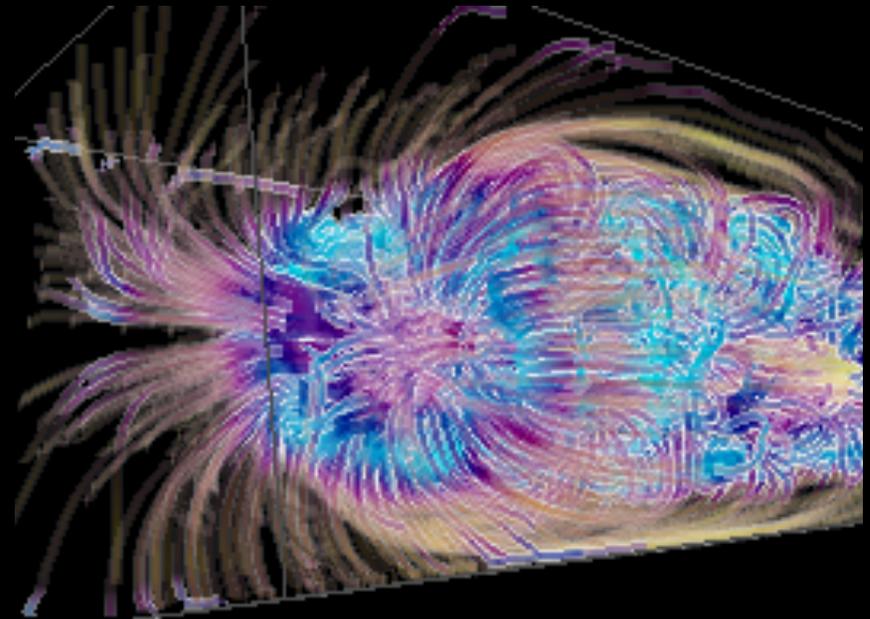
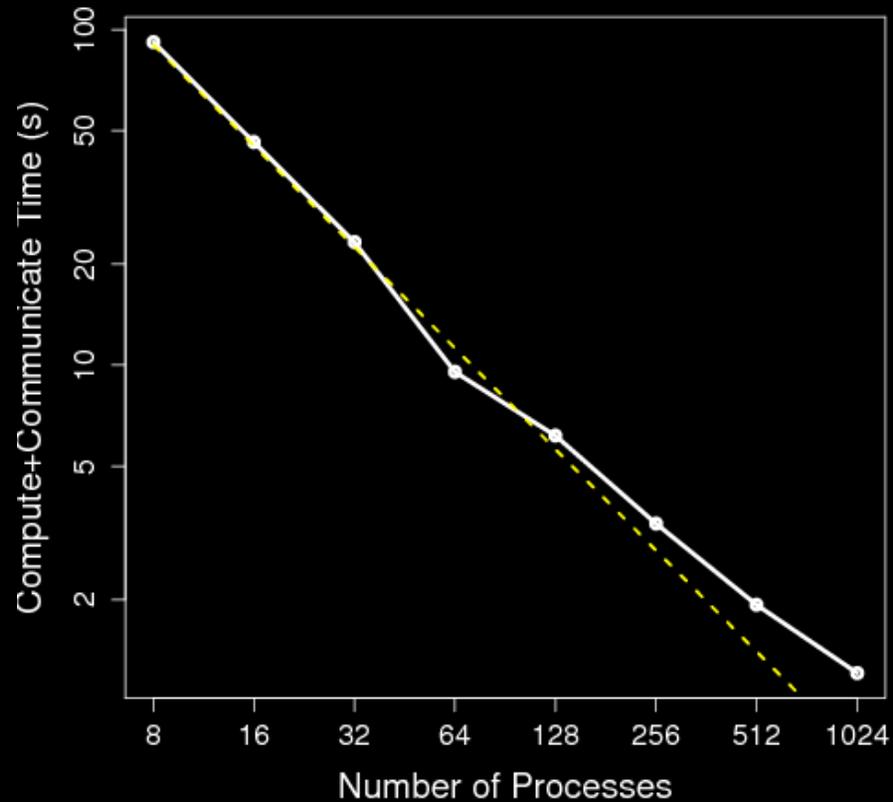
Areas of high information entropy--turbulent regions in original data--are the interesting regions in simulating coolant flow in a nuclear reactor.



Section of information  
entropy field

# Information Entropy

Strong Scaling



Computation of information entropy in  $126 \times 126 \times 512$  solar plume dataset shows 59% strong scaling efficiency.

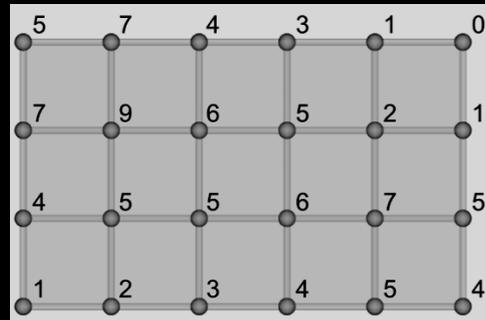
# Parallel Topological Analysis

Collaboration with SCI Institute, University of Utah

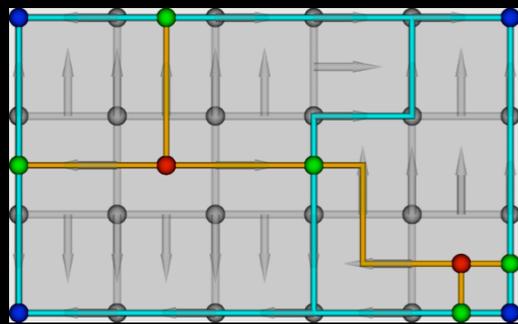
- Transform discrete scalar field into Morse-Smale complex
- Nodes are minima, maxima, saddle points of scalar values
- Arcs represent constant-sign gradient flow
- Used to quickly see topological structure



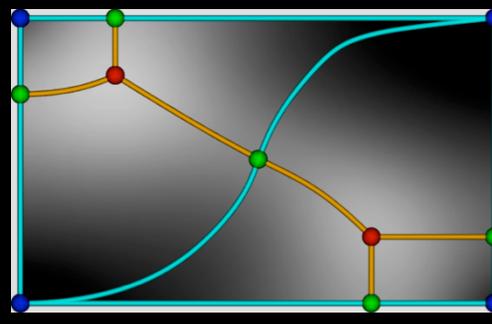
1



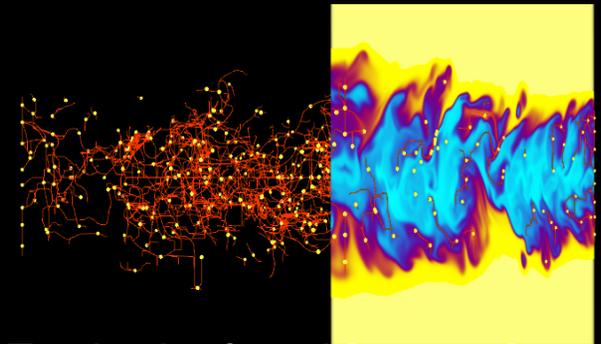
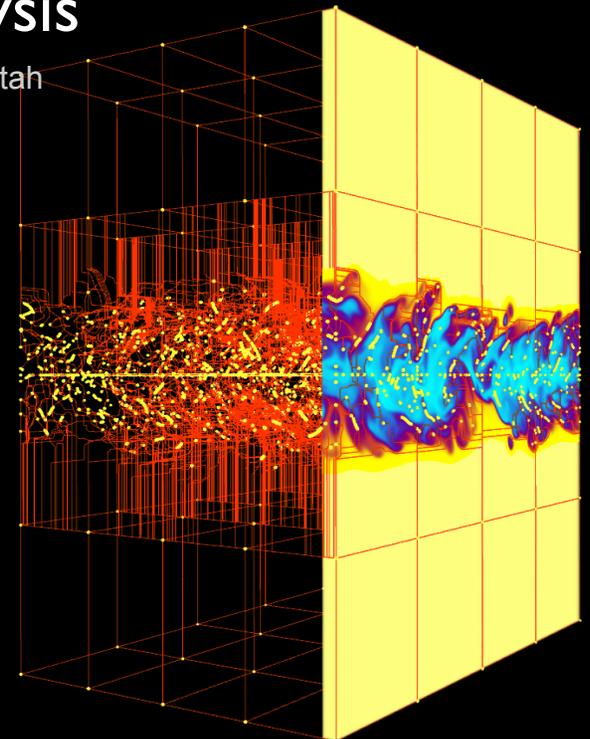
2



3



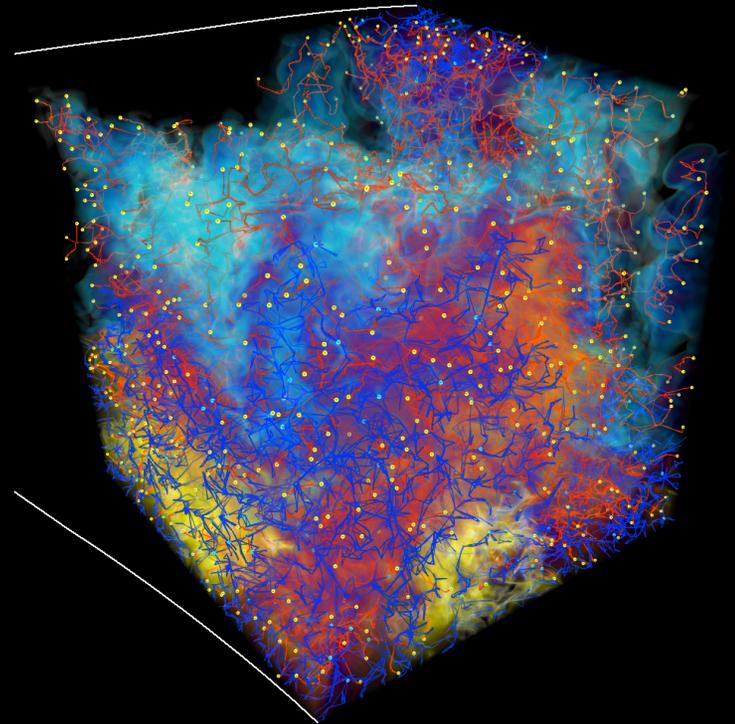
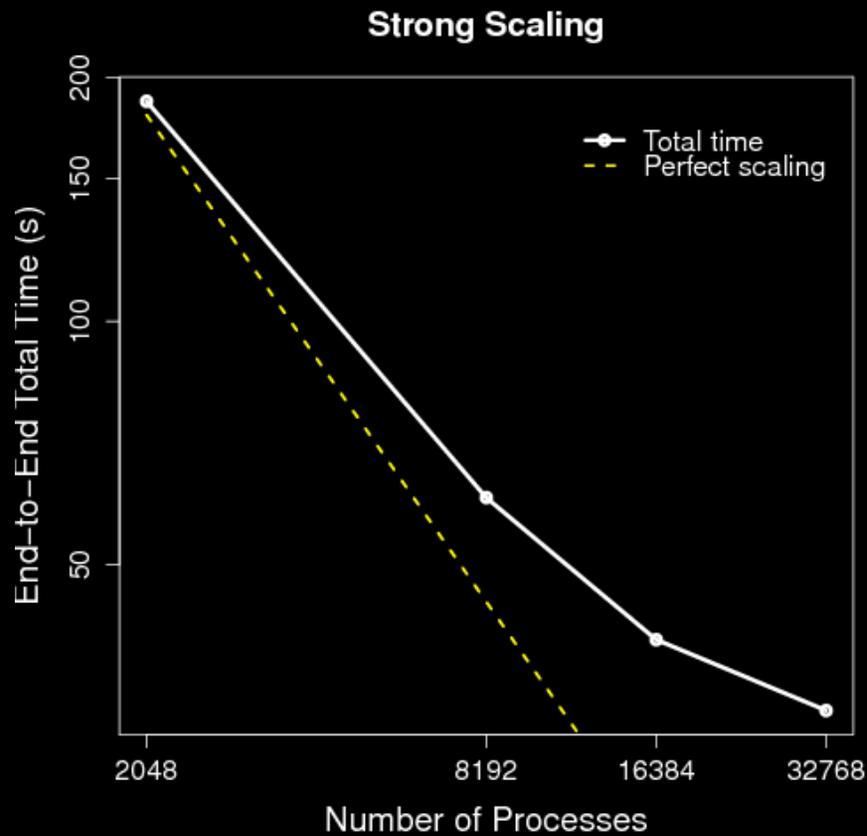
4



Two levels of simplification of the Morse-Smale complex for jet mixture fraction.

Example of computing discrete gradient and Morse-Smale Complex

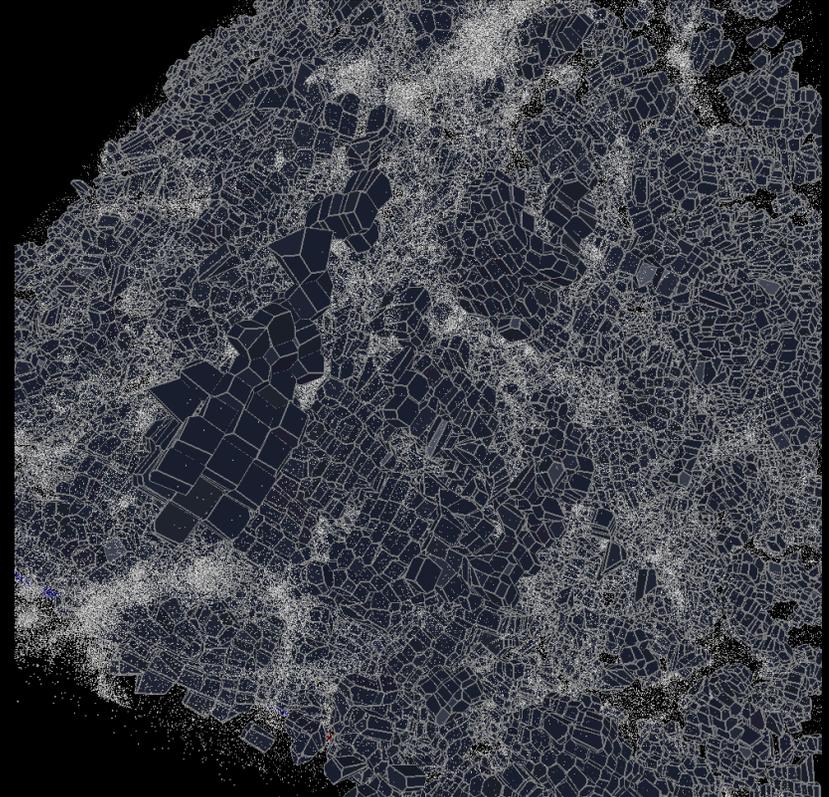
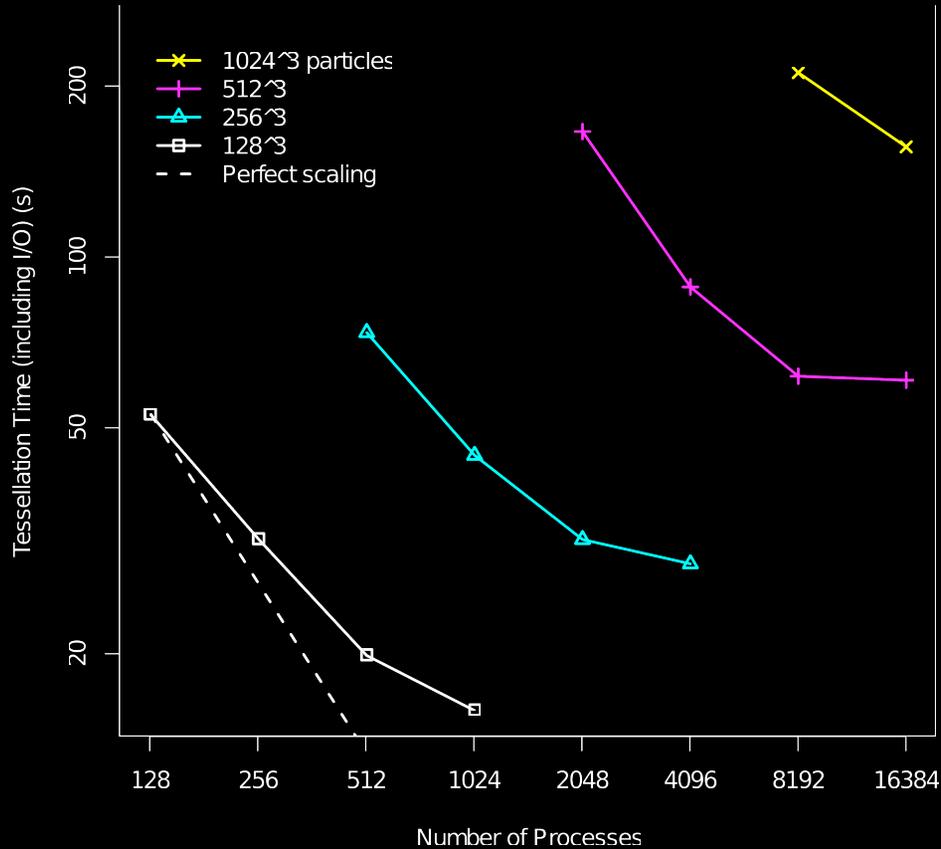
# Morse-Smale Complex



Computation of Morse-Smale complex in  $1152^3$  Rayleigh-Taylor instability data set results in 35% end-to-end strong scaling efficiency, including I/O.

# In Situ Voronoi Tessellation

Strong Scaling



For 128<sup>3</sup> particles, 41 % strong scaling for total tessellation time, including I/O; comparable to simulation strong scaling.

# Recap and Looking Ahead

## Done: Benefits

- Productivity
  - Express complex algorithms flexibly
    - Multiple blocks per process
    - Complete / partial reductions
    - Neighbor inclusion and communication
  - Simplify existing tasks
    - Custom data type creation
    - Compression
- Performance
  - Published scalability
  - Configurable algorithms

## To Do: Research Directions

- Advanced decomposition
  - Block groups
- Improved communication algorithms
  - Less synchronous, more overlap with computation
- High-level communication operations
  - Ghost cell exchange, kernel convolution (stencil)
- Load balancing
  - Block overloading, dynamic reassignment
- Programming models
  - MPI + X on Mira, Titan
- Usability
  - Improved API

## Acknowledgments:

### Facilities

Argonne Leadership Computing Facility (ALCF)  
Oak Ridge National Center for Computational Sciences (NCCS)

### Funding

DOE SDMAV Exascale Initiative  
DOE Exascale Codesign Center  
DOE SciDAC SDAV Institute

<https://svn.mcs.anl.gov/repos/diy/trunk>

Tom Peterka

[tpeterka@mcs.anl.gov](mailto:tpeterka@mcs.anl.gov)

Mathematics and Computer Science Division