DIY Parallel Data Analysis

I'm sure my wife will appreciate all the DIY I'm doing around the house for her!

Image courtesy pigtimes.com

Tom Peterka
tpeterka@mcs.anl.gov
Mathematics and Computer Science Division
Preliminaries
Moving from Postprocessing to Run-Time Scientific Data Analysis in HPC

Postprocessing particle tracing and visualization

Run-time particle tracing and postprocessing visualization
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.

Example of a data flow network
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

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Algorithm</td>
</tr>
<tr>
<td>Stochastic</td>
</tr>
</tbody>
</table>

| OS / Runtime |

or

With tools

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis Algorithm</td>
</tr>
<tr>
<td>Stochastic</td>
</tr>
</tbody>
</table>

| Data Movement |

| OS / Runtime |

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

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

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

You do this yourself
Can use serial libraries such as OSUFlow, Qhull, VTK (don’t have to start from scratch)

DIY handles this
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 (e.g., 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)
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

<table>
<thead>
<tr>
<th>4D</th>
<th>( \equiv )</th>
<th>3D</th>
<th>( \times )</th>
<th>1D</th>
</tr>
</thead>
<tbody>
<tr>
<td>((X_{\text{max}}, Y_{\text{max}}, Z_{\text{max}}, T_{\text{max}}))</td>
<td>((X_{\text{max}}, Y_{\text{max}}, Z_{\text{max}}))</td>
<td>((X_{\text{min}}, Y_{\text{min}}, Z_{\text{min}}))</td>
<td>((T_{\text{min}}))</td>
<td>((T_{\text{max}}))</td>
</tr>
</tbody>
</table>

| 4D Block | 3D Spatial Extent | 1D Temporal Extent |

| 4D Neighborhood (not drawn) | spatial block | vertices | 3D Spatial Neighborhood | 1D Temporal Neighborhood |

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)

Two examples of 3 out of a total of 25 neighborhoods
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
- homogeneity of data
3 Communication Patterns

Nearest neighbor

Round 0
- k = 4
Round 1
- k = 2
Results

Swap-based reduction

Round 0
- k = 4
Round 1
- k = 2
Results

Merge-based reduction

17
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 are 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
Adjustable Synchronization Communication Algorithm

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
}

**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).
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.

<table>
<thead>
<tr>
<th># Items</th>
<th>Bytes/Item</th>
<th>Total Bytes</th>
<th># Procs</th>
<th>Exchange Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>20</td>
<td>1 K</td>
<td>32</td>
<td>0.018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>128</td>
<td>0.028</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>512</td>
<td>0.028</td>
</tr>
<tr>
<td>256</td>
<td>20</td>
<td>5 K</td>
<td>32</td>
<td>0.064</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>128</td>
<td>0.097</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>512</td>
<td>0.098</td>
</tr>
<tr>
<td>1 K</td>
<td>20</td>
<td>20 K</td>
<td>32</td>
<td>0.235</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>128</td>
<td>0.354</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>512</td>
<td>0.357</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Items</th>
<th>Bytes/Item</th>
<th>Total Bytes</th>
<th># Procs</th>
<th>Exchange Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 K</td>
<td>20</td>
<td>80 K</td>
<td>512</td>
<td>1.358</td>
</tr>
<tr>
<td>16 K</td>
<td></td>
<td>320 K</td>
<td>512</td>
<td>5.507</td>
</tr>
<tr>
<td>64 K</td>
<td></td>
<td>1 M</td>
<td>512</td>
<td>22.083</td>
</tr>
<tr>
<td>256 K</td>
<td>20</td>
<td>5 M</td>
<td>512</td>
<td>90.238</td>
</tr>
<tr>
<td>1 M</td>
<td></td>
<td>20 M</td>
<td>512</td>
<td>351.068</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Items</th>
<th>Bytes/Item</th>
<th>Total Bytes</th>
<th># Procs</th>
<th>Exchange Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20 M</td>
<td>20 M</td>
<td>512</td>
<td>0.223</td>
</tr>
</tbody>
</table>

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**

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

**Tess (voronoi tessellation) Data Model**

```c
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;
tot_num_face_verts;
tot_num_face_verts;
int *face_verts;
```

Corollary: analysis X data model ≠ analysis Y data model
Compact DIY Datatypes

<table>
<thead>
<tr>
<th>C data structure</th>
<th>DIY Datatype type</th>
</tr>
</thead>
<tbody>
<tr>
<td>float mins[3];</td>
<td>DIY_Datatype</td>
</tr>
<tr>
<td>float maxs[3];</td>
<td></td>
</tr>
<tr>
<td>double *verts;</td>
<td>struct map_block_t = {</td>
</tr>
<tr>
<td>double *sites;</td>
<td>{ DIY_FLOAT, OFST,  3, offsetof(struct vblock_t, mins) },</td>
</tr>
<tr>
<td>int *complete_cells;</td>
<td>{ DIY_DOUBLE, ADDR, v-&gt;num_verts * 3, DIY_Addr(v-&gt;verts) },</td>
</tr>
<tr>
<td>double *areas;</td>
<td>{ DIY_DOUBLE, ADDR, v-&gt;num_cells * 3, DIY_Addr(v-&gt;sites) },</td>
</tr>
<tr>
<td>double *vols;</td>
<td>{ DIY_INT, ADDR, v-&gt;num_complete_cells, DIY_Addr(v-&gt;complete_cells) },</td>
</tr>
<tr>
<td>int *num_cell_faces;</td>
<td>{ DIY_DOUBLE, ADDR, v-&gt;num_complete_cells, DIY_Addr(v-&gt;areas) },</td>
</tr>
<tr>
<td>int *num_face_verts;</td>
<td>{ DIY_DOUBLE, ADDR, v-&gt;num_complete_cells, DIY_Addr(v-&gt;vols) },</td>
</tr>
<tr>
<td>int *face_verts;</td>
<td>{ DIY_INT, ADDR, v-&gt;num_complete_cells, DIY_Addr(v-&gt;num_cell_faces) },</td>
</tr>
<tr>
<td></td>
<td>{ DIY_INT, ADDR, v-&gt;tot_num_cell_faces, DIY_Addr(v-&gt;num_face_verts) },</td>
</tr>
<tr>
<td></td>
<td>{ DIY_INT, ADDR, v-&gt;tot_num_face_verts, DIY_Addr(v-&gt;face_verts) },</td>
</tr>
<tr>
<td></td>
<td>{ DIY_FLOAT, OFST,  3, offsetof(struct vblock_t, maxs) },</td>
</tr>
<tr>
<td></td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>DIY_Create_struct_datatype(DIY_Addr(vblock), 10, map, dtype);</td>
</tr>
</tbody>
</table>

- 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
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

Kendall et al., Towards a General I/O Layer for Parallel Visualization Applications, CG&A ‘11
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).
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 ℓ

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
#12: Come with Instructions

**Documentation**
- README for installation
- User’s manual with description, examples of custom datatypes, complete API reference

**Tutorial Examples**
- Block I/O: Reading data, writing analysis results
- Static: Merge-based, Swap-based reduction, Neighborhood exchange
- Time-varying: Neighborhood exchange
- Spare thread: Simulation and analysis overlap
- MOAB: Unstructured mesh data model
- VTK: Integrating DIY communication with VTK filters
- R: Integrating DIY communication with R stats algorithms
- Multimodel: multiple domains and communicating between them
// 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();
// 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.: Versatile Communication Algorithms for Data Analysis. 2012 EuroMPI Special Session on Improving MPI User and Developer Interaction IMUDI'12, Vienna, AT.

**DIY applications**

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

Peterka et al., A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields, IPDPS '11
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

\[ H(x) = - \sum_{i=1}^{n} p_i \log p_i \]
Computation of information entropy in 126x126x512 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

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

Gyulassy et al., The Parallel Computation of Morse-Smale Complexes, Submitted to IPDPS '12
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.
For $128^3$ 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
Mathematics and Computer Science Division