The Zoltan Toolkit – Partitioning, Ordering, and Coloring

Erik Boman, Cedric Chevalier, Karen Devine
Sandia National Laboratories, NM

Ümit Çatalyürek
Ohio State University

Dagstuhl Seminar, Feb 2009
Outline

• High-level view of Zoltan
• Requirements, data models, and interface
• Partitioning and Dynamic Load Balancing
• Graph Coloring
• Matrix Ordering
• Alternate Interfaces
• Future Directions
• Demo
• Hands-On Examples
The Zoltan Toolkit

- Library of data management services for unstructured, dynamic and/or adaptive computations.

Dynamic Load Balancing

Data Migration

Graph Coloring

Matrix Ordering

Unstructured Communication

Distributed Data Directories
Zoltan System Assumptions

• Assume distributed memory model.
• Data decomposition + “Owner computes”:
  – The data is distributed among the processors.
  – The owner performs all computation on its data.
  – Data distribution defines work assignment.
  – Data dependencies among data items owned by different processors incur communication.

• Requirements:
  – C compiler (C++ optional)
  – GNU Make (gmake)
  – MPI required for parallel execution
Zoltan Supports Many Applications

- Different applications, requirements, data structures.

- **Multiphysics simulations**
- **Parallel electronics networks**
- **Crash simulations**
- **Particle methods**
- **Adaptive mesh refinement**

\[
A \cdot x = b
\]
Zoltan Interface Design

- Common interface to each class of tools.
- Tool/method specified with user parameters.

- **Data-structure neutral design.**
  - Supports wide range of applications and data structures.
  - Imposes no restrictions on application’s data structures.
  - Application does not have to build Zoltan’s data structures.
Zoltan Interface

• Fairly simple, easy-to-use interface.
  – Small number of callable Zoltan functions.
  – Callable from C, C++, Fortran.

• Requirement: Unique global IDs for objects to be partitioned/ordered/colored. For example:
  – Global element number.
  – Global matrix row number.
  – (Processor number, local element number)
  – (Processor number, local particle number)
Zoltan Application Interface

- Application interface:
  - Zoltan queries the application for needed info.
    - IDs of objects, coordinates, relationships to other objects.
  - Application provides simple functions to answer queries.
  - Small extra costs in memory and function-call overhead.
- Query mechanism supports…
  - Geometric algorithms
    - Queries for dimensions, coordinates, etc.
  - Hypergraph- and graph-based algorithms
    - Queries for edge lists, edge weights, etc.
  - Tree-based algorithms
    - Queries for parent/child relationships, etc.
- Once query functions are implemented, application can access all Zoltan functionality.
  - Can switch between algorithms by setting parameters.
Zoltan Application Interface

APPLICATION

Initialize Zoltan (Zoltan_Initialize, Zoltan_Create)

Select Method and Parameters (Zoltan_Set_Params)

Register query functions (Zoltan_Set_Fn)

(Re)partition (Zoltan_LB_Partition)

Move data (Zoltan_Migrate)

COMPUTE

Clean up (Zoltan_Destroy)

ZOLTAN

Zoltan_LB_Partition:
- Call query functions.
- Build data structures.
- Compute new decomposition.
- Return import/export lists.

Zoltan_Migrate:
- Call packing query functions for exports.
- Send exports.
- Receive imports.
- Call unpacking query functions for imports.
# Zoltan Query Functions

## General Query Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_OBJ_FN</td>
<td>Number of items on processor</td>
</tr>
<tr>
<td>ZOLTAN_OBJ_LIST_FN</td>
<td>List of item IDs and weights.</td>
</tr>
</tbody>
</table>

## Geometric Query Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_GEOM_FN</td>
<td>Dimensionality of domain.</td>
</tr>
<tr>
<td>ZOLTAN_GEOM_FN</td>
<td>Coordinates of items.</td>
</tr>
</tbody>
</table>

## Hypergraph Query Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_HG_SIZE_CS_FN</td>
<td>Number of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_CS_FN</td>
<td>List of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_SIZE_EDGE_WTS_FN</td>
<td>Number of hyperedge weights.</td>
</tr>
<tr>
<td>ZOLTAN_HG_EDGE_WTS_FN</td>
<td>List of hyperedge weights.</td>
</tr>
</tbody>
</table>

## Graph Query Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_EDGE_FN</td>
<td>Number of graph edges.</td>
</tr>
<tr>
<td>ZOLTAN_EDGE_LIST_FN</td>
<td>List of graph edges and weights.</td>
</tr>
</tbody>
</table>
Using Zoltan in Your Application

1. Download Zoltan.
2. Build Zoltan library.
3. Decide what your objects are.
   - Elements? Grid points? Matrix rows? Particles?
4. Decide which tools (partitioning/ordering/coloring/utilities) and class of method (geometric/graph/hypergraph) to use.
5. Include “zoltan.h” in files calling Zoltan.
6. Write required query functions for your application.
   - Required functions are listed with each method in Zoltan User’s Guide.
7. Call Zoltan from your application.
8. Compile application; link with libzoltan.a.
   - `mpicc application.c -lzoltan`
Partitioning and Load Balancing

- Assignment of application data to processors for parallel computation.
- Applied to grid points, elements, matrix rows, particles, ....
Zoltan computes the **difference** ($\Delta$) from current distribution

Choose between:

a) Import lists (data to import **from** other procs)

b) Export lists (data to export **to** other procs)

c) Both (the default)

Note that parts may differ from processors.

```c
err = Zoltan_LB_Partition(zz,
    &changes, /* Flag indicating whether partition changed */
    &numGidEntries, &numLidEntries,
    &numImport, /* objects to be imported to new part */
    &importGlobalGids, &importLocalGids, &importProcs, &importToPart,
    &numExport, /* objects to be exported from old part */
    &exportGlobalGids, &exportLocalGids, &exportProcs, &exportToPart);
```
Static Partitioning

- Static partitioning in an application:
  - Data partition is computed.
  - Data are distributed according to partition map.
  - Application computes.

- Ideal partition:
  - Largest processor time is minimized.
  - Inter-processor communication costs are kept low.

- Zoltan_Set_Param(zz, "LB APPROACH", "PARTITION");
Dynamic Repartitioning (a.k.a. Dynamic Load Balancing)

- Dynamic repartitioning (load balancing) in an application:
  - Data partition is computed.
  - Data are distributed according to partition map.
  - Application computes and, perhaps, adapts.
  - Process repeats until the application is done.

- Ideal partition:
  - Largest processor time is minimized.
  - Inter-processor communication costs are kept low.
  - Cost to redistribute data is also kept low.

- Zoltan_Set_Param(zz, “LB_APPROACH”, “REPARTITION”);
Zoltan Toolkit: Suite of Partitioners

- No single partitioner works best for all applications.
  - Trade-offs:
    - Quality vs. speed.
    - Geometric locality vs. data dependencies.
    - High-data movement costs vs. tolerance for remapping.
- Application developers may not know which partitioner is best for application.

- Zoltan contains suite of partitioning methods.
  - Application changes only one parameter to switch methods.
    - Zoltan_Set_Param(zz, “LB_METHOD”, “new_method_name”);
  - Allows experimentation/comparisons to find most effective partitioner for application.
Partitioning Algorithms in the Zoltan Toolkit

**Geometric (coordinate-based) methods**
- Recursive Coordinate Bisection
- Recursive Inertial Bisection
- Space Filling Curve Partitioning
- Refinement-tree Partitioning

**Combinatorial (topology-based) methods**
- Hypergraph Partitioning
- Hypergraph Repartitioning
- PaToH (Catalyurek & Aykanat)
- Graph Partitioning
  - ParMETIS (Karypis et al.)
  - PT-Scotch (Pellegrini et al.)
Geometric Partitioning

• Partition based on geometric locality of objects.
  – Assign physically close objects to the same processor.
• Communication costs are controlled only implicitly.
  – Assumption: objects that depend on each other are physically near each other.
  – Reasonable assumption for particle simulations, contact detection and some meshes.

Recursive Coordinate Bisection (RCB)
Berger & Bokhari, 1987

Recursive Inertial Bisection (RIB)
Simon, 1991; Taylor & Nour Omid, 1994

Space Filling Curve Partitioning (HSFC)
Warren & Salmon, 1993; Pilkington & Baden, 1994; Patra & Oden, 1995
Recursive Coordinate Bisection

• Zoltan_Set_Param(zz, “LB_METHOD”, “RCB”);
• Idea:
  – Divide work into two equal parts using a cutting plane orthogonal to a coordinate axis.
  – Recursively cut the resulting subdomains.

1st cut

2nd

3rd

2nd

3rd

3rd

3rd
Geometric Repartitioning

- Implicitly achieves low data redistribution costs.
- For small changes in data, cuts move only slightly, resulting in little data redistribution.
Applications of Geometric Methods

- Adaptive Mesh Refinement
- Parallel Volume Rendering
- Particle Simulations
- Crash Simulations and Contact Detection
- Crash Simulations
Geometric Methods: Advantages and Disadvantages

• Advantages:
  – Conceptually simple; fast and inexpensive.
  – All processors can inexpensively know entire partition (e.g., for global search in contact detection).
  – No connectivity info needed (e.g., particle methods).
  – Good on specialized geometries.

• Disadvantages:
  – No explicit control of communication costs.
  – Mediocre partition quality.
  – Can generate disconnected subdomains for complex geometries.
  – Need coordinate information.
Graph Partitioning

• Represent problem as a weighted graph.
  – Vertices = objects to be partitioned.
  – Edges = dependencies between two objects.
  – Weights = work load or amount of dependency.

• Partition graph so that …
  – Parts have equal vertex weight.
  – Weight of edges cut by part boundaries is small.

• Zoltan_Set_Param(zz, “LB_METHOD”, “GRAPH”);
• Zoltan_Set_Param(zz, “GRAPH_PACKAGE”, “ZOLTAN”); or
  Zoltan_Set_Param(zz, “GRAPH_PACKAGE”, “PARMETIS”);

• Kernighan, Lin, Simon, Hendrickson, Leland, Kumar, Karypis, et al.
Graph Repartitioning

• Diffusive strategies (Cybenko, Hu, Blake, Walshaw, Schloegel, et al.)
  – Shift work from highly loaded processors to less loaded neighbors.
  – Local communication keeps data redistribution costs low.

• Multilevel partitioners that account for data redistribution costs in refining partitions (Schloegel, Karypis)
  – Parameter weights application communication vs. redistribution communication.
Applications using Graph Partitioning

Finite Element Analysis

Multiphysics and multiphase simulations

Linear solvers & preconditioners
(square, structurally symmetric systems)
Graph Partitioning: Advantages and Disadvantages

• Advantages:
  – Highly successful model for mesh-based PDE problems.
  – Explicit control of communication volume gives higher partition quality than geometric methods.
  – Excellent software available.
    • Serial: Chaco (SNL)
      Jostle (U. Greenwich)
      METIS (U. Minn.)
      Scotch (U. Bordeaux)
    • Parallel: Zoltan (SNL)
      ParMETIS (U. Minn.)
      PJostle (U. Greenwich)
      PT-Scotch (LaBRI/INRIA)

• Disadvantages:
  – More expensive than geometric methods.
  – Edge-cut model only approximates communication volume.
Hypergraph Partitioning

- Zoltan_Set_Param(zz, “LB_METHOD”, “HYPERGRAPH”);
- Zoltan_Set_Param(zz, “HYPERGRAPH_PACKAGE”, “ZOLTAN”); or Zoltan_Set_Param(zz, “HYPERGRAPH_PACKAGE”, “PATOH”);


- Hypergraph model:
  - Vertices = objects to be partitioned.
  - Hyperedges = dependencies between two or more objects.

- Partitioning goal: Assign equal vertex weight while minimizing hyperedge cut weight.

Graph Partitioning Model

Hypergraph Partitioning Model
Hypergraph Repartitioning

- Augment hypergraph with data redistribution costs.
  - Account for data’s current processor assignments.
  - Weight dependencies by their size and frequency of use.
- Partitioning then tries to minimize total communication volume:
  - Data redistribution volume
  - + Application communication volume
  - Total communication volume
- Data redistribution volume: callback returns data sizes.
  - `Zoltan_Set_Fn(zz, ZOLTAN_OBJ_SIZE_MULTI_FN_TYPE, myObjSizeFn, 0);`
- Application communication volume = Hyperedge cuts * Number of times the communication is done between repartitionings.
  - `Zoltan_Set_Param(zz, “PHG_REPART_MULTIPLIER”, “100”);`
Hypergraph Applications

Finite Element Analysis

Circuit Simulations

Linear programming for sensor placement

Multiphysics and multiphase simulations

Linear solvers & preconditioners (no restrictions on matrix structure)

Data Mining
Hypergraph Partitioning: Advantages and Disadvantages

• Advantages:
  – Communication volume reduced 30-38% on average over graph partitioning (Catalyurek & Aykanat).
    • 5-15% reduction for mesh-based applications.
  – More accurate communication model than graph partitioning.
    • Better representation of highly connected and/or non-homogeneous systems.
  – Greater applicability than graph model.
    • Can represent rectangular systems and non-symmetric dependencies.

• Disadvantages:
  – Usually more expensive than graph partitioning.
Multi-criteria Load-balancing

- **Multiple constraints or objectives**
  - Compute a single partition that is good with respect to multiple factors.
    - Balance both computation and memory.
    - Balance meshes in loosely coupled physics.
    - Balance multi-phase simulations.
  - Extend algorithms to multiple weights
    - Difficult. No guarantee good solution exists.
- **Zoltan_Set_Param(zz, “OBJ_WEIGHT_DIM”, “2”);**
  - Available in RCB, RIB and ParMETIS graph partitioning.
  - In progress in Hypergraph partitioning.
Heterogeneous Architectures

• Clusters may have different types of processors.
• Assign “capacity” weights to processors.
  – E.g., Compute power (speed).
  – Zoltan_LB_Set_Part_Sizes(...);
    • Note: Can use this function to specify part sizes for any purpose.
• Balance with respect to processor capacity.

• Hierarchical partitioning: Allows different partitioners at different architecture levels.
  – Zoltan_Set_Param(zz, “LB_METHOD”, “HIER”);
  – Requires three additional callbacks to describe architecture hierarchy.
    • ZOLTAN_HIER_NUM_LEVELS_FN
    • ZOLTAN_HIER_PARTITION_FN
    • ZOLTAN_HIER_METHOD_FN
Graph Coloring

• **Problem:** Color the vertices of a graph with as few colors as possible such that no two adjacent vertices have the same color.
  – **Distance-2:** No vertices connected by a length-2 path have the same color

• **Applications**
  – Iterative sparse solvers
  – Preconditioners
  – Automatic differentiation
  – Sparse tiling
Zoltan Graph Coloring

- Parallel distance-1 and distance-2 graph coloring.
- Graph built using same application interface and code as graph partitioners.
- Generic coloring interface; easy to add new coloring algorithms.
- Algorithms
  - Distance-1 coloring: Bozdag, Gebremedhin, Manne, Boman, Catalyurek, EuroPar’05, JPDC’08.
  - Distance-2 coloring: Bozdag, Catalyurek, Gebremedhin, Manne, Boman, Ozguner, HPCC’05, SISC’09 (in submission).
Both distance-1 and distance-2 coloring routines are invoked by the `Zoltan_Color` function.

Graph query functions required.

The colors assigned to the objects are returned in an array of integers.
A Parallel Coloring Framework

• Color vertices iteratively in rounds using a first fit strategy.
• Each round is broken into supersteps:
  – Color a certain number of vertices.
  – Exchange recent color information.
• Detect conflicts at the end of each round.
• Repeat until all vertices receive consistent colors.
Experimental Results
Sparse Matrix Ordering Problem

• Work and fill in sparse direct solvers (Cholesky, LU) depend on the matrix ordering.
  – Optimal ordering is NP-hard.
  – Many heuristics: Nested dissection, minimum degree, etc.
  – Nested dissection is preferred for parallel processing.
Matrix ordering within Zoltan

• Computed by third party libraries:
  – ParMETIS
  – Scotch (actually PT-Scotch, the parallel part)
  – Easy to add another one.

• The calls to the external ordering library are transparent for the user. Thus Zoltan’s API can be a standard way to compute ordering.

• Native ordering in Zoltan planned.
Ordering interface in Zoltan

- Compute ordering with one function: `Zoltan_Order`
- Output provided:
  - New order of the unknowns (direct permutation), available in two forms:
    - one is the new number in the interval [0,N-1];
    - the other is the new order of Global IDs.
  - Access to elimination tree, “block” view of the ordering.
## Comparison PT-Scotch vs ParMetis

<table>
<thead>
<tr>
<th>Test case</th>
<th>Number of processes</th>
<th>audikw1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>OPPTS</td>
<td>5.73E+12</td>
<td>5.65E+12</td>
</tr>
<tr>
<td>OPPM</td>
<td>5.82E+12</td>
<td>6.37E+12</td>
</tr>
<tr>
<td>tPTS</td>
<td>73.11</td>
<td>53.19</td>
</tr>
<tr>
<td>tPM</td>
<td>32.69</td>
<td>23.09</td>
</tr>
</tbody>
</table>

### Bar Charts

- Left: NNZ(L)/NNZ(A) vs Number of Processors
- Right: OPC base 1.0 for sequential Scotch vs Number of Processors
Summary of Matrix Ordering

• Zoltan provides access to efficient parallel ordering for sparse matrices.
  – PT-Scotch gives best quality (but longer time).
• Zoltan provides a standard way to call parallel ordering.
• Zoltan will provide also its own ordering tool in the future, for non-symmetric problems.
  – HUND algorithm (talk by S. Donfack)
Other Zoltan Functionality

• Tools needed when doing dynamic load balancing:
  – Data Migration
  – Unstructured Communication Primitives
  – Distributed Data Directories

• All functionality described in Zoltan User’s Guide.
Alternate Interfaces to Zoltan

• C++ and F90 interfaces in Zoltan.

• Isorropia package in Trilinos solver toolkit.
  – Epetra Matrix interface to Zoltan partitioning.
    • \[B = \text{Isorropia}::\text{Epetra}::\text{create\_balanced\_copy}(A, \text{params});\]
  – Trilinos v9 includes ordering and coloring interfaces in Isorropia (in addition to partitioning).

• ITAPS iMesh interface to Zoltan.
  – New iMeshP parallel mesh interface in progress.
Zoltan for CSC Developers

• Zoltan is an open-source project.
  – We welcome contributions from the CSC community!
  – Data-neutral interface makes it easy to integrate new packages as third-party libraries.

• Requirements for 3rd party software:
  – Open source
  – Written in C or C++
  – Library interface

• Talk to us if you may be interested!
Current Work

- Two-dimensional matrix partitioning
  - Fine-grain hypergraph method
    - Catalyurek & Aykanat (2000)
  - Nested dissection matrix partitioning
    - Boman & Wolf (2008)
  - Will require Isorropia (Trilinos).

- Multi-criteria hypergraph partitioning
  - May be used for “checkerboard” matrix partitioning.

- Non-symmetric matrix ordering (HUND).
  - For sparse LU factorization.
Future Zoltan extensions

• May add support for:
  – Matching
    • MatchBox (Dobrian)
    • MatchBoxP (Halappanavar)
  – More coloring
    • ColPack (Gebremedhin)
DEMO and HANDS ON!
Demo: Mesh partitioning

• For a demo, we’ll use the Zoltan test driver (zdrive).
• Zdrive reads data from a file and outputs a static partition.
  – Visualize the result with gnuplot.
• Designed for testing, not for users!
  – Code is ugly; do NOT use as example.
• Show mesh with different partitioning algorithms.
  – BLOCK, RCB, GRAPH, HYPERGRAPH
How to get Zoltan?

• A) Stand-alone:
  – Download tarball from Zoltan home page.

• B) As part of Trilinos:
  – Download from Trilinos web site (~35 packages).
    • [http://trilinos.sandia.gov]
  – Best if you want to use other Trilinos packages.

• You should already have Zoltan!
  – Just ‘cd zoltan’.
Configuring and Building Zoltan

• Create and enter the Zoltan directory.
  – `tar xfz zoltan_distrib_v3.1.tar.gz`
  – `cd Zoltan`

• Configure and make Zoltan library.
  – Currently two build systems:
    • Autotools (preferred)
    • Manual (fallback option if above fails…)
  – Create a build directory: `mkdir BUILD`
    • Zoltan allows multiple builds from same source.
  – `cd BUILD; ..configure <options>`
  – Then just type ‘make’!
    • `make install`
Example of configure script

```bash
../configure \
  --prefix=/home/urmel/zoltan/BUILD \ 
  --enable-mpi --with-mpi-compilers \ 
  --with-parmetis \ 
  --with-parmetis-incdir="/home/urmel/ParMETIS3_1" \ 
  --with-parmetis-libdir="/home/urmel/ParMETIS3_1"
```

Tips:
- Remember to configure in your BUILD directory.
- Use `--enable-mpi` to build for parallel execution.
- Keep configure command in a script.
- See sample scripts in `zoltan/SampleConfigureScripts`. 
How to run Zoltan?

• Recall Zoltan is “just” a library!
  – Run your app and call Zoltan.
• There is no “Hello World” for Zoltan. 😞
• Fairly simple examples in zoltan/example.
  – cd zoltan/example/C
SimpleRCB and SimpleGRAPH

- simpleRCB.c
  - Example of RCB on 5x5 regular mesh.
  - Objects to be partitioned are mesh nodes.
- simpleGRAPH.c
  - Same example, but use graph model.
- Each program has 5 phases:
  - Initialize.
  - Set parameters and callbacks.
  - Partition (call Zoltan).
  - Use the partition (e.g. move data).
  - Clean up.
/ * Initialize MPI */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

  /*
  ** Initialize application data. In this example,
  ** we split a 5*5 mesh among processors, see simpleGraph.h
  */

  /* Initialize Zoltan */
  rc = Zoltan_Initialize(argc, argv, &ver);

  if (rc != ZOLTAN_OK){
    printf("sorry...\n");
    MPI_Finalize();
    exit(0);
  }
Example zoltanRCB.c: Set Parameters and Callbacks

/* Allocate and initialize memory for Zoltan structure */
zz = Zoltan_Create(MPI_COMM_WORLD);

/* Set general parameters */
Zoltan_Set_Param(zz, "DEBUG_LEVEL", "0");
Zoltan_Set_Param(zz, "LB_METHOD", "RCB");
Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");
Zoltan_Set_Param(zz, "RETURN_LISTS", "ALL");

/* Set RCB parameters */
Zoltan_Set_Param(zz, "KEEP_CUTS", "1");
Zoltan_Set_Param(zz, "RCB_OUTPUT_LEVEL", "0");
Zoltan_Set_Param(zz, "RCB_RECTILINEAR_BLOCKS", "1");

/* Register call-back query functions
(defined in simpleQueries.h). */
Zoltan_Set_Num_Obj_Fn(zz, get_number_of_objects, NULL);
Zoltan_Set_Obj_List_Fn(zz, get_object_list, NULL);
Zoltan_Set_Num_Geom_Fn(zz, get_num_geometry, NULL);
Zoltan_Set_Geom_Multi_Fn(zz, get_geometry_list, NULL);
Example simpleRCB.c: Partitioning

Zoltan computes the difference ($\Delta$) from current distribution
Choose between:
a) Import lists (data to import from other procs)
b) Export lists (data to export to other procs)
c) Both (the default)

/* Perform partitioning */
rc = Zoltan_LB_Partition(zz,
    &changes, /* Flag indicating whether partition changed */
    &numGidEntries, &numLidEntries,
    &numImport, /* objects to be imported to new part */
    &importGlobalGids, &importLocalGids,
    &importProcs, &importToPart,
    &numExport, /* objects to be exported from old part */
    &exportGlobalGids, &exportLocalGids,
    &exportProcs, &exportToPart);
Example simpleRCB.c: Use the Partition

/* Process partitioning results;  
** in this case, just print information;  
** in a "real" application, migrate data here.  
*/

draw_partitions("initial distribution", ngids, gid_list, 1,  
wgt_list, 0);
...

/* update gid_flags from import/export lists. */
...

draw_partitions("new partitioning", nextIdx, gid_flags, 1,  
wgt_list, 0);
Example simpleRCB.c: Cleanup

/* Free Zoltan memory allocated by Zoltan_LB_Partition. */
Zoltan_LB_Free_Part(&importGlobalGids, &importLocalGids,
                     &importProcs, &importToPart);
Zoltan_LB_Free_Part(&exportGlobalGids, &exportLocalGids,
                     &exportProcs, &exportToPart);

/* Free Zoltan memory allocated by Zoltan_Create. */
Zoltan_Destroy(&zz);

/********************
** all done **       
********************/

MPI_Finalize();
Compile and Run it!

- We could use autotooled makefiles.
- But let’s do it “from scratch.”

```
mpicc simpleRCB.c -o simpleRCB
-I/home/urmel/zoltan/BUILD/include/
-L/home/urmel/zoltan/BUILD/lib -lzoltan
-L/home/urmel/ParMETIS3_1 -lparmetis
-lmetis

mpirun -np 2 simpleRCB
```
For geometric partitioning (RCB, RIB, HSFC), use …

<table>
<thead>
<tr>
<th>General Query Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ZOLTAN_NUM_OBJ_FN</strong></td>
</tr>
<tr>
<td><strong>ZOLTAN_OBJ_LIST_FN</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometric Query Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ZOLTAN_NUM_GEOM_FN</strong></td>
</tr>
<tr>
<td><strong>ZOLTAN_GEOM_FN</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypergraph Query Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ZOLTAN_HG_SIZE_CS_FN</strong></td>
</tr>
<tr>
<td><strong>ZOLTAN_HG_CS_FN</strong></td>
</tr>
<tr>
<td><strong>ZOLTAN_HG_SIZE_EDGE_WTS_FN</strong></td>
</tr>
<tr>
<td><strong>ZOLTAN_HG_EDGE_WTS_FN</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Query Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ZOLTAN_NUM_EDGE_FN</strong></td>
</tr>
<tr>
<td><strong>ZOLTAN_EDGE_LIST_FN</strong></td>
</tr>
</tbody>
</table>
/*********************************************************
 * Prototype: ZOLTAN_NUM_OBJ_FN
 * Return the number of objects I own.
 ***********************************************************/

* Zoltan partitions a collection of objects distributed
* across processes. In this example objects are vertices.
* They are dealt out like cards based on process rank.
*/

static int get_number_of_objects(void *data, int *ierr)
{
    int i, numobj=0;

    for (i=0; i<simpleNumVertices; i++){
        if (i % numProcs == myRank) numobj++;
    }

    *ierr = ZOLTAN_OK;

    return numobj;
}
Example simpleRCB.c: ZOLTAN_OBJ_LIST_FN

```c
void get_object_list(void *userData,
                     int sizeGID, int sizeLID,
                     ZOLTAN_ID_PTR globalID,
                     ZOLTAN_ID_PTR localID,
                     int wgt_dim, float *obj_wgts,
                     int *err)
{
    int i, next;
    if (sizeGID != 1){ /* My global IDs are 1 integer */
        *ierr = ZOLTAN_FATAL;
        return;
    }
    for (i=0, next=0; i<simpleNumVertices; i++){
        if (i % numProcs == myRank){
            globalID[next] = i+1; /* application wide global ID */
            localID[next] = next; /* process specific local ID */
            obj_wgts[next] = (float)simpleNumEdges[i]; /* weight */
            next++;
        }
    }
    *ierr = ZOLTAN_OK;
    return;
}
```
**Example simpleRCB.c:**

**ZOLTAN_GEOM_MULTI_FN**

```c
void get_geometry_list(void *data, int sizeGID, int sizeLID,
                        int num_obj,
                        ZOLTAN_ID_PTR globalID, ZOLTAN_ID_PTR localID,
                        int num_dim, double *geom_vec, int *ierr)
{
    int i;
    int row, col;

    for (i=0;  i < num_obj ; i++){
        row = (globalID[i] - 1) / 5;
        col = (globalID[i] - 1) % 5;

        geom_vec[2*i] = (double)col;
        geom_vec[2*i+1] = (double)row;
    }

    *ierr = ZOLTAN_OK;
    return;
}
```
Example: simpleGRAPH.c

• Same example, but now use graph partitioning.
• Changes needed:
  – Zoltan_Set_Param(zz, “LB_METHOD”, “GRAPH”);
  – Set method-specific parameters (optional).
  – Register graph call-back query functions.
  – Everything else stays the same!
For graph partitioning, coloring & ordering, use ...

<table>
<thead>
<tr>
<th>General Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_OBJ_FN</td>
<td>Number of items on processor</td>
</tr>
<tr>
<td>ZOLTAN_OBJ_LIST_FN</td>
<td>List of item IDs and weights.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometric Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_GEOM_FN</td>
<td>Dimensionality of domain.</td>
</tr>
<tr>
<td>ZOLTAN_GEOM_FN</td>
<td>Coordinates of items.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypergraph Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_HG_SIZE_CS_FN</td>
<td>Number of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_CS_FN</td>
<td>List of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_SIZE_EDGE_WTS_FN</td>
<td>Number of hyperedge weights.</td>
</tr>
<tr>
<td>ZOLTAN_HG_EDGE_WTS_FN</td>
<td>List of hyperedge weights.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_EDGE_FN</td>
<td>Number of graph edges.</td>
</tr>
<tr>
<td>ZOLTAN_EDGE_LIST_FN</td>
<td>List of graph edges and weights.</td>
</tr>
</tbody>
</table>
Changing Partitioner of Same Type

- By default, “GRAPH” uses Zoltan’s native graph/hypergraph partitioner.
- To use ParMetis or Scotch:
  - `Zoltan_Set_Param(zz, “GRAPH_PACKAGE", "PARMETIS")`;
  - `Zoltan_Set_Param(zz, “GRAPH_PACKAGE", "SCOTCH")`;
  - Define third-party libraries at configure time.
- Single parameter to switch partitioner.
  - Try it!
For More Information...

- **Zoltan Home Page**
  - User’s and Developer’s Guides
  - Download Zoltan software under GNU LGPL.

- **Email:**
  - zoltan-users@software.sandia.gov
  - {kddevin, ccheval, egboman}@sandia.gov
  - umit@bmi.osu.edu
The End
More Details on Query Functions

• **void* data pointer** allows user data structures to be used in all query functions.
  – To use, cast the pointer to the application data type.

• **Local IDs** provided by application are returned by Zoltan to simplify access of application data.
  – E.g. Indices into local arrays of coordinates.

• **ZOLTAN_ID_PTR** is pointer to array of unsigned integers, allowing IDs to be more than one integer long.
  – E.g., (processor number, local element number) pair.
  – numGlobalIds and numLocalIds are lengths of each ID.

• All memory for query-function arguments is allocated in Zoltan.

```c
void ZOLTAN_GET_GEOM_MULTI_FN(void *userDefinedData,
                               int numGlobalIds, int numLocalIds, int numObjs,
                               ZOLTAN_ID_PTR gids, ZOLTAN_ID_PTR lids,
                               int numDim, double *pts, int *err)
```
Zoltan Data Migration Tools

- After partition is computed, data must be moved to a new decomposition.
  - Depends strongly on application data structures.
  - Complicated communication patterns.
- Zoltan can help!
  - Application supplies query functions to pack/unpack data.
  - Zoltan does all communication to new processors.
Using Zoltan’s Data Migration Tools

• Required migration query functions:
  – ZOLTAN_OBJ_SIZE_MULTI_FN:
    • Returns size of data (in bytes) for each object to be exported to a new processor.
  – ZOLTAN_PACK_MULTI_FN:
    • Remove data from application data structure on old processor;
    • Copy data to Zoltan communication buffer.
  – ZOLTAN_UNPACK_MULTI_FN:
    • Copy data from Zoltan communication buffer into data structure on new processor.

int Zoltan_Migrate(struct Zoltan_Struct *zz,
  int num_import, ZOLTAN_ID_PTR import_global_ids,
  ZOLTAN_ID_PTR import_local_ids, int *import_procs,
  int *import_to_part,
  int num_export, ZOLTAN_ID_PTR export_global_ids,
  ZOLTAN_ID_PTR export_local_ids, int *export_procs,
  int *export_to_part);
Zoltan Unstructured Communication Package

- **Simple primitives for efficient irregular communication.**
  - **Zoltan_Comm_Create**: Generates communication plan.
    - Processors and amount of data to send and receive.
  - **Zoltan_Comm_Do**: Send data using plan.
    - Can reuse plan. (Same plan, different data.)
  - **Zoltan_Comm_Do_Reverse**: Inverse communication.
- **Used for most communication in Zoltan.**
  - Similar to BSP model.
Example Application: Crash Simulations

- **Multiphase simulation:**
  - Graph-based decomposition of elements for finite element calculation.
  - Dynamic geometric decomposition of surfaces for contact detection.
  - Migration tools and Unstructured Communication package map between decompositions.
• Helps applications locate off-processor data.
  – Zoltan does not keep track of user data.
• Rendezvous algorithm (Pinar, 2001).
  – Directory distributed in known way (hashing) across processors.
  – Requests for object location sent to processor storing the object’s directory entry.
For hypergraph partitioning and repartitioning, use …

<table>
<thead>
<tr>
<th>General Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_OBJ_FN</td>
<td>Number of items on processor</td>
</tr>
<tr>
<td>ZOLTAN_OBJ_LIST_FN</td>
<td>List of item IDs and weights.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometric Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_GEOM_FN</td>
<td>Dimensionality of domain.</td>
</tr>
<tr>
<td>ZOLTAN_GEOM_FN</td>
<td>Coordinates of items.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypergraph Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_HG_SIZE_CS_FN</td>
<td>Number of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_CS_FN</td>
<td>List of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_SIZE_EDGE_WTS_FN</td>
<td>Number of hyperedge weights.</td>
</tr>
<tr>
<td>ZOLTAN_HG_EDGE_WTS_FN</td>
<td>List of hyperedge weights.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_EDGE_FN</td>
<td>Number of graph edges.</td>
</tr>
<tr>
<td>ZOLTAN_EDGE_LIST_FN</td>
<td>List of graph edges and weights.</td>
</tr>
</tbody>
</table>
Or can use graph queries to build hypergraph.

<table>
<thead>
<tr>
<th>General Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_OBJ_FN</td>
<td>Number of items on processor</td>
</tr>
<tr>
<td>ZOLTAN_OBJ_LIST_FN</td>
<td>List of item IDs and weights.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Geometric Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_GEOM_FN</td>
<td>Dimensionality of domain.</td>
</tr>
<tr>
<td>ZOLTAN_GEOM_FN</td>
<td>Coordinates of items.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hypergraph Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_HG_SIZE_CS_FN</td>
<td>Number of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_CS_FN</td>
<td>List of hyperedge pins.</td>
</tr>
<tr>
<td>ZOLTAN_HG_SIZE_EDGE_WTS_FN</td>
<td>Number of hyperedge weights.</td>
</tr>
<tr>
<td>ZOLTAN_HG_EDGE_WTS_FN</td>
<td>List of hyperedge weights.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph Query Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ZOLTAN_NUM_EDGE_FN</td>
<td>Number of graph edges.</td>
</tr>
<tr>
<td>ZOLTAN_EDGE_LIST_FN</td>
<td>List of graph edges and weights.</td>
</tr>
</tbody>
</table>
Variations on RCB:
Recursive Inertial Bisection

- \texttt{Zoltan\_Set\_Param(zz, \texttt{“LB\_METHOD”}, \texttt{“RIB”});}
- Simon, Taylor, et al., 1991
- Cutting planes orthogonal to principle axes of geometry.
- Not incremental.
Space-Filling Curve Partitioning (SFC)

- **Zoltan_Set_Param(zz, “LB_METHOD”, “HSFC”);**
- **Space-Filling Curve (Peano, 1890):**
  - Mapping between $R^3$ to $R^1$ that completely fills a domain.
  - Applied recursively to obtain desired granularity.
- **Used for partitioning by ...**
  - Pilkington and Baden, 1994, smoothed particle hydrodynamics.
  - Patra and Oden, 1995, adaptive mesh refinement.
SFC Algorithm

• Run space-filling curve through domain.
• Order objects according to position on curve.
• Perform 1-D partition of curve.
SFC Advantages and Disadvantages

• Advantages:
  – Simple, fast, inexpensive.
  – Maintains geometric locality of objects in processors.
  – All processors can inexpensively know entire partition (e.g., for global search in contact detection).
  – Implicitly incremental for repartitioning.

• Disadvantages:
  – No explicit control of communication costs.
  – Can generate disconnected subdomains.
  – Often lower quality partitions than RCB.
  – Geometric coordinates needed.
Applications using SFC

• Adaptive hp-refinement finite element methods.
  – Assigns physically close elements to same processor.
  – Inexpensive; incremental; fast.
  – Linear ordering can be used to order elements for efficient memory access.
Auxiliary Capabilities for Geometric Methods

- Zoltan can store cuts from RCB, RIB, and HSFC inexpensively in each processor.
  - `Zoltan_Set_Param(zz, “KEEP_CUTS”, “1”);`
- Enables parallel geometric search without communication.
  - Useful for contact detection, particle methods, rendering.

Determine the part/processor owning region with a given point.
`Zoltan_LB_Point_PP_Assign`

Determine all parts/processors overlapping a given region.
`Zoltan_LB_Box_PP_Assign`
Distance-2 Graph Coloring

• Problem (NP-hard)
  Color the vertices of a graph with as few colors as possible such that a pair of vertices connected by a path on two or less edges receives different colors.

• Applications
  – Derivative matrix computation in numerical optimization
  – Channel assignment
  – Facility location

• Related problems
  – Partial distance-2 coloring
  – Star coloring
Nested dissection (1)

- **Principle [George 1973]**
  - Find a vertex separator $S$ in graph.
  - Order vertices of $S$ with highest available indices.
  - Recursively apply the algorithm to the two separated subgraphs $A$ and $B$. 

![Diagram showing nested dissection with separator S and subgraphs A and B.](image-url)
Nested dissection (2)

• Advantages:
  – Induces high quality block decompositions.
    • Suitable for block BLAS 3 computations.
  – Increases the concurrency of computations.
    • Compared to minimum degree algorithms.
    • Very suitable for parallel factorization.
      – The ordering itself can be computed in parallel.
Zoltan ordering architecture

User

Zoltan

Parmetis
- Ordering
- Partitioning

PT-Scotch
- Ordering
- Partitioning when available

Others ...

third Party Library interface
Experimental results (1)

- Metric is OPC, the operation count of Cholesky factorization.
- Largest matrix ordered by PT-Scotch: 83 millions of unknowns on 256 processors (CEA/CESTA).
- Some of our largest test graphs.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Size (x1000)</th>
<th>Average degree</th>
<th>$O_{SS}$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>V</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>audikw1</td>
<td>944</td>
<td>38354</td>
<td>81.28</td>
<td>5.48E+12  3D mechanics mesh, Parasol</td>
</tr>
<tr>
<td>cage15</td>
<td>5154</td>
<td>47022</td>
<td>18.24</td>
<td>4.06E+16  DNA electrophoresis, UF</td>
</tr>
<tr>
<td>quimonda07</td>
<td>8613</td>
<td>29143</td>
<td>6.76</td>
<td>8.92E+10  Circuit simulation, Quimonda</td>
</tr>
<tr>
<td>23millions</td>
<td>23114</td>
<td>175686</td>
<td>7.6</td>
<td>1.29E+14  CEA/CESTA</td>
</tr>
</tbody>
</table>
### Experimental results (3)

<table>
<thead>
<tr>
<th>Test case</th>
<th>Number of processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>cage15</td>
<td></td>
</tr>
<tr>
<td>$O_{PTS}$</td>
<td>4.58E+16</td>
</tr>
<tr>
<td>$O_{PM}$</td>
<td>4.47E+16</td>
</tr>
<tr>
<td>$t_{PTS}$</td>
<td>540.46</td>
</tr>
<tr>
<td>$t_{PM}$</td>
<td>195.93</td>
</tr>
</tbody>
</table>

**Graphs:**

- Left graph: Comparison of PTScotch, ParMetis, and Seq. Scotch for $NNZ(L)/NNZ(A)$.
- Right graph: Comparison of PTScotch, ParMetis, and Seq. Scotch for OPC, base 1.0 for sequential Scotch.
Experimental results (4)

- ParMETIS crashes for all other graphs.

<table>
<thead>
<tr>
<th>Test case</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OPTS</strong></td>
<td></td>
<td></td>
<td>5.80E+10</td>
<td>6.38E+10</td>
<td>6.94E+10</td>
<td>7.70E+10</td>
</tr>
<tr>
<td><strong>tPTS</strong></td>
<td></td>
<td></td>
<td>34.68</td>
<td>22.23</td>
<td>17.30</td>
<td>16.62</td>
</tr>
</tbody>
</table>

- 23millions

| **OPTS**  | 1.45E+14 | 2.91E+14 | 3.99E+14 | 2.71E+14 | 1.94E+14 | 2.45E+14 |
| **tPTS**  | 671.60    | 416.45    | 295.38    | 211.68    | 147.35    | 103.73    |
void ZOLTAN_NUM_EDGES_MULTI_FN(void *data,
   int num_gid_entries, int num_lid_entries,
   int num_obj, ZOLTAN_ID_PTR global_id, ZOLTAN_ID_PTR local_id,
   int *num_edges, int *ierr);

Proc 0 Input from Zoltan:
   num_obj = 3
   global_id = {A,C,B}
   local_id = {0,1,2}

Output from Application on Proc 0:
   num_edges = {2,4,3}
      (i.e., degrees of vertices A, C, B)
   ierr = ZOLTAN_OK
Example Graph Callbacks

```c
void ZOLTAN_EDGE_LIST_MULTI_FN(void *data,
    int num_gid_entries, int num_lid_entries,
    int num_obj, ZOLTAN_ID_PTR global_ids, ZOLTAN_ID_PTR local_ids,
    int *num_edges,
    ZOLTAN_ID_PTR nbor_global_id, int *nbor_procs,
    int wdim, float *nbor_ewgts,
    int *ierr);
```

**Proc 0 Input from Zoltan:**
- `num_obj = 3`
- `global_ids = {A, C, B}`
- `local_ids = {0, 1, 2}`
- `num_edges = {2, 4, 3}`
- `wdim = 0` or `EDGE_WEIGHT_DIM` parameter value

**Output from Application on Proc 0:**
- `nbor_global_id = {B, C, A, B, E, D, A, C, D}`
- `nbor_procs = {0, 0, 0, 0, 1, 1, 0, 0, 1}`
- `nbor_ewgts = if wdim then {7, 8, 8, 9, 1, 3, 7, 9, 5}`
- `ierr = ZOLTAN_OK`
Example Hypergraph Callbacks

```c
void ZOLTAN_HG_SIZE_CS_FN(void *data, int *num_lists, int *num_pins,
                          int *format, int *ierr);
```

Output from Application on Proc 0:
- `num_lists = 2`
- `num_pins = 6`
- `format = ZOLTAN_COMPRESSED_VERTEX`
  (owned non-zeros per vertex)
- `ierr = ZOLTAN_OK`

OR

Output from Application on Proc 0:
- `num_lists = 5`
- `num_pins = 6`
- `format = ZOLTAN_COMPRESSED_EDGE`
  (owned non-zeros per edge)
- `ierr = ZOLTAN_OK`

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hyperedges</th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>a</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>b</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>c</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>d</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>e</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>f</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Example Hypergraph Callbacks

void ZOLTAN_HG_CS_FN(void *data, int num_gid_entries,
int nvtxedge, int npins, int format,
ZOLTAN_ID_PTR vtxedge_GID, int *vtxedge_ptr, ZOLTAN_ID_PTR pin_GID,
int *ierr);

Proc 0 Input from Zoltan:
    nvtxedge = 2 or 5
    npins = 6
    format = ZOLTAN_COMPRESSED_VERTEX or
    ZOLTAN_COMPRESSED_EDGE

Output from Application on Proc 0:
    if (format = ZOLTAN_COMPRESSED_VERTEX)
        vtxedge_GID = {A, B}
        vtxedge_ptr = {0, 3}
        pin_GID = {a, e, f, b, d, f}
    if (format = ZOLTAN_COMPRESSED_EDGE)
        vtxedge_GID = {a, b, d, e, f}
        vtxedge_ptr = {0, 1, 2, 3, 4}
        pin_GID = {A, B, B, A, A, B}
    ierr = ZOLTAN_OK
Performance Results

• Experiments on Sandia’s Thunderbird cluster.
  – Dual 3.6 GHz Intel EM64T processors with 6 GB RAM.
  – Infiniband network.
• Compare RCB, HSFC, graph and hypergraph methods.
• Measure …
  – Amount of communication induced by the partition.
  – Partitioning time.
Test Data

SLAC *LCLS
Radio Frequency Gun
6.0M x 6.0M
23.4M nonzeros

Xyce 680K ASIC Stripped Circuit Simulation
680K x 680K
2.3M nonzeros

Cage15 DNA Electrophoresis
5.1M x 5.1M
99M nonzeros

SLAC Linear Accelerator
2.9M x 2.9M
11.4M nonzeros
Communication Volume: Lower is Better

**SLAC 6.0M LCLS**

- **Number of parts = number of processors.**

**SLAC 2.9M Linear Accelerator**

**Xyce 680K circuit**

**Cage15 5.1M electrophoresis**
Partitioning Time: Lower is better

SLAC 6.0M LCLS

1024 parts. Varying number of processors.

Xyce 680K circuit

SLAC 2.9M Linear Accelerator

RCB
HSFC
Graph
Hypergraph

Cage15 5.1M electrophoresis
Repartitioning Experiments

- Experiments with 64 parts on 64 processors.
- Dynamically adjust weights in data to simulate, say, adaptive mesh refinement.
- Repartition.
- Measure repartitioning time and total communication volume:
  
  Data redistribution volume
  + Application communication volume
  
  Total communication volume
Repartitioning Results: Lower is Better

SLAC 6.0M LCLS

Xyce 680K circuit

Repartitioning Time (secs)