ZOLTAN DATA MANAGEMENT SERVICES FOR PARALLEL DYNAMIC APPLICATIONS

By Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan

THE ZOLTAN LIBRARY IS A COLLECTION OF DATA MANAGE-MENT SERVICES FOR PARALLEL, UNSTRUCTURED, ADAPTIVE, AND DYNAMIC APPLICATIONS THAT IS AVAILABLE AS OPEN-SOURCE SOFTWARE FROM WWW.CS.SANDIA.GOV/ZOLTAN. IT

simplifies the load-balancing, data movement, unstructured-communication, and memory usage difficulties that arise in dynamic applications such as adaptive finite-element methods, particle methods, and crash simulations. Zoltan's data-structure-neutral design also lets a wide range of applications use it without imposing restrictions on application data structures. Its object-based interface provides a simple and inexpensive way for application developers to use the library and researchers to make new capabilities available under a common interface.

Zoltan provides tools that help application developers without imposing strict requirements on them. For example, it includes a suite of parallel partitioning algorithms and data migration tools that redistribute data to reflect, say, changing processor workloads. Zoltan also includes distributed data directories and unstructured communication services that let applications perform complicated communication using only a few simple primitives. To simplify debugging of dynamic memory usage, Zoltan provides dynamic memory management tools that enhance common memory allocation functions. In this article, we describe Zoltan's features and ways to use it in dynamic applications.

Zoltan software design

Our design of the Zoltan library does not restrict it to any particular type of application. Rather, Zoltan operates on uniquely identifiable data items that we call *objects*. For example, in finite-element applications, objects might be elements or nodes of the mesh. In particle applications, objects might be particles. In linear solvers, objects might be matrix rows. Each object must have a unique global identifier (ID) represented as an array of unsigned integers. Common choices include global numbers for elements (nodes, rows, particles, and so on) that already exist in many applications, or a structure consisting of an owning processor number and the object's local-memory index. Objects might also have local (to a processor) IDs that do not have to be unique globally. Local IDs such as addresses or local-array indices of objects can improve the performance (and convenience) of Zoltan's interface to applications.

To make Zoltan easy to use, we do not impose any particular data structure on an application, nor do we require an application to build a particular data structure for Zoltan. Instead, Zoltan uses a callback function interface in which the tool queries the ap-

plication for needed data. The application must provide simple functions that answer these queries.

For example, Figure 1 shows how Zoltan's callback function interface works for performing dynamic load balancing in an application. An application starts Zoltan (`Zoltan_Initialize`) and allocates the memory it needs (`Zoltan_Create`). Through calls to `Zoltan_Set_Fn`, the application passes pointers to its callback functions to Zoltan. It also selects a partitioning method (`Zoltan_Set_Method`) and sets appropriate parameters for load balancing(`Zoltan_LB_Set_Param`). Then, within the main computation loop, the application calls `Zoltan_LB_Balance` to compute a new partition of its data.

As a first step in `Zoltan_LB_Balance`, Zoltan must build the data structures needed for the particular partitioning method selected. It calls the

Zoltan Details

The Zoltan library's toolkit includes parallel partitioning algorithms, data migration tools, distributed directories, and unstructured communication and memory management packages. It has many key features for developers:

- Its source code is freely available at www.cs.sandia.gov/Zoltan.
- It's callable from C, C++, and F90 (but is implemented in C).
- It uses MPI communication.

Café Dubois

XP Torture Test

Both my kids' computers had a Windows XP upgrade coming to them. My daughter is at college where I can't help her when things break, and my son is an operating system programmer's worst nightmare. Therefore I decided, despite my opposition to the XP "activation" requirement, to do the upgrades because of the alleged improvement in reliability. There aren't many areas of life like this one, where we buy another product from someone because the last one they sold us is so defective.

My daughter brought her Vaio home for Christmas. Sony's upgrade included a second CD that prepared the machine, instructed you to do the upgrade, and then returned to replace even more of your software. They did a good job and it all worked, but it took about two hours. For some reason, it did not install the network configuration correctly, somehow disabling Dubois Castlenet and angering an impatient mob.

In my son's case, he had worked Windows Me into an interesting state where every attempt to open a folder crashed Explorer. Figuring that the upgrade was not likely to work very well in such an environment, I reformatted his drive and reinstalled from his original Windows 98-2e OEM disks. After the usual search team was sent out to find the product key, I was able to start the 98-2e -> XP upgrade. An additional 90 minutes of sheer boredom punctuated by moments of terror later, I had a working machine with no sound. Another two hours were spent figuring that out.

Both machines later announced that the DVD decoder was toast and that I should click here to go get a new one. My son's manufacturer charged me $30 for it. I was too tired to sue.

A friend who went through upgrading on his machine said he only got "a few heart-stopping blue screens of death" during the process.

Cavaet upgraditor. Carpe monopolist.

Building

I spend a lot of time at work wrestling with our build. Our product is an open-source, Python-based set of tools for scientific analysis and visualization with a special emphasis on climate data (cdat.sf.net). As such it spans many directories and also requires the building software we need such as Python, zlib, Tcl, Tk, and readline on a variety of machines. Each of these pieces has its own configuration procedures.

Until Python got its "Distutils" that help you build Python packages across platforms, things were in an untenable mess. But now we can build our packages pretty well with just a few frightening problems. Most of these problems fall into one of these categories:

- Headers and libraries from the software we use, such as X11, do not have standardized locations.
- Some OSs include different things than others.
- Some OSs come with versions of things that we use but they are older versions that we cannot use and must somehow avoid.
- Command lines for compilers and linkers are not standardized.
- Make is not standardized.
- Many users don't read the README before installing.
- Cross-platform portability to Windows and Mac seems even farther out of reach.

Here are some resources for dealing with this problem:

- *Autoconf* (www.gnu.org/manual/autoconf) is widely used but is hard to learn and use; too hard for scientists, I think. Am I wrong?
- *Imake* (www.dubois.ws/software/imake-stuff) was something my group tried for a year or so but again it was too hard for scientists. (This Web site by Paul Dubois is not my Web site. As Yoda said, there is *another*.)
- *Cons* (www.dsmit.com/cons) is a Perl-based system that some people think highly of.
- *SCons* (www.scons.org) is an implementation of the winning design for a build tool in the Software Carpentry contest. It is just getting going. I think it has a lot of potential because it lets you write the configuration file (the equivalent of a Makefile) in Python. SCons is supposed to "fix what is wrong with Cons." I tried SCons and it built a C code on Windows without a problem. That is usually a terror-filled process without building a Visual Studio project.
- *JAM / MR* (perforce.com/jam/jam.html) is another system that has attracted an enthusiastic audience. I haven't had time to look into it, but the same people wrote Perforce, my favorite source-code control system, so I know they live in the real world of having a complex multiplatform product.

Send me your tales of woe and success and maybe we can together figure out what works. Meantime, I'll time how long it takes my son to destroy XP.

Application

Initialize Zoltan
(Zoltan_Initialize,
Zoltan_Create)

Select partitioning method
(Zoltan_LB_Set_Method,
Zoltan_Set_Params)

Register callback functions
(Zoltan_Set_Fn)

Compute

Repartition
(Zoltan_LB_Balance)

Move data
(Zoltan_Help_Migrate)

Clean up
(Zoltan_Destroy)

Zoltan

Zoltan_LB_Balance:
• Call registered callback function
• Build data structures
  for method
• Compute new decomposition
• Return import/export lists

Zoltan_Help_Migrate:
• Call packing callback function for
  exports
• Send exports
• Receive imports
• Call unpacking callback function
  for imports

Figure 1. An example shows how to use Zoltan's callback function interface for parallel partitioning in applications. The application (left) initializes Zoltan, selects a partitioning method, and passes pointers to its callback functions to Zoltan. When parallel partitioning is invoked, Zoltan (right) uses the callback functions to build its data structures to perform partitioning. A similar callback function interface helps migrate data between processors after a new partition is computed.

registered callback functions to build its data structures. It then performs the partitioning and returns lists of objects to be imported to and exported from the processor for the new decomposition. At no time does the application developer have to build (or debug) complicated data structures for use within Zoltan.

To keep the application interface simple, we use a small set of callback functions and make them easy to write by requesting only information that is easily accessible to applications. For the most basic partitioning algorithms, Zoltan requires only four callback functions. These functions return the number of objects owned by a processor, a list of weights and IDs for owned objects, the problem's dimensionality, and a given object's coordinates. (Figure 2 includes simple examples of some of these callback functions.) More sophisticated graph-based partitioning

algorithms require only two additional callback functions, which return the number of edges per object and edge lists for objects.

Zoltan's parallel data services

Zoltan provides a toolkit of parallel data services for dynamic and unstructured applications. These services are layered in Zoltan so that application developers can use as much or as little of the toolkit as desired. Zoltan's parallel partitioning tools are efficient algorithms for dividing an application's data among processors while trying to minimize interprocessor communication. *Incremental* partitioning algorithms—algorithms that account for data's current location in determining their new location—are included to keep data movement costs low. Zoltan also includes data migration utilities, distributed data directories, an unstructured

communication package, and a dynamic memory management package.

Parallel partitioning and dynamic load balancing

In our experience, no single partitioning strategy is effective for all parallel computations. Some applications require partitions based only on the problem's workloads and geometry; others benefit from explicit consideration of dependencies between objects. Some applications require the highest-quality partitions possible, regardless of the cost to generate them; others can sacrifice some quality so long as new partitions can be generated quickly. For some applications, the cost to relocate data is prohibitively high, so incremental partitioning algorithms are needed; other applications can tolerate greater remapping costs. Most important, application developers might not know in advance which strategies work best in their applications, so they need a convenient means of comparing algorithms.

We provide three classes of parallel partitioning algorithms in the Zoltan library: geometric bisection, space-filling curves, and graph partitioning. Each class includes several different algorithms. Once users write the callback functions for each class, switching between classes and methods requires only a call to Zoltan_LB_Set_Method with the new algorithm name. In this way, developers can easily compare algorithms within their applications to find the strategy that works best for them.

Geometric bisection. Recursive coordinate bisection is conceptually the simplest partitioning algorithm in Zoltan.[1] RCB divides a problem's work into two equal parts using a cutting plane orthogonal to a coordinate axis and assigns objects to subdomains based on their geometric positions relative to the cutting plane (see Figure 3). It then recur-

```
int Number_Owned_Nodes;
struct Node_Type {

  double Coordinates[3];
  float Weight;
  int Global_ID_Num;
 } Nodes[MAX_NODES];

void owned_objects_callback(void *data,
  int num_gid_entries, int num_lid_entries;
  ZOLTAN_ID_PTR global_ids, ZOLTAN_ID_PTR local_ids,
  int wgt_dim, float *obj_wgts,
  int *ierr)
{
int i;
  /* return global node numbers as global_ids. */
  /* return index into Nodes array for local_ids. */
  for (i = 0; i < Number_Owned_Nodes; i++)[
    global_ids[i] = Nodes[i].Global_ID_Num;
    local_ids[i] = i;
    obj_wgts[i] =  Nodes[i].Weight;
  }
  *ierr = ZOLTAN_OK;
}

void coordinates_callback(void *data,
  int num_gid_entries, int num_lid_entries,
  ZOLTAN_ID_PTR global_id, ZOLTAN_ID_PTR local_id,
  double *geom_vec, int *ierr)
{
  /* use local_id to index into the Nodes array. */
  geom_vec[0] = Nodes[local_id[0]].Coordinates[0];
  geom_vec[1] = Nodes[local_id[0]].Coordinates[1];
  geom_vec[2] = Nodes[local_id[0]].Coordinates[2];
  *ierr = ZOLTAN_OK;
}
```

Figure 2. Example callback functions for a simple mesh-based application. Finite-element nodes are the objects to be passed to Zoltan. The function `owned_objects_callback` returns a list of node weights and IDs for each node owned by a processor. The function `coordinates_callback` returns a given node's coordinates.

sively cuts the resulting subdomains until the number of subdomains is equal to the number of processors. Recursive inertial bisection (RIB) is a variant of RCB that chooses cutting planes orthogonal to the principle axes of the geometry, rather than to the coordinate axes.[2,3]

RCB is effective for many applications because it maintains geometric locality of objects in processors; it is also fast and implicitly incremental (small changes in workloads produce only small changes in the decomposition). It is widely used in crash simulations and particle simulations because it assigns physically close surfaces or particles to a single processor. Figure 4 shows an adaptive mesh-refinement application that uses RCB to repartition after local refinement. RCB assigns parent and child elements to the same processor, preventing the need for communication between mesh levels.

Space-filling curves (SFC). Researchers have used space-filling curves (invented in the 1890s by Peano and Hilbert) for partitioning in gravitational simulations, smoothed-particle hydrodynamics, and adaptive finite-element methods. Like RCB, SFC partitioners rely only on the weights and geometric coordinates of objects. Figure 5 shows how the SFC algorithm works. The SFC partitioner refines the computational domain in Figure 5a into subregions of a desired granularity (for example, one subregion per object). A subregion's weight is the sum of the workloads of all objects in that subregion. The partitioner then passes an SFC through the subregions, mapping a multidimensional domain to a one-dimensional line (see Figure 5b). The partitioner cuts the line into equally weighted segments and assigns all objects along a segment to a single processor (see Figure 5c). Like RCB, SFC partitioning is fast and incremental and assigns geometrically close objects to a single processor.
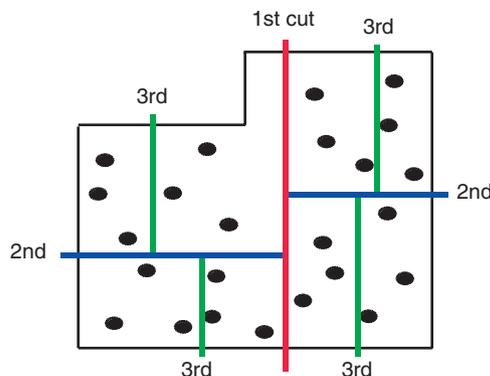

Figure 3. A diagram of cuts made during recursive coordinate bisection partitioning (RCB). Recursive cuts orthogonal to the coordinate axes divide a domain's work into evenly sized subdomains.

Zoltan includes several variants of SFC partitioning. Octree partitioning explicitly builds the tree representing the refinement of a domain into subregions; traversals of the tree produce SFCs.[4] Binned SFC partitioning effectively lets several objects exist in a single subregion; adaptive refinement of the domain efficiently obtains the desired granularity (see the user's guide at www.cs.sandia.gov/Zoltan). Refinement tree partitioning is designed specifically for adaptive mesh-refinement applications.[5] It uses parent–child relationships rather than geometric coordinates to construct a tree that re-
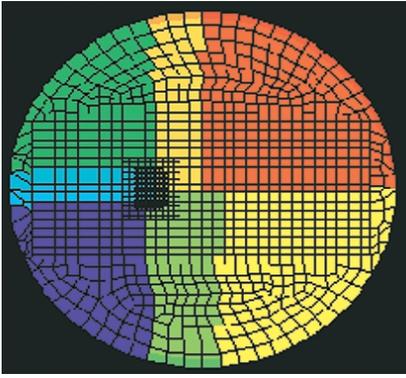
Figure 4. An RCB decomposition for an adaptive mesh-refinement application. Colors indicate processor assignments. (Image courtesy of Carter Edwards and his colleagues at Sandia National Laboratories.)

flects the domain's refinement; traversals of this tree also produce SFCs.

Graph partitioning. In graph partitioning algorithms, the problem to be partitioned is represented as a weighted graph. The graph's weighted nodes represent the objects to be partitioned (see Figure 6). Graph edges represent dependencies between objects and serve as an approximation of communication costs. Graph partitioning algorithms divide the graph to assign roughly equal nodal weight to partitions while attempting to minimize the weight of edges cut by partition boundaries.

Zoltan provides graph partitioning through interfaces to the popular graph-partitioning packages ParMETIS and Jostle.[6,7] Both ParMETIS and Jostle provide parallel implementations of the multilevel graph-partitioning approach pioneered in Chaco.[8,9] Multilevel algorithms first coarsen an input graph, with coarse nodes representing clusters of input-graph nodes. The coarse graph is easy to partition. The multilevel algorithms then project this coarse partition back to the input graph, with local improvements of the partition performed at each finer graph level. This approach produces high-quality partitions for many applications (see Figure 7). However, multilevel graph partitioning is more expensive than the geometric methods described earlier, and the partitions produced are not incremental.

ParMETIS and Jostle also include diffusive graph-partitioning algorithms.[10] Diffusive algorithms move work from heavily loaded processors to more lightly loaded neighboring ones. The problem's graph helps determine which objects should be moved to which neighbors. Diffusive algorithms generate partitions whose quality can degrade over several invocations of the partitioning algorithm. However, they are fast and incremental and can be effective for several dynamic applications.

Data migration tools

A complicated part of dynamic repartitioning is the need to move data from old processors to new ones. This data migration requires deletions and insertions from the application data structures as well as communication between the processors. A general-purpose library such as Zoltan can do little to help manipulate application data structures, but it can help communicate object data among processors.

To help an application with data migration, Zoltan requires an application to supply a callback function that packs a given object's data into a communication buffer. A function that unpacks an object's data from a communication buffer must also be supplied by the application. Zoltan uses these callback functions just as it used callback functions for repartitioning in Figure 1. For each object to be exported, Zoltan calls the packing function to load commu-

**(a)**   **(b)**   **(c)**
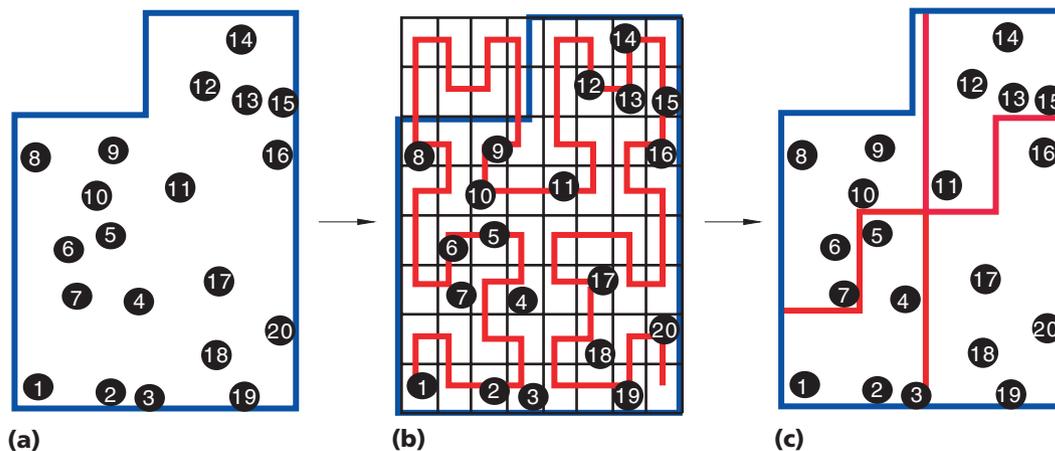
Figure 5. Space-filling curve partitioning for four processors: (a) the original domain with particles to be partitioned; (b) the refined domain with space-filling curve (red) and linear ordering of particles; and (c) a partition of the linear ordering to assign an equal number of particles to processors.

nication buffers, performs all communication needed to move the data, and then calls the unpacking function to load that object's data into the data structures on its new processor.

Distributed data directories

Dynamic applications often need to locate off-processor information. After repartitioning, a processor might need to rebuild ghost cells and lists of objects to be communicated. It might know which objects it needs but not where they are. To help locate off-processor data, Zoltan includes a distributed data directory algorithm based on Ali Pinar's rendezvous algorithm.[11] Figure 8 shows how to use the directory. Processors register their owned objects' IDs along with their processor number in a directory (by calling `Zoltan_DD_Update`). This directory is distributed evenly across processors predictably (through either a linear decomposition of the IDs or a hashing of IDs to processors). Then, other processors can obtain a given object's processor number by sending a request for the information to the processor holding the directory entry (by calling `Zoltan_DD_Find`).
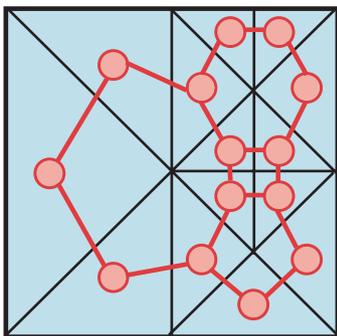
Figure 6. A representation of a triangular finite-element mesh as a graph (red) for graph partitioning. Graph nodes represent mesh elements; graph edges represent shared element faces.
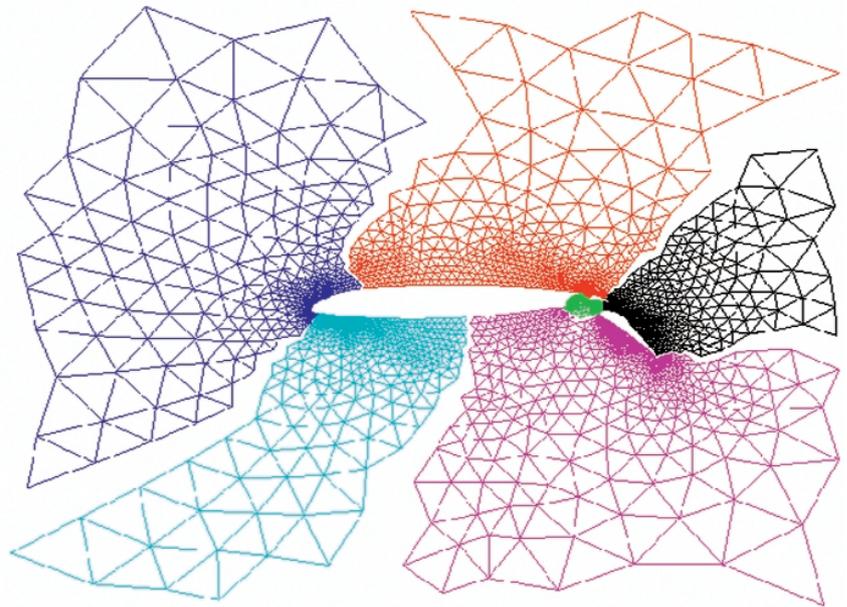
Figure 7. A multilevel graph-based partition of a finite-element mesh for six processors. Colors represent processor assignments of finite-element nodes. (Mesh courtesy of Steve Hammond, NCAR.)
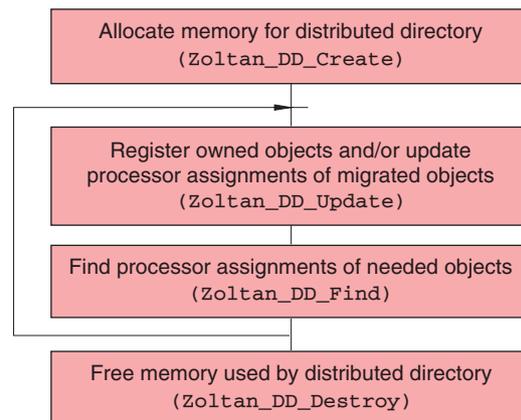
Figure 8. An outline of distributed data directory usage. Once created, directories can be updated and used throughout an application to reflect, say, changing processor assignments due to dynamic load balancing.

Unstructured communication library

Unlike static applications where communication patterns remain fixed throughout the computation, dynamic applications can have complicated, changing communication patterns. For example, after adaptive mesh refinement, new communication patterns must reflect dependencies between newly created elements. Multiphysics simulations, such as crash simulations, might require complicated communication patterns to transfer data between decompositions for different simulation phases.

Zoltan provides an unstructured communication package to simplify communication. It generates a communication plan based on the number of objects to be sent and their destination processors. This plan includes information about both the sends and receives for a given processor. The plan can be used and reused throughout the application or destroyed and rebuilt when communication patterns change. It can also be used in reverse to return

data to requesting processors. The package includes simple communication primitives that insulate the user from details of sends and receives.

Figure 9 shows how to use the communication package to transfer data between two different meshes in a loosely coupled physics simulation. This crash simulation uses a static graph-based decomposition generated by Chaco (left) for the finite-element analysis and a dynamic RCB decomposition (right) for contact detection. Through a call to `Zoltan_Comm_Create`, the application obtains a communication plan (built by Zoltan) that describes data movement between the two decompositions. Using the plan, the application can transfer data between the graph-based and RCB decompositions through calls to `Zoltan_Comm_Do` and `Zoltan_Comm_Do_Reverse`.

Dynamic memory management package

Dynamic applications rely heavily on the ability to allocate and free memory as needed. After repartitioning, for example, new memory is needed for im-

ported objects, and exported objects' memory is freed. Memory leaks and invalid memory accesses are common in developing software. Although many software development tools let users track memory bugs, these tools are often not available on state-of-the-art parallel-computing platforms. Thus, simple in-application debugging tools can be beneficial.

Zoltan's memory management package includes wrappers around malloc and free that provide enhanced debugging capability for memory management. The arguments to these wrappers are identical to those passed to malloc and free, so they are easy to use in applications. For each malloc, the memory management package records the line number and function name of where the allocation took place. When a location is freed, its allocation record is removed. Thus, tracking memory leaks can be simplified by asking the memory management package to print its list of unfreed memory along with the source-code location from which it was allocated. Statistics about memory allocations and frees are also available.

Although we designed Zoltan to be a general-purpose tool for applications, it has proven valuable to us as a research testbed for new software development. By allowing easy comparison of partitioning algorithms, Zoltan is ideal for implementing and testing new load-balancing strategies such as multiconstraint geometric partitioning strategies. Its flexible design lets us easily add new technologies (such as graph-coloring and graph-matching algorithms) to Zoltan. We are building a layer within Zoltan to provide partitioning and data services for heterogeneous computing architectures. **C**\
SE

References

1. M. Berger and S. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Trans. Computers*, vol. C-36, 1989, pp. 279–301.
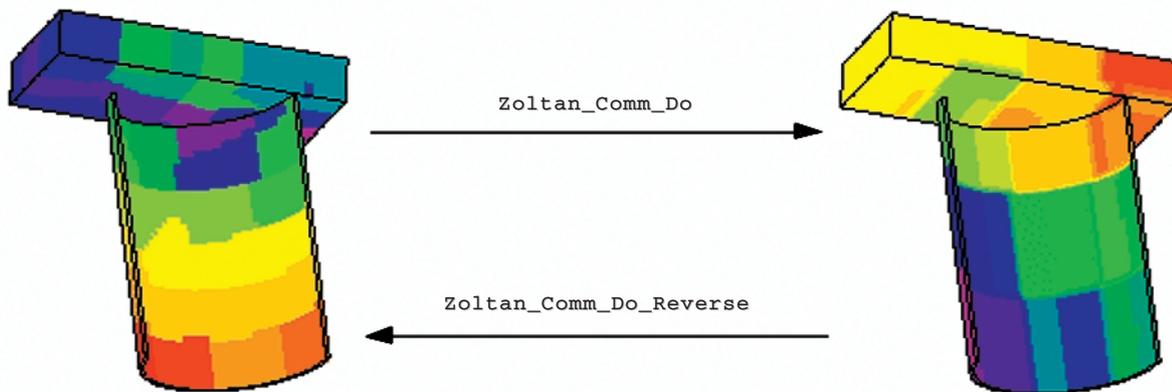
Figure 9. A demonstration of Zoltan's unstructured communication package for loosely coupled physics. In this example, communication primitives simplify data mapping between two different decompositions. The left side shows graph-based decomposition; the right, RCB decomposition. (Image courtesy of Steve Attaway and his colleagues at Sandia National Laboratories.)

2. H. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Eng.,* vol. 2, nos. 2–3, 1991, pp. 135–148.

3. V. Taylor and B. Nour-Omid, "A Study of the Factorization Fill-in for a Parallel Implementation of the Finite Element Method," *Int'l J. Numerical Methods Eng.,* vol. 37, 1994, pp. 3809–3823.

4. R. Loy, *Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws,* doctoral dissertation, Dept. of Computer Science, Rensselaer Polytechnic Inst., 1998.

5. W. Mitchell, "A Comparison of Three Fast Repartition Methods for Adaptive Grids," *Proc. 9th SIAM Conf. Parallel Processing for Scientific Computing,* SIAM, Philadelphia, 1999.

6. G. Karypis, K. Schloegel, and V. Kumar, *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library,* tech. report 97-060, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, 1997.

7. C. Walshaw, M. Cross, and M. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing,* vol. 47, no. 2, 15 Dec. 1997, pp. 102–108.

8. B. Hendrickson and R. Leland, *The Chaco User's Guide, version 2.0,* tech. report SAND 94-2692, Sandia Nat'l Laboratories, Albuquerque, N.M., 1994.

9. B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Proc. Supercomputing '95,* ACM Press, New York, 1995.

10. G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *J. Parallel and Distributed Computing,* vol. 7, no. 2, 1989, pp. 279–301.

11. A. Pinar, *Combinatorial Algorithms in Scientific Computing,* doctoral dissertation, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, 2001.

Karen Devine is the principal investigator for the Zoltan project. Her interests include parallel partitioning, adaptive numerical methods, and software development. She received her PhD in computer science from Rensselaer Polytechnic Institute. Contact her at the Computation, Computers, and Mathematics Center, Sandia Nat'l Labs, Albuquerque, NM 87185-1111; kddevin@sandia.gov.

Erik Boman has a PhD in scientific computing and computational mathematics from Stanford University. His technical interests include scientific computing, numerical linear algebra, and combinatorial algorithms. Contact him at the Computation, Computers, and Mathematics Center, Sandia Nat'l Labs, Albuquerque, NM 87185-1111; egboman@sandia.gov.

Robert Heaphy has a PhD in physics from the University of New Mexico. His research interests include the development of adaptive real-time control systems and software development for massively parallel systems. Contact him at the Computation, Computers, and Mathematics Center, Sandia Nat'l Labs, Albuquerque, NM 87185-1111; rheaphy@sandia.gov.

Bruce Hendrickson received a PhD in computer science from Cornell University. His research interests include algorithms and software tools for high-performance scientific computing. Contact him at the Computation, Computers, and Mathematics Center, Sandia Nat'l Labs, Albuquerque, NM 87185-1111; bahendr@sandia.gov.

Courtenay Vaughan received his PhD in applied mathematics from the University of Virginia. His research focus is parallel computing. Contact him at the Computation, Computers, and Mathematics Center, Sandia Nat'l Labs, Albuquerque, NM 87185-1111; ctvaugh@sandia.gov.