Submission to the International Conference on Supercomputing
Santa Fe, NM, May, 2000

Design of Dynamic Load-Balancing Tools for Parallel Applications[1]

Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St.John and Courtenay Vaughan
Parallel Computing Sciences Department
Sandia National Laboratories
Albuquerque, NM 87185-1111

ABSTRACT: The design of general-purpose dynamic load-balancing tools for parallel applications is more challenging than the design of static partitioning tools. Both algorithmic and software engineering issues arise. We have addressed many of these issues in the design of the Zoltan dynamic load-balancing library. Zoltan has an object-oriented interface that makes it easy to use and provides separation between the application and the load-balancing algorithms. It contains a suite of dynamic load-balancing algorithms, including both geometric and graph-based algorithms. Its design makes it valuable both as a partitioning tool for a variety of applications and as a research test-bed for new algorithmic development. In this paper, we describe Zoltan's design and demonstrate its use in an unstructured-mesh finite element application.

1. INTRODUCTION

In parallel simulations, an important first step is the division of the problem to be solved among the processors. The goal is to assign work to processors evenly (so that no processors are idle while others are computing) while keeping inter-processor communication costs low. For applications with simple data structures (such as regular-grid finite difference schemes in simple geometries), this division may be done implicitly by the application; i.e., the application divides the computational domain into equally sized blocks that are assigned to processors. Applications with more complicated data, geometry and communication requirements, such as finite element methods on unstructured meshes, need more sophisticated partitioning tools. Several high-quality static partitioning tools have been developed for such problems, such as Sandia's Chaco [9] and METIS from the University of Minnesota [10].

Even more complicated are applications in which the per-processor work load or the geometric locality of objects changes as the computation proceeds so that an initially balanced decomposition with low communication costs is then either unbalanced or has unacceptably high amounts of communication. For example, adaptive finite element methods adjust the

1. This work was partially funded by the U.S. Department of Energy's Mathematical, Information and Computational Sciences Division, and was carried out at Sandia National Laboratories operated for the U.S. Department of Energy under contract no. DE-AC04-94AL85000.

computational mesh (*h*-refinement) or the degree of the approximation (*p*-refinement) to satisfy numerical accuracy requirements; dynamic load balancing is needed to redistribute work as the number of degrees of freedom changes [5, 14, 15, 16, 19]. In particle methods and contact detection algorithms, dynamic redistributions that maintain geometric locality of nearby particles or surfaces throughout simulations can greatly reduce communication costs and improve performance of simulations [13, 17, 22].

Dynamic load balancing poses significant algorithmic challenges over static partitioning [8]. Static partitioning algorithms are usually run as pre-processors to applications; thus, they can be run serially with only moderate concern for computation time and memory usage. Dynamic load balancing algorithms, however, run side-by-side with applications; they must be implemented in parallel and use little memory so that the load-balancing algorithms do not hurt the applications' scalability. Dynamic algorithms must also run quickly, as the time to perform load balancing should not exceed an application's time to execute in an unbalanced state. Moreover, dynamic load balancing algorithms take an existing decomposition as input. To establish a new decomposition, data must be moved among processors. This data migration time can be large relative to the cost of actually computing the new decomposition. Thus, algorithms that attempt to minimize data movement are preferred. We say such algorithms are *incremental*; i.e., small changes in processor work loads produce only small changes in the decomposition.

In the development of dynamic load-balancing tools, several software engineering issues also must be addressed. As pre-processors, static partitioning tools like Chaco [9] and METIS [10] use a file-based interface; geometry or graph information is read from a file, and the resulting partition is written to a file which is then read by an application. The file-based interface prevents any dependencies between the partitioners and an application's data structures. Because dynamic load-balancing tools run side-by-side with applications, however, they must have function-call interfaces. To be general-purpose tools, they must be able to obtain information from applications

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

while remaining *data-structure neutral*; i.e., they must not depend upon any particular application's data structures nor restrict the data structures an application may use. Moreover, dynamic load-balancing tools should provide some support for data migration.

To address the issues described above, we have developed a dynamic load-balancing tool called Zoltan [2, 3]. Zoltan is a suite of dynamic load-balancing algorithms for parallel applications. It is designed to be flexible, extensible, and easy-to-use. In the following sections, we describe the software engineering solutions developed for Zoltan and demonstrate Zoltan's use in an unstructured-mesh finite element application.

2. Load Balancing Tools

In most applications using dynamic load balancing, the load-balancing algorithm is implemented directly in the application, with close coupling between the application's and load-balancing algorithm's data structures. This typical approach has two disadvantages. First, it is unlikely that the application developer has incorporated the best algorithm into his application, but he is unable to compare the algorithm with others without taking time to implement many algorithms in his application. Second, the close coupling of data structures limits the algorithm's use to a single application. Developers wanting to use the algorithm in new applications have to re-write the algorithm using the new applications' data structures. Research into and use of dynamic load-balancing algorithms are severely impaired.

To overcome these drawbacks, we have designed an object-oriented, callback-function interface to Zoltan that makes it data-structure neutral and allows it to be used easily by many different applications. The application developer must provide simple functions that return information such as the number of objects (elements, particles, etc.) on a processor, the coordinates and/or connectivity of the objects, and the computational load of the objects. Zoltan then calls these callback functions to get data needed to build the load-balancing data structures.

A typical interaction between an application and the dynamic load-balancing tools is shown in Figure 1. For this example, the nodes of a finite element mesh are the objects to be balanced by Zoltan. Through a call to **LB_Create**, memory is allocated to hold pointers to registered functions, an MPI communicator, and load-balancing data. A pointer to this memory is passed to all other load-balancing functions. The application then selects a load-balancing method to be used (Recursive Coordination Bisection [1], "RCB," in the example) through a call to **LB_Set_Method**. Several callback functions needed by the RCB algorithm are registered through calls to **LB_Set_Fn**. These callback functions include application-defined functions to return the number of objects on the processor (*return_num_nodes*), a list of the objects (*return_node_list*), and the coordinates for a given object (*return_coords*). After some computation, the application calls **LB_Balance** to compute a new decomposition.

Within **LB_Balance**, Zoltan follows pointers to the registered callback functions to build the data structures needed for the RCB algorithm. An array of data is built, with one entry for each object owned by the processor. The number of objects is determined by following the *Get_Num_Obj* pointer to the *return_num_nodes*. Storage is allocated for the objects, and lists of the objects' identifiers (IDs) are obtained by following the *Get_Obj_List* function pointer to the *return_node_list*. Then, for each object, the object's coordinates are obtained through calls through the *Get_Geom* function pointer to the registered function *return_coords*. Once the data structures are built, the load-balancing library can perform the RCB decomposition and return arrays of information describing the new decomposition to the application.

This callback function design has a number of advantages. Most applications use the information needed by the callback functions themselves, so the callback functions are generally very easy to implement. Certainly it is easier for an application developer to write the callback functions than to build specific data structures (graphs, octrees, etc.) required by a particular load-balancing algorithm. Changes in a load-balancing algorithm's data structures do not propagate

Application

```
...
/* Register method and application functions */
lb = LB_Create();
LB_Set_Method(lb, "RCB");
LB_Set_Fn(lb, LB_GEOM_FN, return_coords, data);
LB_Set_Fn(lb, LB_NUM_OBJ_FN, return_num_nodes, data);
LB_Set_Fn(lb, LB_OBJ_LIST_FN, return_node_list, data);
...
/* Call the load balancer */
LB_Balance(lb,&new,&num_imp,&imp_glob_ids,&imp_loc_ids,&imp_procs,
           &num_exp,&exp_glob_ids,&exp_loc_ids,&exp_procs);
...
```

Zoltan

```
...
/* call registered functions to build RCB data structures */
num_objs = lb->Get_Num_Obj();
allocate memory for object global and local IDs
lb->Get_Objs(global_ids, local_ids);
for (i = 0; i < num_objs; i++) {
  lb->Data[i].Global_Tag = global_ids[i];
  lb->Data[i].Local_Tag = local_ids[i];
  lb->Get_Geom(global_ids[i],local_ids[i],lb->Data[i].Coords);
};
/* perform balancing on lb->Data */
...
```

Figure 1. Example demonstrating the interaction between an application and Zoltan.

back to applications. Moreover, once the functions are implemented, applications have access to all algorithms in Zoltan; no changes are needed to use new technology in Zoltan.

The callback function interface does add some time and memory overhead to the load-balancing algorithms. A copy of data needed for load balancing is created; however, since only geometric and/or connectivity information (and not solution information) is needed by Zoltan, this copy is, in general, much smaller than the application's data set. Applications often have free memory available as temporary work space; Zoltan can use this memory and free it upon completion of load balancing without limiting the scalability of an application. The callback function protocol also adds some execution time for building the data structures needed by Zoltan.

In experiments, however, this overhead is only a small fraction of the time needed to compute a new partition.

Zoltan's interface supports both geometric and graph-based algorithms. Geometric algorithms require callback functions that return object IDs and coordinates for the objects; object weights for weighted partitioning are optional. Graph-based algorithms require callback functions that return object IDs and object edge lists describing the connectivity of objects in the application (e.g., the connectivity of the nodes in a finite element mesh); object and edge weights are optional. Zoltan currently includes the Recursive Coordinate Bisection (RCB) [1] and Octree Partitioning [4, 7] geometric algorithms. Graph-based algorithms are provided through interfaces to the ParMETIS [11] and Jostle [21] packages. New algorithms can be added easily by using the callback functions to build the data structures needed by the new algorithms. Recursive inertial bisection [20], space-filling curves [6, 22], and additional tree-based partitioning algorithms [12] are currently being added to Zoltan.

3. Data Migration Tools

Data migration is an extremely application-dependent part of establishing new decompositions. It involves gathering objects from the data structures on one processor, sending those objects to a new processor, inserting the objects into the new processor's data structures, and removing the objects from the original processor. In addition, auxiliary data may have to be sent to the new processor to support the objects migrated there. For example, in a finite element application, the "objects" used in load balancing may be finite-element nodes. But when nodes are migrated to new processors, the elements associated with those nodes must also be sent to the new processors, increasing the dependence of data migration on the application.

A general-purpose load-balancing library like Zoltan can not perform all the operations required for data migration in all applications. However, it can assist an application with the communication required for data migration. Using the results from the load-balancing algorithm,

Zoltan knows where data must be sent to establish the new decomposition and can perform all needed communication using communication tools within the library. The application, then, must specify how to gather data associated with migrating objects and how to insert that data into the new processor's data structures.

Following the registered callback-function design of its dynamic load-balancing tools, Zoltan provides applications with migration-help tools that perform the communication necessary for data migration. An example of the interaction between an application and the migration-help tools is shown in Figure 2. As in Figure 1, the application in this example is a finite element application. The application registers three additional callback functions: a function that returns the size (in bytes) of the data buffer needed to gather all of one object's data (*node_size*), a function that packs one object's data into a buffer (*pack_one_node*), and a function that unpacks one object's data and inserts it into the new processor's data structure (*unpack_one_node*). The application then calls **LB_Help_Migrate** to perform data migration.

The **LB_Help_Migrate** function uses the registered functions with the results of the load-balancing algorithm to move data between processors. The migration-help tools follow the *Get_Obj_Data_Size* function pointer to *node_size* to obtain the size of the data buffer needed for an object's data. They allocate appropriately sized import and export buffers. Through repeated calls to the registered *Pack_Object* function (*pack_one_node*), the migration-help tools fill a communication buffer with data for each object to be exported. The migration-help tools then send the export data to other processors and receive import data. For each object imported, the migration help tools call the registered *Unpack_Object* function (*unpack_one_node*) to unpack the data from the import buffer and insert it into the processor's data structure. Under this model, the application developer does not have to implement additional communication routines to perform data migration; the migration-help tools handle all communication required for data movement.

Zoltan's migration-help tools are separate modules from its dynamic load-balancing tools. Thus, an application does not have to use the migration-help tools even though it uses the dynamic load-balancing tools to compute a new decomposition. If the application has its own migration routines, it can use them in conjunction with Zoltan's load-balancing routines.

Application

```
...
/* Register packing and unpacking functions */
LB_Set_Fn(lb, LB_OBJECT_SIZE_FN, node_size, data);
LB_Set_Fn(lb, LB_PACK_OBJ_FN, pack_one_node, data);
LB_Set_Fn(lb, LB_UNPACK_OBJ_FN, unpack_one_node. data);
LB_Help_Migrate(lb,&num_imp,&imp_glob_ids,&imp_loc_ids,&imp_procs,
                &num_exp,&exp_glob_ids,&exp_loc_ids,&exp_procs);
...
```

Zoltan's Migration-Help Tools

```
...
size = lb->Get_Obj_Data_Size();
/* pack all objects for export  */
for each object i being exported
     lb->Pack_Object(exp_global_id[i],exp_local_id[i],
                     exp_procs[i], size, export_buf[i]);

/* perform communication using map  */
communicate(lb->Comm_Map, export_buf, &import_buf);

/* unpack all imported objects  */
for each object i received
     lb->Unpack_Object(imp_global_id[i], size, import_buf[i]);
...
```

Figure 2. Example demonstrating the interaction between an application and Zoltan's migration-help tools.

4. Summary of Zoltan's Interface

In this section, we describe each of Zoltan's interface and callback functions. We have attempted to keep the number of functions small so that Zoltan will be easy to use in applications. In Table 1, Zoltan's interface functions are summarized. They may be called by an application to perform operations within Zoltan.

Interface Function	Description
LB_Initialize	Initializes MPI if the application has not already done so.
LB_Create	Allocates memory for an **LB_Struct** data structure which stores pointers to registered callback functions, an MPI communicator to be used in the load-balancing algorithms and data structures used by load-balancing algorithms. The address of the **LB_Struct** is returned to the application and passed to all other Zoltan interface functions.
LB_Destroy	Frees memory associated with an **LB_Struct** data structure.
LB_Set_Fn	Registers application-defined callback functions and stores pointers to them in **LB_Struct**.
LB_Set_Method	Indicates which load-balancing algorithm should be used.
LB_Set_Param	Sets parameters to be used in Zoltan; example parameters include tolerance for imbalance, the dimension of object or edge weights to be used, a debugging level, and algorithm-specific parameters.
LB_Balance	Calls the load-balancing algorithm in Zoltan; lists of objects to be exported and imported to establish the new decomposition are returned.
LB_Free_Data	Frees the lists of export and import data returned by **LB_Balance**.
LB_Eval	Computes imbalance and edge-cut cost in an existing decomposition.
LB_Compute_Destinations	Given lists of objects to import on each processor, returns lists of objects to be exported on each processor; useful in multi-phase data migration.
LB_Help_Migrate	Performs data migration using the **LB_PACK_OBJ_FN** and **LB_UNPACK_OBJ_FN** callback functions.

Table 1: Summary of Zoltan interface functions.

In Table 2, we describe the callback functions through which applications provide data to Zoltan. An application does not have to provide all callback functions; it may provide only those needed by the particular load-balancing algorithms used. Objects to be load-balanced in the application must have unique global identifiers (IDs). In addition, they may have local IDs such as memory addresses or array indices within local processor memory. The data types for both global and local IDs can be defined by the application. Zoltan stores both global and local IDs for each object. Zoltan uses only the global IDs, but it passes local IDs to the callback functions to simplify data access for the application.

Callback Function	Description
LB_NUM_OBJ_FN	Returns the number of objects owned by a processor. (required)
LB_OBJ_LIST_FN	Returns lists of global and local IDs for objects owned by a processor. (an LB_OBJ_LIST_FN or an LB_FIRST_OBJ_FN/LB_NEXT_OBJ_FN pair is required)
LB_FIRST_OBJ_FN / LB_NEXT_OBJ_FN	Returns the global and local IDs of the first/next object owned by a processor; used as an iterator over all objects owned by a processor. (an LB_OBJ_LIST_FN or an LB_FIRST_OBJ_FN/LB_NEXT_OBJ_FN pair is required)
LB_NUM_BORDER_OBJ_FN	Returns the number of objects owned by a processor that share a border with a given processor. (optional)
LB_BORDER_OBJ_LIST_FN	Returns lists of global and local IDs for owned objects that share a border with a given processor. (optional)
LB_FIRST_BORDER_OBJ_FN / LB_NEXT_BORDER_OBJ_FN	Returns the IDs of the first/next owned object that shares a border with a given processor; used as an iterator over border objects. (optional)
LB_NUM_EDGES_FN	Given the IDs of an object, returns the number of edges from that object in the application's connectivity. (required for graph-based load-balancing methods)
LB_EDGE_LIST_FN	Given the IDs of an object, returns the IDs of all objects connected to the given object by an edge. (required for graph-based load-balancing methods)
LB_NUM_GEOM_FN	Given the IDs of an object, returns the number of coordinates needed to describe the object's location. (required for geometric load-balancing methods)
LB_GEOM_FN	Given the IDs of an object, returns its coordinates (required for geometric load-balancing methods)
LB_PRE_MIGRATE_FN / LB_POST_MIGRATE_FN	Allows Zoltan to do application-specified processing before and after data migration; called by LB_Help_Migrate. (optional)
LB_OBJ_SIZE_FN	Returns the size of an object to be migrated. (required only if using migration-help tools)
LB_PACK_OBJ_FN	Given the IDs of an object to be exported, packs the object's data into a communication buffer provided by Zoltan. (required only if using migration-help tools)
LB_UNPACK_OBJ_FN	Given the global ID of an object to be imported, unpacks the object's data from a communication buffer and inserts it into the application data structures on the new processor. (required only if using migration-help tools)

Table 2: Summary of Zoltan interface functions.

5. EXAMPLE

To examine the overhead cost of Zoltan in a real application, we incorporated Zoltan into MPSalsa, an unstructured-mesh finite element code for simulating chemically reacting flows [18]. The matrix-fill operation for multi-physics simulations in MPSalsa can be highly unbalanced, and the computational costs of each regime cannot necessarily be predicted in advance. For example, in the catalytic partial-oxidation reactor shown in Figure 3, three different regions are modeled. In the reactor wall (shown as the shaded region of Figure 3), only heat transfer is modeled; computation time per node is approximately 0.006 seconds. In the volume of the reactor (the cross-hatched region), the "whole enchilada" -- heat transfer, mass transfer, fluid flow, and gas-phase reactions with 22 species -- is modeled; computation time per node is approximately 0.05 seconds. On the surface where the catalyst is located (the dark line), the "whole enchilada" and surface reactions are modeled. The surface reactions are very expensive to compute, as a small non-linear solve is performed at each node to compute site deposition fractions. The resulting computation time per surface node is approximately 9.0 seconds. Even though the reacting surface contains less than 2% of the nodes in the entire mesh, the disparate computation times lead to great load imbalance.

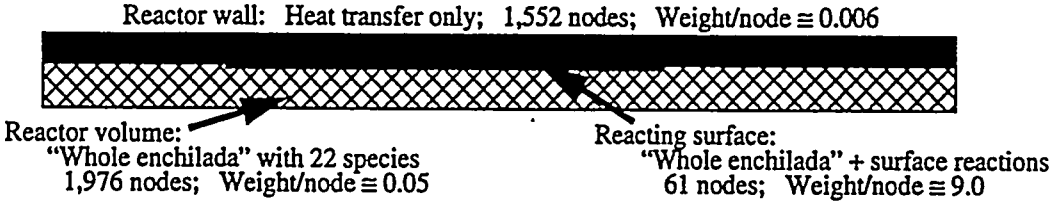

Figure 3. Catalytic partial-oxidation reactor configuration and computational weight used by Zoltan in an unstructured-mesh finite-element multiphysics simulation.

The implementation of the callback functions for load balancing required fewer than 200 lines of C code in MPSalsa, demonstrating that Zoltan's interface is clean and easy to use. Manipulation of data structures for data migration required many more lines of code, as different types of entities (nodes, elements and faces) had to be moved. However, no additional

communication routines were written in MPSalsa to perform data migration; Zoltan performed all communication for data migration. A separate migration phase was used for each type of entity. In each phase, appropriate packing and unpacking callback functions were registered with Zoltan, and **LB_Help_Migrate** was called.

The performance of Zoltan in MPSalsa is summarized in Table 3. Using 50 processors of the Sandia/Intel Tflops (ASCI Red) computer, we ran MPSalsa using an initial decomposition generated by the RCB algorithm without weights on the finite element nodes. We then performed MPSalsa's matrix-fill operation, which required 82.22 seconds. During the matrix-fill operation, nodal computation times were measured and stored. These nodal computation times were used as weights in Zoltan to compute a new RCB decomposition. Using this new decomposition, the time for subsequent matrix-fill operations was reduced 53% for this problem.

The cost of load balancing and data migration in this experiment was less than 2% of the original matrix fill time. The time to compute the new decomposition was 0.28 seconds. Of this time, the actual RCB computation required 0.17 seconds; Zoltan added 0.11 seconds of overhead for executing the callback functions and building the arguments returned by Zoltan. Data migration required 1.14 seconds. Of this time, Zoltan used 0.17 seconds to actually move data among the processors; the remaining time was required by MPSalsa to rebuild MPSalsa data structures such as a solution vector and Jacobian matrix.

6. Conclusion and Future Work

We have described many of the problems involved in designing general-purpose dynamic load-balancing tools and demonstrated solutions in the implementation of the Zoltan library. Zoltan's object-oriented interface provides separation between the data structures of applications and load-balancing algorithms, enabling the library to be used by a wide variety of applications. Zoltan contains a suite of load-balancing methods, including both geometric and graph-based methods. This tool-kit approach has advantages for both application developers and researchers.

Total Load-Balancing Time (Weighted RCB, 50 processors of ASCI Red)		0.28 seconds
Callbacks to build RCB data structure	0.00007 seconds	
RCB Algorithm	0.17 seconds	
Build Return Arguments	0.11 seconds	
Total Data Migration Time		1.14 seconds
Migration of nodes, elements, and faces.	0.17 seconds	
Rebuilding solution vector, matrix, etc.	0.97 seconds	
Matrix Fill Time		
Before Load Balancing		82.22 seconds
After Load Balancing	(53% reduction)	38.10 seconds

Table 3: Performance of Zoltan in an unstructured-mesh finite-element multiphysics simulation.

By providing a number of different types of algorithms, Zoltan allows application developers to experiment easily to find the best methods for their applications. For researchers, Zoltan enables easy incorporation of new algorithms and provides good implementations of standard algorithms to which fair comparisons may be made.

In static partitioning, comparisons of algorithms are performed based on their execution times and quality of partition (measured by the partition's load distribution and communication costs). Comparisons of dynamic load-balancing algorithms, however, must also account for the data migration costs of a new partition. Using Zoltan, we will investigate the relative importance of partition quality, time to compute new partitions, and data migration costs in a number of applications. For some applications, lower-quality partitions may be acceptable if data migration costs can be minimized. Other applications may perform better with high-quality partitions, regardless of the cost of obtaining them. Zoltan's design enables such comparisons. Since it includes a suite of methods, many different methods can be tried in a single application. And because it is easy to incorporate Zoltan in new applications, load-balancing algorithms' performance can be analyzed in different types of simulations.

Heterogeneous computing architectures create new challenges in dynamic load balancing. Partitioning algorithms must account for widely varying processor powers, memory capacities,

and network connections to correctly balance load, prevent memory overflows, and reduce communication over slow-speed network links. Even the use of static partitioners as pre-processors is ineffective in such systems. Instead, a simulation must obtain a set of processors, determine the characteristics of the processors and network, partition based on those characteristics and then execute. Tools and algorithms will be needed to provide this capability.

To address this need, we have completed the design of a heterogeneous computing model for Zoltan. This work is a collaboration with Karypis and Kumar (Univ. of Minnesota). The model represents a heterogeneous computing system as a hierarchy of components. The top level of the hierarchy represents the topology and network speeds of the entire system. The intermediate levels represent components of the computer, including the topology and network speeds of the components. The lowest level of the hierarchy represents the individual processors with their computing power and memory size. Using this model, we will partition both the machine and the application data in an attempt to create partitions having equal execution times and communicating mostly across fast network links. Karypis and Kumar will incorporate the model into the ParMETIS library [11] which provides graph-partitioning capabilities in Zoltan. We will incorporate the model into other existing algorithms in Zoltan and will use the model in the design of new algorithms specifically for heterogeneous architectures.

7. ACKNOWLEDGEMENTS

The authors thank Andrew Salinger, John Shadid, and Steve Plimpton at Sandia National Laboratories for their help with MPSalsa and the implementation of RCB used in Zoltan.

8. REFERENCES

1. M. Berger and S. Bokhari. "A partitioning strategy for nonuniform problems on multiprocessors." *IEEE Trans. Computers*, C-36 (1987) 570-580.
2. E. Boman, K. Devine, B. Hendrickson, M. St.John, and C. Vaughan. "Zoltan: A Dynamic Load-Balancing Library for Parallel Applications, Developer's Guide." Tech. Rep. SAND99-1376, Sandia National Laboratories, Albuquerque, 1999. http://www.cs.sandia.gov/~kddevin/Zoltan_html/dev_html/dev.html

3. E. Boman, K. Devine, B. Hendrickson, M. St.John, and C. Vaughan. "Zoltan: A Dynamic Load-Balancing Library for Parallel Applications, User's Guide." Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, 1999. http://www.cs.sandia.gov/~kddevin/Zoltan_html/ug_html/ug.html
4. H. deCougny, K. Devine, J. Flaherty, R. Loy, C. Ozturan and M. Shephard. "Load balancing for the parallel adaptive solution of partial differential equations." *Appl. Numer. Math.*, **16** (1994) 157-182.
5. K. Devine and J. Flaherty. "Parallel adaptive *hp*-refinement techniques for conservation laws." *Appl. Numer. Math.*, **20** (1996) 367-386.
6. C. Edwards. "A parallel infrastructure for scalable adaptive finite element methods and its application to least squares C^∞ collocation." Ph.D. Dissertation, Univ. of Texas-Austin, 1997.
7. J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco and L. Ziantz. "Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws." *J. Parallel Distrib. Comput.*, **47** (1998) 139-152.
8. B. Hendrickson and K. Devine. "Dynamic Load Balancing in Computational Mechanics." *Comp. Meth. Appl. Mech. Engrg.*, to appear.
9. B. Hendrickson and R. Leland. "The Chaco user's guide, version 2.0." Tech. Rep. SAND94-2692, Sandia National Laboratories, Albuquerque, 1994. http://www.cs.sandia.gov/CRF/chac.html
10. G. Karypis and V. Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs." Tech. Rep. CORR 95-035, Dept. of Computer Science, Univ. of Minnesota, 1995. http://www-users.cs.umn.edu/~karypis/metis
11. G. Karypis and V. Kumar. "ParMETIS: Parallel graph partitioning and sparse matrix ordering library." Tech. Rep. 97-060, Dept. of Computer Science, Univ. of Minnesota, 1997. http://www-users.cs.umn.edu/~karypis/metis/parmetis/main.shtml
12. W. Mitchell. "The Refinement-Tree Partition for Parallel Solution of Partial Differential Equations." *NIST Journal of Research*, **103** (1998) 405-414.
13. A. Nakano and T. Campbell. "An adaptive curvilinear-coordinate approach to dynamic load balancing of parallel multiresolution molecular-dynamics." *Parallel Computing*, **23** (1997) 1461-1478.
14. L. Oliker. *PLUM: Parallel load balancing for unstructured adaptive methods*. Ph.D. Dissertation, Univ. of Colorado, 1998.
15. M. Parashar, J. C. Browne, C. Edwards and K. Klimkowski. "A common data management infrastructure for adaptive algorithms for PDE solutions." *Proc. SC97*, 1997.
16. A. Patra and J.T. Oden. "Problem Decomposition for Adaptive *hp* Finite Element Methods." *J. Computing Systems in Engineering*, **6** (1995).
17. S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan and D. Gardner. "Parallel Transient Dynamics Simulations: Algorithms for Contact Detection and Smoothed Particle Hydrodynamics." *J. Parallel Distrib. Comput.*, **50** (1998) 104-122.
18. A. Salinger, K. Devine, G. Hennigan, H. Moffat, S. Hutchinson, and J. Shadid. "MPSalsa: A Finite Element Computer Program for Reacting Flow Problems --Part 2: User's Guide." Tech. Rep. SAND96-2331, Sandia National Laboratories, Albuquerque, 1996. http://www.cs.sandia.gov/CRF/MPSalsa
19. K. Schloegel, G. Karypis and V. Kumar. "Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes." *J. Parallel Distrib. Comput.*, **47** (1997) 109-124.
20. H. Simon. "Partitioning of unstructured problems for parallel processing." *Proc. Conf. on Parallel Methods on Large Scale Structural Analysis and Physics Applications*, 1991.
21. C. Walshaw. "Parallel Jostle Library Interface: Version 1.1.7." Tech. Rep., Univ. of Greenwich, London, 1995. http://www.gre.ac.uk/jostle
22. M. Warren and J. Salmon. "A Parallel Hashed Oct-Tree N-Body Algorithm." *Proc. Supercomputing '93*, 1993.