# Getting Started with Zoltan: a Short Tutorial*

Karen D. Devine[1], Erik G. Boman[1], Lee Ann Riesen[1], Umit V. Catalyurek[2]
Cédric Chevalier[1]

[1] Sandia National Laboratories+, Scalable Algorithms Department
Albuquerque, NM 87185-1318, USA
{kddevin,egboman,lafisk,ccheval}@sandia.gov
[2] Ohio State University, Biomedical Informatics Department
Columbus, OH 43210, USA
umit@bmi.osu.edu

**Abstract.** The Zoltan library is a toolkit of parallel combinatorial algorithms for unstructured and/or adaptive computations. In this paper, we describe the most significant tools in Zoltan: dynamic partitioning, graph coloring and ordering. We also describe how to obtain, build, and use Zoltan in parallel applications.

**Keywords.** Parallel Computing, Partitioning, Load Balancing, Coloring, Ordering.

## 1 Introduction

The Zoltan library [1] is a toolkit of combinatorial algorithms for parallel, unstructured, and/or adaptive scientific applications. Its data-structure neutral design allows Zoltan to be used by a wide range of applications, including adaptive finite element methods, particle simulations, linear solvers and preconditioners, crash and contact detection, and electrical circuit simulations. Zoltan's largest component is a suite of dynamic load-balancing and partitioning algorithms that increase applications' parallel performance by reducing processor idle time. Zoltan also has graph coloring and graph ordering algorithms useful in, e.g., task schedulers, parallel preconditioners and linear solvers. In addition to native implementations of many algorithms, Zoltan interfaces to the graph and hypergraph partitioning libraries PT-Scotch [2], PaToH [3] and ParMETIS [4].

This paper provides a short guide to common uses of Zoltan. It describes how to obtain and build zoltan, use Zoltan in application programs, provide application data to Zoltan, and perform load balancing, coloring and ordering

* This work was supported by the U.S. Department of Energy's Office of Science through the CSCAPES SciDAC Institute; by the U.S. National Science Foundation under Grants #CNS-0643969 and #CNS-0403342; by Ohio Supercomputing Center.
+ Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

using Zoltan. At the end of the paper, we list several resources for obtaining more information about Zoltan.

Throughout the paper, we refer the reader to specific pages of the Zoltan User's Guide [5]. The Zoltan User's Guide is included in the Zoltan distribution in `Zoltan/doc/Zoltan_html/ug_html`; it is also on-line at http://www.cs.sandia.gov/Zoltan/ug_html. In this paper, references to specific pages are given as the corresponding file in these directories.

## 2 Downloading and Building Zoltan

Zoltan is available both as a stand-alone software package, or as a package within the Trilinos [6] framework. You can download Zoltan from the Zoltan website: http://www.cs.sandia.gov/Zoltan; untarring the file produces the Zoltan main directory `Zoltan`. In Trilinos v9 or later, the Zoltan main directory is `Trilinos/packages/zoltan`.

Zoltan requires MPI for interprocessor communication. To build Zoltan, a C compiler is required. C++ and Fortran90 compilers are needed only if users want Zoltan's optional Fortran90 and C++ interfaces. In this paper, we focus on the C interface.

Zoltan must be built in a separate, user-created directory (e.g., `Zoltan/BuildDir`), not in the main Zoltan directory. To build Zoltan using Autotools, users first run (in their build directory) the auto-configuration tool `configure` and then compile using `make`. The configuration tool allows paths to third-party libraries such as ParMETIS, PT-Scotch and PaToH to be specified through arguments to `configure`. These options can be seen with the following command issued in their build directory: `../configure --help`.

Although Zoltan can be built to run without MPI, most Zoltan users prefer a parallel build that runs on multiple processors. These users must link with MPI and may choose to compile with the MPI compilers. To enable the parallel build, users must use the `--enable-mpi` option; they can also require that MPI compilers (e.g., mpicc, mpic++) be used by specifying `--with-mpi-compilers`.

The script in Figure 1 is an example of configuration and build commands using Autotools. It specifies that Zoltan should be built with both the ParMETIS and PT-Scotch interfaces. Paths to both ParMETIS and PT-Scotch are given. The prefix option states where Zoltan should be installed; in this example, Zoltan's include files will be installed in `/home/zoltan/BuildDir/include`, and the libraries will be in `/home/zoltan/BuildDir/lib`. Zoltan is a library, so no executables are installed. Additional examples are in the directory `Zoltan/SampleConfigurationScripts`.

Zoltan also has a manual build system for users who cannot or choose to not use Autotools. Details of this system are in the Zoltan User's Guide: `ug_usage.html`. Zoltan's Fortran90 interface cannot be built with Autotools; the manual build system must be used for the Fortran90 interface.

Users should include file `zoltan.h` in all source files accessing Zoltan. All applications using Zoltan must link with `-lzoltan`, MPI and any third-party libraries specified in the Zoltan build.

## 3 Basic Zoltan Usage

Figure 2 shows the basic use of Zoltan in an application needing dynamic load balancing. The application begins as usual, reading input files and creating its data structures. Then it calls several Zoltan set-up functions. It initializes Zoltan by calling

```
../configure \
--prefix=/home/zoltan/BuildDir \
--enable-mpi --with-mpi-compilers --with-gnumake \
--enable-zoltan \
--with-scotch \
--with-scotch-incdir="/Net/local/proj/all/src/Scotch5" \
--with-scotch-libdir="/Net/local/proj/linux64/lib/Scotch5" \
--with-parmetis \
--with-parmetis-incdir="/Net/local/proj/all/src/ParMETIS3" \
--with-parmetis-libdir="/Net/local/proj/linux64/lib/ParMETIS3"
make everything
make install
```

**Fig. 1.** Example script for configuring and building Zoltan using Autotools.

`Zoltan_Initialize`, which checks that MPI is initialized. It also calls `Zoltan_Create` to allocate memory for Zoltan; a pointer to this memory is returned by `Zoltan_Create` and must be passed to all other Zoltan functions. Next, by calling `Zoltan_Set_Param`, the application selects the partitioning method it wishes to use and sets method-specific parameters. It registers pointers to callback functions through calls to `Zoltan_Set_Fn`. These callback functions provide Zoltan with information about the application data; they are described in Section 4. After the set-up is completed, the application computes a new partition by calling `Zoltan_LB_Partition` and moves the data to its new part assignments by calling `Zoltan_Migrate`. After migration, `Zoltan_LB_Free_Data` frees the arrays returned by `Zoltan_LB_Partition`. The application then proceeds with its computation using the newly balanced partition. Partitioning and computation can occur in many iterations of the application, with part assignments changing to adjust for changes in the computation. After the iterations are completed, the application calls `Zoltan_Destroy` to free the memory allocated in `Zoltan_Create`, and completes its execution by returning the results of the computation.

The basic set-up of Zoltan — initializing, allocating memory, setting parameters, and registering callback functions, and freeing memory — is the same regardless of whether one uses Zoltan for partitioning, ordering, or coloring. Only the operations in the iteration loop would change if ordering or coloring were needed. Syntax for set-up functions is in the Zoltan User's Guide: `ug_interface_init.html`.

## 4 Describing Application Data to Zoltan

Zoltan is designed to support a wide range of applications whose basic data entities can include (but are not limited to) finite elements, particles, matrix rows/columns/nonzeros, circuits, and agents. Rather than limit Zoltan's capabilities to a specific entity, we consider an entity to be merely an object or thing on which Zoltan operates. Each object must have a `GlobalID`: a name that is unique across all processes. Each `GlobalID` is an array of unsigned integers. Examples of single-integer `GlobalIDs` include global element numbers and global matrix row numbers. Applications that don't support global numbering can use, e.g., a two-integer `GlobalID` consisting of the process rank of the

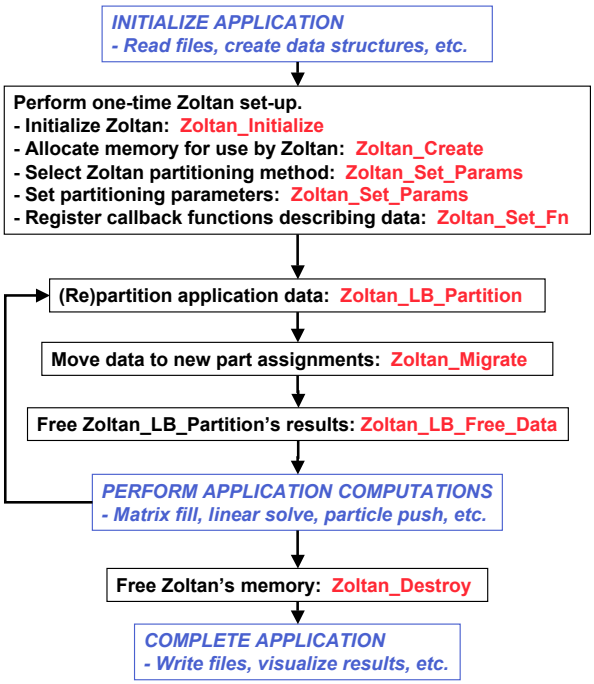


**Fig. 2.** Use of Zoltan in a typical dynamic application. Calls to Zoltan functions are shown in red; application operations are in blue.

process that owns the entity and a local entity counter in the process. Zoltan uses these `GlobalIDs` only to identify objects, so any unique naming scheme is acceptable.

Zoltan also gives users the option of providing a `LocalID` for each object. Each `LocalID` is also an array of unsigned integers. Examples of useful `LocalIDs` include objects' indices in an array of data and pointers to the objects in memory. Using `LocalIDs`, applications can quickly locate objects without having to map from `GlobalIDs` to the local location of the objects. More information on `GlobalIDs` and `LocalIDs` is in the Zoltan User's Guide: `ug_usage.html`.

Just as Zoltan isn't limited to use for specific types of entities, it is also not limited to use with specific application data structures. Instead, the interface to Zoltan completely separates Zoltan's data structures from applications' data structures. This separation is achieved through the use of callback functions — small functions written by the user that access the user's data structures and return needed data to Zoltan. When applications call, say, `Zoltan_LB_Partition`, Zoltan calls these user-provided callback functions to get the application data it needs to do partitioning.

At a minimum, users must write a `ZOLTAN_NUM_OBJ_FN` that returns the number of objects owned by a process, and a `ZOLTAN_OBJ_LIST_FN` that returns the `GlobalIDs` and optional `LocalIDs` for those objects. Other callback functions needed depend on the operations to be performed by Zoltan. Geometric partitioning methods, for example, require a `ZOLTAN_NUM_GEOM_FN` that returns the geometric dimension of the data and a `ZOLTAN_GEOM_MULTI_FN` that returns each object's geometric coordinates. Graph-

based partitioning, coloring and ordering algorithms require the graph-based callbacks `ZOLTAN_NUM_EDGE_MULTI_FN` and `ZOLTAN_EDGE_LIST_MULTI_FN` to return information about connectivity between objects. A list of commonly used functions is provided in Figure 3; full details of callbacks are in the Zoltan User's Guide: `ug_query_lb.html`.

Users register pointers to their callback functions by calling `Zoltan_Set_Fn`. In each `Zoltan_Set_Fn` call, users provide the function pointer and the function type for one callback function. Users may also provide a pointer to one of their data structures to allow that data structure to be accessed in the callback function. Whenever Zoltan calls a callback function, the data pointer provided at registration is passed to the function.

An example of a `Zoltan_Geom_Multi_Fn` callback function for a particle based simulation is included in Figure 4. For each `GlobalID` passed to the function, it returns the coordinates of the particle corresponding to the `GlobalID`. The function uses `LocalIDs` to locate the requested particles in the application data structure. The user's data pointer registered with `user_geom_multi_fn` in `Zoltan_Set_Fn` is provided through the `void *data` pointer. All arrays that callback functions fill (e.g., the `geomVec` array in Figure 4) are allocated by Zoltan.

| **Callback** | **Return values** |
|---|---|
| *All methods* | |
| `ZOLTAN_NUM_OBJ_FN` | Number of objects on processor |
| `ZOLTAN_OBJ_LIST_FN` | List of object IDs and weights |
| *Geometric partitioning* | |
| `ZOLTAN_NUM_GEOM_FN` | Dimensionality of domain |
| `ZOLTAN_GEOM_MULTI_FN` | Coordinates of items |
| *Hypergraph partitioning* | |
| `ZOLTAN_HG_SIZE_CS_FN` | Number of hyperedge pins |
| `ZOLTAN_HG_CS_FN` | List of hyperedge pins |
| `ZOLTAN_HG_SIZE_EDGE_WTS_FN` | Number of hyperedge weights |
| `ZOLTAN_HG_EDGE_WTS_FN` | List of hyperedge weights |
| *Graph/hypergraph partitioning, ordering, coloring* | |
| `ZOLTAN_NUM_EDGE_MULTI_FN` | Number of graph edges |
| `ZOLTAN_EDGE_LIST_MULTI_FN` | List of graph edges and weights |
| *Data migration* | |
| `ZOLTAN_PACK_OBJ_MULTI_FN` | Object data in a communication buffer |
| `ZOLTAN_UNPACK_OBJ_MULTI_FN` | Object data inserted in data structure |

**Fig. 3.** Commonly used Zoltan callback functions and their return values.

## 5 Using Zoltan for Load Balancing and Partitioning

The goal of load balancing and partitioning is dividing data and work among processes in a way that minimizes the overall application execution time. The goal is most often achieved when work is distributed evenly to processes (eliminating process idle time) while at the same time minimizing communication among processes. A *partition* is an assignment of data and work to subsets called *parts* that are mapped to processes. *Static* partitioning is done once at the beginning of an application, with the resulting

```
#include "zoltan.h"
/* Application data type for particle simulation. */
struct Particle {
  int id;
  double x, y, z;
  /* ... solution values, etc. ... */
};

/* Return coordinates for objects requested by Zoltan in globalIDs array. */
void user_geom_multi_fn(void *data, int nge, int nle, int numObj,
                        ZOLTAN_ID_PTR globalIDs, ZOLTAN_ID_PTR localIDs,
                        int dim, double *geomVec, int *err)
{
  /* Cast data pointer provided in Zoltan_Set_Fn to application data type. */
  /* Application data is an array of Particle structures. */
  struct Particle *user_particles = (struct Particle *) data;

  /* Assume for this example that each globalID and localID is one integer. */
  /* Each globalID is a global particle number; each localID is an index    */
  /* into the user's array of Particles.                                    */
  if (nge != 1 || nle != 1) {*err = ZOLTAN_FATAL; return;}

  /* Loop over objects for which coordinates are requested */
  int i, j = 0;
  for (i = 0; i < numObj; i++) {
    /* Copy the coordinates for the object globalID[i] (with localID[i]) */
    /* into the geomVec vector.  Note that Zoltan allocates geomVec.     */
    geomVec[j++] = user_particles[localIDs[i]].x;
    if (dim > 1) geomVec[j++] = user_particles[localIDs[i]].y;
    if (dim > 2) geomVec[j++] = user_particles[localIDs[i]].z;
  }
  *err = ZOLTAN_OK;
}
```

**Fig. 4.** An example of a ZOLTAN_GEOM_MULTI_FN callback for a particle simulation.

partition used throughout the computation. *Dynamic* partitioning is needed in applications whose data locality or work loads vary during the course of the computation. Dynamic partitioning methods have the additional goal of minimizing the cost to move data from the existing partition to the new one.

No single partitioning algorithm is effective for all applications. Applications such as contact detection and particle methods require partitioners that preserve geometric locality of data, while linear solvers and finite element methods benefit from exploiting the structure of interdependencies of their data. Dynamic applications require fast partitioners with low data movement costs, while static applications can tolerate greater partitioning time in exchange for higher quality partitions.

For this reason, we have included a suite of partitioning algorithms in Zoltan. These algorithms fall into three main catagories: geometric, graph-based, and hypergraph-based. Each category requires different callback functions (see Figure 3). Users select a method by setting the Zoltan parameter `LB_METHOD`. They specify whether they are want partitioning (where existing partition is ignored in computing the new one) or repartitioning (where the existing partition is accounted for to reduce data movement costs) with the parameter `LB_APPROACH`. By setting method-specific parameters, users can further customize their choice of partitioning algorithms. High-level Zoltan partitioning parameters are described in the Zoltan User's Guide: `ug_alg.html`; method specific parameters are listed in the User's Guide with each method's description.

All partitioners require the `ZOLTAN_NUM_OBJ_FN` and `ZOLTAN_OBJ_LIST_FN` callbacks to get the objects to be partitioned. Users may also specify weights representing computation costs with each object; if no weights are specified, Zoltan assumes each object has unit weight. Users may also adjust the desired number of parts (parameters `NUM_GLOBAL_PARTS` and/or `NUM_LOCAL_PARTS`), the desired size of each part (function `Zoltan_LB_Set_Part_Sizes`), and the amount of load imbalance acceptable in the new partition (parameter `IMBALANCE_TOL`).

Geometric methods partition data based on their geometric locality. Objects that are physically close to each other are assigned to the same process. Geometric methods in Zoltan include Recursive Coordinate Bisection [7] (RCB), Recursive Inertial Bisection [8,9] (RIB), and Space-Filling Curve partitioning [10,11] (HSFC). Geometric methods are fast and easy to use, making them appropriate for dynamic applications requiring frequent repartitioning. Because they preserve geometric locality of objects, they are ideal for contact detection and particle simulations. However, because they do not explicitly model communication, they can produce partitions with relatively high communication costs.

Graph partitioning is perhaps the most well-known partitioning method. The application is represented as a graph, where data objects are vertices and pairwise data dependencies are edges. The graph partitioning problem is then to partition the vertices into equal-weighted parts, while minimizing the weight of edges with endpoints in different parts. This is an NP-hard optimization problem, but fast multilevel algorithms and software produce good solutions in practice [12,13]. In general, graph partitioning produces better quality partitions than geometric methods, but the partitions take longer to compute. Zoltan includes interfaces to two popular graph partitioning libraries: PT-Scotch [2] and ParMETIS [4]. It also has native graph-partitioning capabilities through its hypergraph partitioner.

Hypergraph partitioning improves the communication model used in graph partitioning. Like the graph model, the hypergraph model represents data objects as vertices. But hypergraph edges represent data dependencies among among sets of objects, not just pairs. Unlike the graph model, the hypergraph model can represent

non-symmetric dependencies between objects. Moreover, it more accurately represents communication volume for edges that cross part boundaries, leading to higher quality partitions. The main drawback of hypergraph methods is that they take longer to run than graph algorithms. Zoltan has a native parallel hypergraph partitioner [14,15] (PHG), as well as an interface to the serial hypergraph partitioner PaToH [3].

After selecting a partitioning method, users call `Zoltan_LB_Partition` to compute the new data distribution. This call only computes a suggested partition; it does not actually migrate data to new parts. `Zoltan_LB_Partition` returns lists of objects to be exported and/or imported to new parts, along with their destinations. These arrays are exactly the input needed by Zoltan's data migration function `Zoltan_Migrate`, described below. Because their size cannot be pre-determined by the user, the returned arrays are allocated by Zoltan; they should be freed later by function `Zoltan_LB_Free_Part`. More details about Zoltan partitioning are in the Zoltan User's Guide: `ug_alg.html` and `ug_interface_lb.html`.

## 6 Using Zoltan for Data Migration

Data migration is, perhaps, the most complicated step in doing dynamic partitioning. After a new partition is computed, application data must be removed from its old part and added to its new part, and interprocessor dependencies between data must be re-established. Because Zoltan does not have information about the application's data structures, it cannot modify those data structures directly. It can, however, help with the communication needed to send objects from their current parts to their new ones. Applications may choose to migrate data on their own, or they can use the `Zoltan_Migrate` function to help with migration. `Zoltan_Migrate` accepts as input the import and/or export lists returned by `Zoltan_LB_Partition`.

As in partitioning, Zoltan uses callback functions to access applications' data for data migration. To use `Zoltan_Migrate`, users must provide a `Zoltan_Pack_Obj_Multi_Fn` that packs data that is being sent to a new part into a communication buffer. Users must also provide a `Zoltan_Unpack_Obj_Multi_Fn` that inserts received data into a part's data structures. The additional callbacks (`Zoltan_Pre_Migrate_PP_Fn`, `Zoltan_Mid_Migrate_PP_Fn`, and `Zoltan_Post_Migrate_PP_Fn`) allow the user to specify operations that should occur before data is packed, between the send and the receive, and after data is unpacked, respectively. More details are in the Zoltan User's Guide: `ug_query_mig.html` and `ug_interface_mig.html`.

## 7 Using Zoltan for Coloring

Zoltan's parallel coloring algorithms are based on the framework described in [**?**]. Coloring is often used for identifying concurrency in parallel computations and efficiently computing sparse Jacobian and Hessian matrices. The problem input is described as a graph, using the graph-based callbacks in Figure 3. Each object or task is a graph vertex, with graph edges describing dependencies between the vertices. In distance-1 coloring, vertices are assigned an integer label such that no two adjacent vertices have the same label. In distance-2 coloring, vertices are labeled such that no two vertices connected by a path of length one or two share the same label.

Users call the function `Zoltan_Color` to compute a coloring. `Zoltan_Color` returns, for each object on a process, the label assigned to the object. Before calling `Zoltan_Color`, users must allocate the arrays that return the objects and labels.

Zoltan parameters control coloring distance (parameter `DISTANCE`), method (parameter `COLORING_METHOD`), and performance characteristics. Zoltan's coloring capability is described in the Zoltan User's Guide: `ug_color.html` and `ug_interface_color.html`.

## 8 Using Zoltan for Ordering

Zoltan provides interfaces for vertex ordering of graphs (sparse matrices). The parallel ordering algorithms in Zoltan are provided through interfaces to the PT-Scotch [2] and ParMETIS [4] libraries; to do ordering, users must specify one of these libraries during Zoltan's build process. Both libraries perform parallel nested-dissection ordering [16], which is typically used to reduce fill in direct solvers and Cholesky factorizations. As in coloring, the problem input is described as a graph, using the graph-based callbacks in Figure 3. Ordering is currently available only for undirected graphs representing sparse symmetric matrices. Here graph vertices represent matrix rows (and columns), while graph edges represent matrix nonzeros.

Users call the Zoltan function `Zoltan_Order` to compute an ordering. `Zoltan_Order` returns the permutation and inverse permutation vectors describing the new ordering. Before calling `Zoltan_Order`, users must allocate the arrays that return the objects and permutations. Additional Zoltan functions return information about the separators used in computing the nested-dissection ordering. More information on Zoltan's ordering interface is in the Zoltan User's Guide: `ug_order.html` and `ug_interface_order.html`.

## 9 Resources for Further Information about Zoltan

Other tools available in Zoltan include distributed data directories and unstructured communication tools. Distributed data directories provide memory efficient, constant-time look-ups of off-processor data. They have been used for updating ghost information and building communication maps in finite element and particle simulations. The unstructured communication tools provide simple interfaces to complicated communication patterns. They are used throughout Zoltan, and are available to applications as well for, say, mapping data between multiple decompositions in multiphase simulations. These utilities are described in the Zoltan User's Guide: `ug_util.html`.

Examples using Zoltan are included with the Zoltan distribution in the directory `Zoltan/examples`. The examples exercise both the C and C++ interfaces to Zoltan. Graph, geometric, and hypergraph callbacks are included for simple input data.

Zoltan news is listed at the Zoltan homepage: `http://www.cs.sandia.gov/Zoltan`. Detailed specification of Zoltan's interface and callbacks functions is in the Zoltan User's Guide: `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`. Questions about Zoltan can be emailed to zoltan-users@software.sandia.gov.

## References

1. Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan data management services for parallel dynamic applications. Computing in Science and Engineering **4** (2002) 90–97
2. Pelligrini, F.: PT-SCOTCH 5.1 user's guide. Research rep., LaBRI (2008)

3. Çatalyürek, U.V., Aykanat, C.: PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/~umit/software.htm. (1999)
4. Karypis, G., Schloegel, K., Kumar, V.: Parmetis: Parallel graph partitioning and sparse matrix ordering library, version 3.1. Technical report, Dept. Computer Science, University of Minnesota (2003) http://www-users.cs.umn.edu/~karypis/metis/parmetis/download.html.
5. Boman, E., Devine, K., Heaphy, R., Hendrickson, B., Leung, V., Riesen, L.A., Vaughan, C., Catalyurek, U., Bozdag, D., Mitchell, W., Teresco, J.: Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide. Sandia National Laboratories, Albuquerque, NM. (2007) Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.
6. Heroux, M., Bartlett, R., Howle, V., Hoekstra, R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, Albuquerque, NM (2003)
7. Berger, M.J., Bokhari, S.H.: A partitioning strategy for nonuniform problems on multiprocessors. IEEE Trans. Computers **C-36** (1987) 570–580
8. Simon, H.D.: Partitioning of unstructured problems for parallel processing. Comp. Sys. Engng. **2** (1991) 135–148
9. Taylor, V.E., Nour-Omid, B.: A study of the factorization fill-in for a parallel implementation of the finite element method. Int. J. Numer. Meth. Engng. **37** (1994) 3809–3823
10. Warren, M.S., Salmon, J.K.: A parallel hashed oct-tree n-body algorithm. In: Proc. Supercomputing '93, Portland, OR (1993)
11. Pilkington, J.R., Baden, S.B.: Partitioning with spacefilling curves. CSE Technical Report CS94–349, Dept. Computer Science and Engineering, University of California, San Diego, CA (1994)
12. Bui, T.N., Jones, C.: A heuristic for reducing fill-in sparse matrix factorization. In: Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM (1993) 445–452
13. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Proc. Supercomputing '95, ACM (1995)
14. Devine, K., Boman, E., Heaphy, R., Bisseling, R., Catalyurek, U.: Parallel hypergraph partitioning for scientific computing. In: Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06), IEEE (2006)
15. Catalyurek, U., Boman, E., Devine, K., Bozdag, D., Heaphy, R., Riesen, L.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: Proc. of 21th International Parallel and Distributed Processing Symposium (IPDPS'07), IEEE (2007)
16. George, A.: Nested dissection of a regular finite-element mesh. SIAM Journal on Numerical Ananlysis **10** (1973) 345–363