Zoltan Tutorial

Erik G. Boman, Karen D. Devine, Lee Ann Riesen

January 31, 2007; revised Jan. 9, 2009

Contents

Chapter 1

Introduction

This document provides an introduction to data partitioning and load balancing using the Zoltan Toolkit (version 3.1). Zoltan is a parallel toolkit for load balancing (and several other combinatorial scientific computing tasks) targeted to the high performance computing community. This tutorial only covers load balancing, and other capabilities in Zoltan are barely mentioned. The Zoltan library is written in C but can be linked with C, C++ and Fortran90 applications. It requires MPI to run in parallel. Most of the examples in this tutorial are C language examples.

In Chapter 2 we discuss the load balancing problem in general and describe common algorithms, which are also supported in Zoltan. Zoltan's role in addressing the load balancing problem is to efficiently partition, or repartition, the problem data across mulitple processes while an application is running, when requested to do so by the application.

In Chapter 3 we briefly explain the interface through which your application employs Zoltan. This topic is convered in more detail in the Zoltan User's Guide available at `http://cs.sandia.gov/Zoltan/ug_html/ug.html`. An example can be worth a thousand words, so you may wish to skip this chapter and look through the examples in the last section first.

The final chapter of this document (4) provides source code examples for simple applications that use Zoltan. These examples may be found in the **examples** directory of the Zoltan source code. Zoltan is open source and freely available.

You will find additional documents and publications at the Zoltan web site at `http://cs.sandia.gov/Zoltan/`. The User's Guide is a very useful reference and provides many details we could not cover here.

Chapter 2

Data partitioning and load balancing

As problem sizes grow, parallel computing has become an important tool in computational science. Many large-scale computing tasks are today run on parallel computers, with multiple processors or cores. An important issue is then how to best divide the data and work among processes (processors). This problem is known as *data partitioning* or *load balancing*. We will use these phrases interchangably. Partitioning or load balancing can either be performed once (static partitioning) or multiple times (dynamic load balancing).

We wish to divide work evenly but at the same time minimize communication among processes. Communication could either be message passing (on distributed memory systems) or memory access (on shared-memory systems).

We give a brief overview of the different categories of partitioning methods, and later explain how to use them in Zoltan.

2.1 Geometric methods

Geometric methods rely on each object having a set of coordinates. Data objects are partitioned based on geometric locality. We assume that it is beneficial to keep objects that are close together on the same processor, but there is no explicit model of communication. Examples of geometric methods include recursive coordinate bisection (RCB) and space-filling curves. An advantage of geometric methods is that they are very fast to compute, but communication volume in the application may be high.

2.2 Graph partitioning

Graph partitioning is perhaps the most popular method. This approach is based on representing the application (computation) as a graph, where data objects are vertices and data dependencies are edges. The graph partitioning problem is then to partition the vertices into equal-sized sets, while minimizing the number of edges with endpoints in different sets (parts). This is an NP-hard optimization problem, but fast multilevel algorithms and software produce good solutions in practice. In general, graph partitioning produces better quality partitions than geometric methods but also take longer to compute.

2.3 Hypergraph partitioning

The graph model has several deficiencies. First, only symmetric relations between pairs of objects can be represented. Second, the communication model is inaccurate and does not model communication volume correctly. Hypergraphs generalize graphs, but can represent relationships among arbitrary sets of objects (not just pairs). Also, communication volume is exact, so hypergraph methods produce very high quality partitions. The main drawback of hypergraph methods is that they take longer to run than graph algorithms.

2.4 Methods in Zoltan

Zoltan is a toolkit containing many load balancing methods. We explain how to use Zoltan in the next chapter. The methods currently available in Zoltan (version 3.1) are:

Simple: BLOCK, RANDOM. These are intended for testing, not real use.

Geometric: RCB, RIB, HSFC.

Graph: Zoltan has a native graph partitioner, and optionlly supports ParMetis and PT-Scotch.

Hypergraph: Zoltan has a native parallel hypergraph partitioner.

Chapter 3

Using the Zoltan library

3.1 Overview

Zoltan supports dynamic load balancing, but it is important to understand what Zoltan is *not*. Zoltan can suggest a better data distribution, but does not transparently move your data around. Zoltan does not know anything about application data structures. A typical use of Zoltan is load balancing in dynamic applications. The application should periodically call Zoltan to get a suggested improved data decomposition, but it's up to the application to decide when to call Zoltan and whether to move (migrate) data or not. Zoltan is a toolkit that contains many different algorithms. No single algorithm is best in all situations, and it is up to the application to decide which algorithm to use. Zoltan makes it easy to try out and compare many different algorithms. Often, just changing a single parameter is sufficient.

Now let's dive into the software issues. Zoltan is a big and complex piece of software. It was designed for production use, not as an educational tool, so it may take a while to get up to speed. The Zoltan library is a C library that you can link with your C, C++ or Fortran application. Details of the C++ and Fortran bindings are provided in the User's Guide (`http://cs.sandia.gov/Zoltan/ug_html/ug.html`). Because Zoltan uses MPI for communication, you must link your application with the Zoltan library and with MPI. (It is actually possible to build Zoltan without MPI, using the bundled serial MPI library, siMPI, but this is rarely useful.)

The following points summarize the interface between your application and the Zoltan library:

- Your parallel application must have a global set of IDs that uniquely identify each object that will be included in the partitioning. Zoltan is flexible about the data type or size of your IDs.
- You make a call to the Zoltan library to create a load balancing instance or handle. All subsequent interactions with Zoltan related to this partitioning problem are done through this handle.
- You set parameters that state which partitioning method you wish Zoltan to use, how you want the method to behave, and what type of data you will be providing.
- You create functions that Zoltan can call during partitioning to obtain your data. You provide the names of these functions to Zoltan.
- When you want to partition or repartition your data, all processes in your parallel application must call Zoltan. When Zoltan returns, each process will have lists of the IDs of the data it must move to another process and of the data it will receive from other processes. You must free these lists when you are done with them.
- The Zoltan functions that you call will return success or failure information.

Most Zoltan users will only use the partitioning functions of Zoltan. But Zoltan provides some other useful capabilities as well, such as functions to aid with data migration, and global data dictionaries to locate the partition holding a data object. After discussing the partitioning interface, we will briefly introduce those capabilities.

3.2 Downloading and building Zoltan

Zoltan is available both as a stand-alone software package, or as a package within the Trilinos framework. The source code is the same, so it makes little difference which one you get. If you currently use Trilinos, you probably have Zoltan already. Zoltan currently supports two different build systems, one manual and one automatic (automake or soon, Cmake). Again, you can choose which system to use. For detailed instructions, see the Zoltan User's Guide. After you build Zoltan, you will have a library **libzoltan.a**, to which you should link your application.

3.3 Initializing and releasing Zoltan

Every process in your parallel application must initialize Zoltan once with a call to **Zoltan_Initialize**. Every partitioning instance that is created with **Zoltan_Create** must be freed at the end with **Zoltan_Destroy**. And all lists returned by Zoltan must be freed with **Zoltan_LB_Free_Part**. These functions are defined in the User's Guide at `http://cs.sandia.gov/Zoltan/ug\_html/ug\_interface\_init.html` and they are are illustrated in each of the examples in chapter 4.

3.4 Application defined query functions

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

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 example, the most basic partitioning algorithms require 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. 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.

The User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_query.html`) provides detailed prototypes for application defined query functions. Several working examples of query functions can be found in this document, for example in the Recursive Coordinate Bisection section (4.2.1);

3.5 Using parameters to configure Zoltan

The behavior of Zoltan is controlled by several parameters and debugging-output levels. For example, you will set the parameter **NUM_GID_ENTRIES** to the size of your application's global IDs, and you will set **DEBUG_LEVEL** to indicate how verbose you want Zoltan to be. These parameters can be set by calls to **Zoltan_Set_Param**. Reasonable default values for all parameters are specified by Zoltan. Many of the parameters are specific to individual algorithms, and are listed in the descriptions of those algorithms in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_param.html`).

Each example in the next chapter (4) includes code that sets general parameters and parameters for specific algorithms.

3.6 Errors returned by Zoltan

All interface functions, with the exception of **Zoltan_Create**, return an error code to the application. The possible return codes are defined in **include/zoltan_types.h** and Fortran module **zoltan**.

They are:

ZOLTAN_OK Function returned without warnings or errors.

ZOLTAN_WARN Function returned with warnings. The application will probably be able to continue to run.

ZOLTAN_FATAL A fatal error occured within the Zoltan library.

ZOLTAN_MEMERR An error occurred while allocating memory. When this error occurs, the library frees any allocated memory and returns control to the application. If the application then wants to try to use another, less memory-intensive algorithm, it can do so.

3.7 Additional functionality provided by Zoltan

The Zoltan library includes many functions, in addition to its partitioning codes, which are designed to aid parallel large-data message passing applications.

While the focus of this tutorial is partitioning, we list these additional capabilities here. Most are described more fully in the User's Guide. Where examples of use exist in the source code, we will point that out.

Data migration Existing applications that are being ported to Zoltan may already contain code to redistribute data across a parallel application.

However new codes may wish to use Zoltan's data migration capabilities. The application must supply functions to pack and unpack the data. Zoltan can either do the data migration automatically after computing the new partitioning, or the application can explicitly call **Zoltan_Migrate** to perform the migration. Details and examples can be found in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_interface\_mig.html`).

Distributed directory utility The owner (i.e. the processor number) of any computational object is subject to change during load balancing. An application may use this directory utility to manage its objects' locations. A distributed directory balances the load (in terms of memory and processing time) and avoids the bottle neck of a centralized directory design. This distributed directory module may be used alone or in conjunction with Zoltan's load balancing capability and memory and communication services. Details of this feature can be found in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_util\_dd.html`).

Timers **Zoltan_Timer_Create** is a useful function that creates a platform independent timer that can be used by a parallel application. The application can specify, for each timer, whether timing checks should involve a global barrier or not. The Zoltan timer is not documented in the User's Guide, but the example **examples/C/zoltanExample1.c** uses it.

Unstructured communication The unstructured communication package provides a simple interface for doing complicated patterns of point-to-point communication, such as those associated with data remapping. This package consists of a few simple functions which create or modify communication plans, perform communication, and destroy communication plans upon completion. The package has proved useful in a variety of different applications. For this reason, it is maintained as a separate library and can be used independently from Zoltan. Details of this feature can be found in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_util\_comm.html`).

Memory allocation wrappers This package consists of wrappers around the standard C memory allocation and deallocation routines which add error-checking and debugging capabilities. These routines are packaged separately from Zoltan to allow their independent use in other applications. They are described more fully in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_util\_mem.html`).

Parallel ordering With the **Zoltan_Order** function, Zoltan provides limited capability for ordering a set of objects, typically given as a graph. This feature is described more fully in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_interface\_order.html`).

Parallel coloring Zoltan provides limited capability for coloring a set of objects, typically given as a graph. In graph coloring, each vertex is assigned an integer label such that no two adjacent vertices have the same label. This feature is described more fully in the User's Guide (`http://cs.sandia.gov/Zoltan/ug\_html/ug\_interface\_color.html`).

Chapter 4

Examples

This chapter begins with a very simple example. We use the Zoltan library in a parallel application to partition a small set of objects, each of which has only a global ID and a weight.

We follow this with three more realistic examples. These examples use the Zoltan library to apply a geometric method, a graph method and a then hypergraph method to a small mesh, graph and hypergraph respectively.

If you have the Zoltan source code, you can find these examples in the **examples/C** directory. The versions found in the source code include code to display the initial partitioning followed by Zoltan's computed partitioning. It may be helpful to compile and run them, and then try changing parameters to see how the partitioning is affected. Consult the Zoltan User's Guide at `http://cs.sandia.gov/Zoltan/ug_html/ug.html` for a complete list of all partitioning methods and all of the parameters for each partitioning method.

The next four examples are written in C. At the end of this section, we also provide a C++ version of the geometric example.

As you read through these examples you will see that, regardless of the load balancing method chosen, every program must:

- initialize the Zoltan library
- supply load balancing parameters to Zoltan
- define query functions that Zoltan will use to obtain information from the application about the objects to be partitioned
- call Zoltan_LB_Partition to begin the load balancing calculation

- free memory allocated by Zoltan when done

4.1 A very simple partitioning method

This example is a C program which uses Zoltan to partition in parallel 36 objects, each of which has a weight. The goal is to balance the weights across the parallel application.

It uses the Zoltan method called BLOCK, which simply assignes the first n/p data objects to the first process and so on. This method really exists as an example for developers to learn how to write partitioning methods. It can also be used for testing in applications, but was never intended to be a serious partitioning strategy for real applications. However, it is a good choice for our first example.

If you are a new user of Zoltan, you may want to initially get your application working with the BLOCK method. Once you have that first step accomplished, you can modify it to use one of the other partitioning methods.

In order to use the BLOCK method, we must define two query functions:

- A function of type ZOLTAN_NUM_OBJ_FN which provides the number of objects belonging to this process. In our example we call this function **get_number_of_objects**.
- A function of type ZOLTAN_OBJ_LIST_FN which supplies the global IDs and optional weights for the objects belonging to the process. We name this function **get_object_list**.

The **ZOLTAN_*** function prototyes named above are defined in the Zoltan file **include/zoltan.h**.

Zoltan requires that the parallel application supply a unique global ID to identify each of the objects to be partitioned. Zoltan is very flexible about the data type used to represent a global ID, and only requires that it be allowed to represent your global ID internally as an array of integers of some user-defined length. Zoltan also permits the use of an additional set of IDs that are local to a process. If you supply these optional local IDs, Zoltan will also include them when querying your application for data.

A version of the code listed below may be found in the Zoltan source in file **examples/C/simpleBLOCK.c**.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include "zoltan.h"

static int numObjects = 36;
static int numProcs, myRank, myNumObj;
static int myGlobalIDs[36];

static float objWeight(int globalID)
{
float w;     /* simulate an initial imbalance */
  if (globalID % numProcs == 0) w = 3;
  else                          w = (globalID % 3 + 1);
  return w;
}
static int get_number_of_objects(void *data, int *ierr)
{
  *ierr = ZOLTAN_OK;
  return myNumObj;
}
static void get_object_list(void *data, int sizeGID, int sizeLID,
            ZOLTAN_ID_PTR globalID, ZOLTAN_ID_PTR localID,
            int wgt_dim, float *obj_wgts, int *ierr)
{
int i;
  *ierr = ZOLTAN_OK;
  for (i=0; i<myNumObj; i++){
    globalID[i] = myGlobalIDs[i];
    if (obj_wgts) obj_wgts[i] = objWeight(myGlobalIDs[i]);
  }
}
int main(int argc, char *argv[])
{
  int rc, i, changes, numGidEntries, numLidEntries, numImport, numExport;
  float ver;
  struct Zoltan_Struct *zz;
  ZOLTAN_ID_PTR importGlobalGids, importLocalGids;
  ZOLTAN_ID_PTR exportGlobalGids, exportLocalGids;
  int *importProcs, *importToPart, *exportProcs, *exportToPart;
```

```
/*****************************************************************
** Initialize MPI and Zoltan
*****************************************************************/

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

rc = Zoltan_Initialize(argc, argv, &ver);

if (rc != ZOLTAN_OK){
  printf("Error initializing Zoltan\n");
  MPI_Finalize();
  exit(0);
}

/*****************************************************************
** Create a simple initial partitioning for this example
*****************************************************************/

for (i=0, myNumObj=0; i<numObjects; i++)
  if (i % numProcs == myRank)
    myGlobalIDs[myNumObj++] = i+1;

/*****************************************************************
** Create a Zoltan library structure for this instance of load
** balancing.  Set the parameters and query functions.
*****************************************************************/

zz = Zoltan_Create(MPI_COMM_WORLD);

/* General parameters */

Zoltan_Set_Param(zz, "LB_METHOD", "BLOCK");  /* name of Zoltan method */
Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1"); /* size of global ID    */
Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "0"); /* no local IDs         */
Zoltan_Set_Param(zz, "OBJ_WEIGHT_DIM", "1"); /* weights are 1 float   */
```

```
/* Query functions */

Zoltan_Set_Num_Obj_Fn(zz, get_number_of_objects, NULL);
Zoltan_Set_Obj_List_Fn(zz, get_object_list, NULL);

/******************************************************************
** Call Zoltan to partition the objects.
******************************************************************/

rc = Zoltan_LB_Partition(zz, /* input (all remaining fields are output) */
      &changes,        /* 1 if partitioning was changed, 0 otherwise */
      &numGidEntries,  /* Number of integers used for a global ID */
      &numLidEntries,  /* Number of integers used for a local ID */
      &numImport,      /* Number of objects to be sent to me */
      &importGlobalGids,  /* Global IDs of objects to be sent to me */
      &importLocalGids,   /* Local IDs of objects to be sent to me */
      &importProcs,    /* Process rank for source of each incoming object */
      &importToPart,   /* New partition for each incoming object */
      &numExport,      /* Number of objects I must send to other processes*/
      &exportGlobalGids,  /* Global IDs of the objects I must send */
      &exportLocalGids,   /* Local IDs of the objects I must send */
      &exportProcs,    /* Process to which I send each of the objects */
      &exportToPart);  /* Partition to which each object will belong */

if (rc != ZOLTAN_OK){
  printf("Error in Zoltan library\n");
  MPI_Finalize();
  Zoltan_Destroy(&zz);
  exit(0);
}

/******************************************************************
** Here you would send the objects to their new partitions.
******************************************************************/

/******************************************************************
** Free the arrays allocated by Zoltan_LB_Partition, and free
** the storage allocated for the Zoltan structure.
```

```
  *****************************************************************/

  Zoltan_LB_Free_Part(&importGlobalGids, &importLocalGids,
                      &importProcs, &importToPart);
  Zoltan_LB_Free_Part(&exportGlobalGids, &exportLocalGids,
                      &exportProcs, &exportToPart);

  Zoltan_Destroy(&zz);

  MPI_Finalize();

  return 0;
}
```

4.2 Geometric, Graph and Hypergraph Examples

The simple graph used in all three examples in this section is illustrated in Figure 4.2. It is defined in the Zoltan source file **examples/C/simpleGraph.h**, which is listed in Figure 4.2.

```
21----22----23----24---25
|     |     |     |    |
16----17----18----19---20
|     |     |     |    |
11----12----13----14---15
|     |     |     |    |
6-----7-----8-----9----10
|     |     |     |    |
1-----2-----3-----4----5
```

Figure 4.1: Mesh with 25 vertices

```
#ifndef SIMPLEGRAPH_H
#define SIMPLEGRAPH_H
static int numvertices=25;
static int simpleNumEdges[25] = {
2, 3, 3, 3, 2,
3, 4, 4, 4, 3,
3, 4, 4, 4, 3,
3, 4, 4, 4, 3,
2, 3, 3, 3, 2
};
static int edges[25][4]={
{2,6},       /* adjacent to vertex 1 */
{1,3,7},     /* adjacent to vertex 2 */
{2,8,4},
{3,9,5},
{4,10},
{1,7,11},
{6,2,8,12},
{7,3,9,13},
{8,4,10,14},
{9,5,15},
{6,12,16},
{11,7,13,17},
{12,8,14,18},
{13,9,15,19},
{14,10,20},
{11,17,21},
{16,12,18,22},
{17,13,19,23},
{18,14,20,24},
{19,15,25},
{16,22},
{21,17,23},
{22,18,24},
{23,19,25},
{24,20}      /* adjacent to vertex 25 */
};
#endif
```

Figure 4.2: Adjacencies for a simple mesh

4.2.1 Recursive Coordinate Bisection

This example uses *Recursive Coordinate Bisection*, one of Zoltan's geometric methods, to partition the vertices of the simple mesh. This method will use the coordinates of each vertex, and the weight of each vertex, in determining a partitioning. The mesh adjacencies are irrelevant in this example, although we define the weight of a vertex to be the number of mesh neighbors it has.

First we will define the query functions required by RCB. The full source code for this example may be found in **examples/C/simpleRCB.c**.

Application defined query functions

We must define four query functions. The function prototyes named below are defined in the Zoltan file **include/zoltan.h**.

- A function of type ZOLTAN_NUM_OBJ_FN (Figure 4.3) which provides the number of objects belonging to this process
- A function of type ZOLTAN_OBJ_LIST_FN (Figure 4.4) which supplies the global IDs and optional weights for the objects belonging to the process
- A function of type ZOLTAN_NUM_GEOM_FN (Figure 4.5) which provides the dimension of the objects
- A function of type ZOLTAN_GEOM_MULTI_FN (Figure 4.6) which supplies the coordinates of the objects and optionally their weights

An alternative to defining a single ZOLTAN_OBJ_LIST_FN is to define a ZOLTAN_FIRST_OBJ_FN and a ZOLTAN_NEXT_OBJ_FN. The first function would return the global ID for first object. Each call to the second function would return a subsequent global ID.

An alternative to ZOLTAN_GEOM_MULTI_FN is to define a ZOLTAN_GEOM_FN which returns the coordinates for a single object rather than for a list of objects.

The variables referred to in the query functions below were defined in the header file 4.2 to describe the mesh in Figure 4.2.

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

  for (i=0; i<simpleNumVertices; i++){
    if (i % numProcs == myRank) numobj++;
  }
  *ierr = ZOLTAN_OK;
  return numobj;
}
```

Figure 4.3: A ZOLTAN_NUM_OBJ_FN query function

```
static void get_object_list(void *data, int sizeGID, int sizeLID,
            ZOLTAN_ID_PTR globalID, ZOLTAN_ID_PTR localID,
                  int wgt_dim, float *obj_wgts, int *ierr)
{
int i, next;

  if ( (sizeGID != 1) || (sizeLID != 1) || (wgt_dim != 1)){
    *ierr = ZOLTAN_FATAL;
    return;
  }

  for (i=0, next=0; i<simpleNumVertices; i++){
    if (i % numProcs == myRank){
      globalID[next] = i+1;   /* application wide global ID */
      localID[next] = next;   /* process specific local ID  */
      obj_wgts[next] = (float)simpleNumEdges[i];  /* weight */
      next++;
    }
  }

  *ierr = ZOLTAN_OK;

  return;
}
```

Figure 4.4: A ZOLTAN_OBJ_LIST_FN query function

```
static int get_num_geometry(void *data, int *ierr)
{
  *ierr = ZOLTAN_OK;
  return 2;
}
```

Figure 4.5: A ZOLTAN_NUM_GEOM_FN query function

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

  if ( (sizeGID != 1) || (sizeLID != 1) || (num_dim != 2)){
    *ierr = ZOLTAN_FATAL;
    return;
  }

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

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

  *ierr = ZOLTAN_OK;
  return;
}
```

Figure 4.6: A ZOLTAN_GEOM_MULTI_FN query function

Calling the Zoltan library

The source code below may be found in the file **examples/C/simpleRCB.c**. More information about RCB parameters may be found in the Zoltan User's Guide at `http://cs.sandia.gov/Zoltan/ug_html/ug_alg_rcb.html`.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "zoltan.h"
#include "simpleGraph.h"
#include "simpleQueries.h"

int main(int argc, char *argv[])
{
  int rc, i, ngids, nextIdx;
  struct Zoltan_Struct *zz;
  int changes, numGidEntries, numLidEntries, numImport, numExport;
  ZOLTAN_ID_PTR importGlobalGids, importLocalGids;
  ZOLTAN_ID_PTR exportGlobalGids, exportLocalGids;
  int *importProcs, *importToPart, *exportProcs, *exportToPart;
  float *wgt_list, ver;
  int *gid_flags, *gid_list, *lid_list;

  /***************************************************************
  ** Initialize MPI and Zoltan
  ***************************************************************/

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

  rc = Zoltan_Initialize(argc, argv, &ver);

  if (rc != ZOLTAN_OK){
    MPI_Finalize();
    exit(0);
  }
```

```
/***************************************************************
** Create a Zoltan library structure for this instance of load
** balancing.  Set the parameters and query functions that will
** govern the library's calculation.  See the Zoltan User's
** Guide for the definition of these and many other parameters.
***************************************************************/

zz = Zoltan_Create(MPI_COMM_WORLD);

/* General parameters */

Zoltan_Set_Param(zz, "DEBUG_LEVEL", "0");
Zoltan_Set_Param(zz, "LB_METHOD", "RCB");
Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");
Zoltan_Set_Param(zz, "OBJ_WEIGHT_DIM", "1");
Zoltan_Set_Param(zz, "RETURN_LISTS", "ALL");

/* RCB parameters */

Zoltan_Set_Param(zz, "KEEP_CUTS", "1");
Zoltan_Set_Param(zz, "RCB_OUTPUT_LEVEL", "0");
Zoltan_Set_Param(zz, "RCB_RECTILINEAR_BLOCKS", "1");

/* Query functions - defined in simpleQueries.h, to return
 * information about objects defined in simpleGraph.h      */

Zoltan_Set_Num_Obj_Fn(zz, get_number_of_objects, NULL);
Zoltan_Set_Obj_List_Fn(zz, get_object_list, NULL);
Zoltan_Set_Num_Geom_Fn(zz, get_num_geometry, NULL);
Zoltan_Set_Geom_Multi_Fn(zz, get_geometry_list, NULL);
```

```
/******************************************************************
** Zoltan can now partition the vertices in the simple mesh.
** In this simple example, we assume the number of partitions is
** equal to the number of processes.  Process rank 0 will own
** partition 0, process rank 1 will own partition 1, and so on.
*****************************************************************/

rc = Zoltan_LB_Partition(zz, /* input (all remaining fields are output) */
      &changes,        /* 1 if partitioning was changed, 0 otherwise */
      &numGidEntries,  /* Number of integers used for a global ID */
      &numLidEntries,  /* Number of integers used for a local ID */
      &numImport,      /* Number of vertices to be sent to me */
      &importGlobalGids,  /* Global IDs of vertices to be sent to me */
      &importLocalGids,   /* Local IDs of vertices to be sent to me */
      &importProcs,    /* Process rank for source of each incoming vertex */
      &importToPart,   /* New partition for each incoming vertex */
      &numExport,      /* Number of vertices I must send to other processes*
      &exportGlobalGids,  /* Global IDs of the vertices I must send */
      &exportLocalGids,   /* Local IDs of the vertices I must send */
      &exportProcs,    /* Process to which I send each of the vertices */
      &exportToPart);  /* Partition to which each vertex will belong */

if (rc != ZOLTAN_OK){
  printf("sorry...\n");
  MPI_Finalize();
  Zoltan_Destroy(&zz);
  exit(0);
}

/******************************************************************
** In a real application, you would rebalance the problem now by
** sending the objects to their new partitions.
*****************************************************************/
```

```
  /******************************************************************
  ** Free the arrays allocated by Zoltan_LB_Partition, and free
  ** the storage allocated for the Zoltan structure.
  ******************************************************************/

  Zoltan_LB_Free_Part(&importGlobalGids, &importLocalGids,
                      &importProcs, &importToPart);
  Zoltan_LB_Free_Part(&exportGlobalGids, &exportLocalGids,
                      &exportProcs, &exportToPart);

  Zoltan_Destroy(&zz);

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

  MPI_Finalize();

  return 0;
}
```

4.2.2 A graph problem

In this section we provide an example of a parallel C program that uses the Zoltan library to partition the simple graph that was pictured in Figure 4.2. You can compile and run this example if you have the Zoltan source code. It is in the examples directory, in file **examples/C/simpleGRAPH.c**.

Application defined query functions

We must define four query functions in order to use the particular graph submethod we chose for this example. See the User's Guide at `http://cs.sandia.gov/Zoltan/ug_html/ug_alg_parmetis.html` for the requirements of other submethods.

The function prototypes referred to below are defined in the Zoltan file **include/zoltan.h**. The four query functions are:

- A function of type ZOLTAN_NUM_OBJ_FN (Figure 4.3) which provides the number of objects belonging to this process

- A function of type ZOLTAN_OBJ_LIST_FN (Figure 4.4) which supplies the global IDs and optional weights for the objects belonging to the process

- A function of type ZOLTAN_NUM_EDGES_MULTI_FN (Figure 4.7) which provides the number of edges for each object

- A function of type ZOLTAN_EDGE_LIST_MULTI_FN (Figure 4.8) which, for a given object ID, responds with the IDs for each adjacent object, the process owning that adjacent object, and optionally a weight for each edge

An alternative to defining a single ZOLTAN_OBJ_LIST_FN is to define a ZOLTAN_FIRST_OBJ_FN and a ZOLTAN_NEXT_OBJ_FN. The first function would return the global ID for first object. Each call to the second function would return a subsequent global ID.

An alternative to defining a ZOLTAN_NUM_EDGES_MULTI_FN is to define a ZOLTAN_NUM_EDGES_FN, which returns the number of adjacent objects for a single object ID.

```
static void get_num_edges_list(void *data, int sizeGID, int sizeLID,
        int num_obj, ZOLTAN_ID_PTR globalID, ZOLTAN_ID_PTR localID,
        int *numEdges, int *ierr)
{
int i, idx;

  if ( (sizeGID != 1) || (sizeLID != 1)){
    *ierr = ZOLTAN_FATAL;
    return;
  }

  for (i=0;  i < num_obj ; i++){
    idx = globalID[i] - 1;
    numEdges[i] = simpleNumEdges[idx];
  }

  *ierr = ZOLTAN_OK;
  return;
}
```

Figure 4.7: A ZOLTAN_NUM_EDGES_MULTI_FN query function

An alternative to defining a ZOLTAN_EDGE_LIST_MULTI_FN is to define a ZOLTAN_EDGE_LIST_FN, which returns the same information but for a single object.

The variables used in these query functions to describe our simple graph refer to those defined in **examples/C/simpleGraph.h** which were shown in Figure 4.2.

```
static void get_edge_list(void *data, int sizeGID, int sizeLID,
        int num_obj, ZOLTAN_ID_PTR globalID, ZOLTAN_ID_PTR localID,
        int *num_edges, ZOLTAN_ID_PTR nborGID, int *nborProc,
        int wgt_dim, float *ewgts, int *ierr)
{
int i, j, idx;
ZOLTAN_ID_PTR nextID;
int *nextProc;

  if ( (sizeGID != 1) || (sizeLID != 1) || (wgt_dim != 0)){
    *ierr = ZOLTAN_FATAL;
    return;
  }

  nextID = nborGID;
  nextProc = nborProc;

  for (i=0;  i < num_obj ; i++){
    idx = globalID[i] - 1;
    if (num_edges[i] != simpleNumEdges[idx]){
      *ierr = ZOLTAN_FATAL;
      return;
    }
    for (j=0; j<num_edges[i]; j++){
      *nextID++ = simpleEdges[idx][j];
      *nextProc++ = (simpleEdges[idx][j] - 1) % numProcs;
    }
  }

  *ierr = ZOLTAN_OK;

  return;
}
```

Figure 4.8: A ZOLTAN_EDGE_LIST_MULTI_FN query function

Calling the Zoltan library

This code may be found in the file **examples/C/simpleGRAPH.c**. More information about graph parameters may be found in the Zoltan User's Guide at `http://cs.sandia.gov/Zoltan/ug_html/ug_alg_parmetis.html`.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "zoltan.h"
#include "simpleGraph.h"
#include "simpleQueries.h"

int main(int argc, char *argv[])
{
  int rc, i, ngids, nextIdx;
  float ver;
  struct Zoltan_Struct *zz;
  int changes, numGidEntries, numLidEntries, numImport, numExport;
  ZOLTAN_ID_PTR importGlobalGids, importLocalGids;
  ZOLTAN_ID_PTR exportGlobalGids, exportLocalGids;
  int *importProcs, *importToPart, *exportProcs, *exportToPart;
  float *wgt_list;
  int *gid_flags, *gid_list, *lid_list;

  /******************************************************************
  ** Initialize MPI and Zoltan
  ******************************************************************/

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

  rc = Zoltan_Initialize(argc, argv, &ver);

  if (rc != ZOLTAN_OK){
    MPI_Finalize();
    exit(0);
  }
```

```
/***************************************************************
** Create a Zoltan library structure for this instance of load
** balancing.  Set the parameters and query functions that will
** govern the library's calculation.  See the Zoltan User's
** Guide for the definition of these and many other parameters.
***************************************************************/

zz = Zoltan_Create(MPI_COMM_WORLD);

/* General parameters */

Zoltan_Set_Param(zz, "DEBUG_LEVEL", "0");
Zoltan_Set_Param(zz, "LB_METHOD", "GRAPH");
Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "1");
Zoltan_Set_Param(zz, "OBJ_WEIGHT_DIM", "1");
Zoltan_Set_Param(zz, "RETURN_LISTS", "ALL");

/* Graph parameters */

Zoltan_Set_Param(zz, "PARMETIS_METHOD", "PARTKWAY");
Zoltan_Set_Param(zz, "PARMETIS_COARSE_ALG", "2");
Zoltan_Set_Param(zz, "CHECK_GRAPH", "2");

/* Query functions - defined in simpleQueries.h */

Zoltan_Set_Num_Obj_Fn(zz, get_number_of_objects, NULL);
Zoltan_Set_Obj_List_Fn(zz, get_object_list, NULL);
Zoltan_Set_Num_Edges_Multi_Fn(zz, get_num_edges_list, NULL);
Zoltan_Set_Edge_List_Multi_Fn(zz, get_edge_list, NULL);
```

```
/****************************************************************
** Zoltan can now partition the simple graph.
** In this simple example, we assume the number of partitions is
** equal to the number of processes.  Process rank 0 will own
** partition 0, process rank 1 will own partition 1, and so on.
****************************************************************/

rc = Zoltan_LB_Partition(zz, /* input (all remaining fields are output) */
      &changes,        /* 1 if partitioning was changed, 0 otherwise */
      &numGidEntries,  /* Number of integers used for a global ID */
      &numLidEntries,  /* Number of integers used for a local ID */
      &numImport,      /* Number of vertices to be sent to me */
      &importGlobalGids,  /* Global IDs of vertices to be sent to me */
      &importLocalGids,   /* Local IDs of vertices to be sent to me */
      &importProcs,    /* Process rank for source of each incoming vertex */
      &importToPart,   /* New partition for each incoming vertex */
      &numExport,      /* Number of vertices I must send to other processes*/
      &exportGlobalGids,  /* Global IDs of the vertices I must send */
      &exportLocalGids,   /* Local IDs of the vertices I must send */
      &exportProcs,    /* Process to which I send each of the vertices */
      &exportToPart);  /* Partition to which each vertex will belong */

if (rc != ZOLTAN_OK){
  printf("sorry...\n");
  MPI_Finalize();
  Zoltan_Destroy(&zz);
  exit(0);
}

/****************************************************************
** In a real application, you would rebalance the problem now by
** sending the objects to their new partitions.  Your query
** functions would need to reflect the new partitioning.
****************************************************************/
```

```
  /*****************************************************************
  ** Free the arrays allocated by Zoltan_LB_Partition, and free
  ** the storage allocated for the Zoltan structure.
  *****************************************************************/

  Zoltan_LB_Free_Part(&importGlobalGids, &importLocalGids,
                      &importProcs, &importToPart);
  Zoltan_LB_Free_Part(&exportGlobalGids, &exportLocalGids,
                      &exportProcs, &exportToPart);

  Zoltan_Destroy(&zz);

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

  MPI_Finalize();

  return 0;
}
```

4.2.3 A hypergraph problem

In this example, we use the simple graph in Figure 4.2 to create a hypergraph, and then partition it using Zoltan's parallel hypergraph partitioner.

In a hypergraph, each edge can be associated with more than two vertices. This edge is called a hyperedge.

For each vertex in the simple graph, we create a hyperedge with a vertex list composed of that vertex and all of its graph neighbors. So our hypergraph has 25 hyperedges, one for each vertex in the simple graph.

We are creating this hypergraph in the hypergraph query functions. But you can use the Zoltan parameter *PHG_FROM_GRAPH_METHOD=neighbors* described in the User's Guide at `http://cs.sandia.gov/Zoltan/ug_html/ug_alg_phg.html`, to accomplish the same result. In this case you use graph query functions instead of hypergraph query funtions to supply the graph to Zoltan. The Zoltan library would then do the conversion to a hypergraph.

First we will define the required query functions.

Application defined query functions

We must define six query functions. The function prototyes named below are defined in the Zoltan file **include/zoltan.h**.

- A function of type ZOLTAN_NUM_OBJ_FN (Figure 4.3) which provides the number of objects (vertices) belonging to this process .
- A function of type ZOLTAN_OBJ_LIST_FN (Figure 4.9) which supplies the object global IDs and optional weights. It differs from the query function created for the last two examples (Figure 4.4) because it omits the optional local IDs and weights.
- A function of type ZOLTAN_HG_SIZE_CS_FN (Figure 4.10). Each process will supply a portion of the hypergraph in a compressed storage format. This function tells the Zoltan library the size of the information that will be returned by the application process.
- A function of type ZOLTAN_HG_CS_FN (Figure 4.11) which supplies a portion of the hypergraph in a compressed storage format.
- A function of type ZOLTAN_HG_SIZE_EDGE_WTS_FN (Figure 4.12) which supplies the number of hyperedges for which the process will provide edge weights.

- A function of type ZOLTAN_HG_EDGE_WTS_FN (Figure 4.13) which supplies optional edge weights for the hyperedges.

An alternative to defining a single ZOLTAN_OBJ_LIST_FN is to define a ZOLTAN_FIRST_OBJ_FN and a ZOLTAN_NEXT_OBJ_FN. The first function would return the global ID for first object. Each call to the second function would return a subsequent global ID.

In this example the application is maintaining a small structure with information about the size of the hypergraph. The application supplies the address of this structure to the Zoltan library when setting up the query functions. Zoltan supplies the address of the structure to the application when calling the query functions.

If you have the Zoltan source, the following query functions are defined in the file **examples/C/simpleQueries.h**. The variables used in these query functions refer to those defined in **examples/C/simpleGraph.h** which were shown in Figure 4.2.

```
static void get_hg_object_list(void *data, int sizeGID, int sizeLID,
            ZOLTAN_ID_PTR globalID, ZOLTAN_ID_PTR localID,
            int wgt_dim, float *obj_wgts, int *ierr)
{
int i, next;

  if ((sizeGID != 1) || (sizeLID != 0) || (wgt_dim != 0)) {
    *ierr = ZOLTAN_FATAL;
    return;
  }

  for (i=0, next=0; i<simpleNumVertices; i++){
    if (i % numProcs == myRank){
      globalID[next] = i+1;   /* application wide global ID */
      next++;
    }
  }

  *ierr = ZOLTAN_OK;

  return;
}
```

Figure 4.9: A ZOLTAN_OBJ_LIST_FN query function

```
static struct _hg_data{
  int numEdges;
  int numPins;
}hg_data;

void get_hg_size(void *data, int *num_lists, int *num_pins,
                 int *format, int *ierr)
{
int i;
struct _hg_data *hgd;

  hgd = (struct _hg_data *)data;

  hgd->numEdges = 0;
  hgd->numPins  = 0;

  for (i=0; i<simpleNumVertices; i++){
    if (i % numProcs == myRank){
      hgd->numPins += simpleNumEdges[i] + 1;  /* my hyperedge */
      hgd->numEdges++;
    }
  }

  *num_lists = hgd->numEdges;
  *num_pins  = hgd->numPins;
  *format    = ZOLTAN_COMPRESSED_EDGE;
  *ierr      = ZOLTAN_OK;

  return;
}
```

Figure 4.10: A ZOLTAN_HG_SIZE_CS_FN query function

```
void get_hg(void *data, int sizeGID, int num_rows, int num_pins,
            int format, ZOLTAN_ID_PTR edge_GID, int *edge_ptr,
            ZOLTAN_ID_PTR pin_GID, int *ierr)
{
int i, j, npins;
struct _hg_data *hgd;

  hgd    = (struct _hg_data *)data;

  if ((num_rows != hgd->numEdges) || (num_pins != hgd->numPins) ||
      (format != ZOLTAN_COMPRESSED_EDGE)){
    *ierr = ZOLTAN_FATAL;
    return;
  }

  npins = 0;

  for (i=0; i<simpleNumVertices; i++){
    if (i % numProcs == myRank){    /* my hyperedge */
      *edge_ptr++ = npins;      /* index into start of pin list */
      *edge_GID++ = i+1;        /* hyperedge global ID */

      /* list global ID of each pin (vertex) in hyperedge */
      for (j=0; j<simpleNumEdges[i]; j++){
        *pin_GID++ = simpleEdges[i][j];
      }
      *pin_GID++ = i+1;
      npins += simpleNumEdges[i] + 1;
    }
  }

  *ierr = ZOLTAN_OK;

  return;
}
```

Figure 4.11: A ZOLTAN_HG_CS_FN query function

```
void get_hg_num_edge_weights(void *data, int *num_edges, int *ierr)
{
struct _hg_data *hgd;

  hgd    = (struct _hg_data *)data;

  *num_edges = hgd->numEdges;
  *ierr = ZOLTAN_OK;

  return;
}
```

Figure 4.12: A ZOLTAN_HG_SIZE_EDGE_WTS_FN query function

```
void get_hyperedge_weights(void *data, int sizeGID,
         int sizeLID, int num_edges, int edge_weight_dim,
         ZOLTAN_ID_PTR edge_GID, ZOLTAN_ID_PTR edge_LID,
         float  *edge_weight, int *ierr)
{
int i;
struct _hg_data *hgd;

  hgd    = (struct _hg_data *)data;

  if ((sizeGID != 1) || (sizeLID != 0) ||
      (num_edges != hgd->numEdges) || (edge_weight_dim != 1)){
    *ierr = ZOLTAN_FATAL;
    return;
  }

  for (i=0; i<simpleNumVertices; i++){
    if (i % numProcs == myRank){
      *edge_GID++ = i+1;        /* hyperedge global ID */
      *edge_weight++ = 1.0;  /* all hyperedges same weight */
    }
  }
  *ierr = ZOLTAN_OK;
  return;
}
```

Figure 4.13: A ZOLTAN_HG_EDGE_WTS_FN query function

Calling the Zoltan library

If you have the Zoltan source, the file containing this code is **examples/C/simpleHG.c**. More information about the hypergraph parameters may be found in the Zoltan User's Guide at `http://cs.sandia.gov/Zoltan/ug_html/ug_alg_phg.html`.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "zoltan.h"
#include "simpleGraph.h"
#include "simpleQueries.h"

int main(int argc, char *argv[])
{
  int rc, i, ngids, nextIdx;
  float ver;
  struct Zoltan_Struct *zz;
  int changes, numGidEntries, numLidEntries, numImport, numExport;
  ZOLTAN_ID_PTR importGlobalGids, importLocalGids;
  ZOLTAN_ID_PTR exportGlobalGids, exportLocalGids;
  int *importProcs, *importToPart, *exportProcs, *exportToPart;
  int *gid_flags, *gid_list;

  /******************************************************************
  ** Initialize MPI and Zoltan
  ******************************************************************/

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

  rc = Zoltan_Initialize(argc, argv, &ver);

  if (rc != ZOLTAN_OK){
    MPI_Finalize();
    exit(0);
  }
```

```
/***************************************************************
** Create a Zoltan library structure for this instance of load
** balancing.  Set the parameters and query functions that will
** govern the library's calculation.  See the Zoltan User's
** Guide for the definition of these and many other parameters.
***************************************************************/

zz = Zoltan_Create(MPI_COMM_WORLD);

/* General parameters */

Zoltan_Set_Param(zz, "DEBUG_LEVEL", "0");
Zoltan_Set_Param(zz, "LB_METHOD", "HYPERGRAPH");
Zoltan_Set_Param(zz, "NUM_GID_ENTRIES", "1");
Zoltan_Set_Param(zz, "NUM_LID_ENTRIES", "0");
Zoltan_Set_Param(zz, "OBJ_WEIGHT_DIM", "0");
Zoltan_Set_Param(zz, "RETURN_LISTS", "ALL");

/* Hypergraph parameters */

Zoltan_Set_Param(zz, "HYPERGRAPH_PACKAGE", "PHG");
Zoltan_Set_Param(zz, "PHG_COARSENING_METHOD", "ipm");
Zoltan_Set_Param(zz, "PHG_COARSEPARTITION_METHOD", "greedy");
Zoltan_Set_Param(zz, "PHG_REFINEMENT_METHOD", "fm");
Zoltan_Set_Param(zz, "PHG_EDGE_WEIGHT_OPERATION", "error");
Zoltan_Set_Param(zz, "PHG_EDGE_SIZE_THRESHOLD", ".7");
Zoltan_Set_Param(zz, "ADD_OBJ_WEIGHT", "unit");

/* Query functions - defined in simpleQueries.h */

Zoltan_Set_Num_Obj_Fn(zz, get_number_of_objects, &hg_data);
Zoltan_Set_Obj_List_Fn(zz, get_hg_object_list, &hg_data);
Zoltan_Set_HG_Size_CS_Fn(zz, get_hg_size, &hg_data);
Zoltan_Set_HG_CS_Fn(zz, get_hg, &hg_data);
Zoltan_Set_HG_Size_Edge_Wts_Fn(zz, get_hg_num_edge_weights, &hg_data);
Zoltan_Set_HG_Edge_Wts_Fn(zz, get_hyperedge_weights, &hg_data);
```

```
/*****************************************************************
** Zoltan can now partition the vertices.
** In this simple example, we assume the number of partitions is
** equal to the number of processes.  Process rank 0 will own
** partition 0, process rank 1 will own partition 1, and so on.
*****************************************************************/

rc = Zoltan_LB_Partition(zz, /* input (all remaining fields are output) */
      &changes,        /* 1 if partitioning was changed, 0 otherwise */
      &numGidEntries,  /* Number of integers used for a global ID */
      &numLidEntries,  /* Number of integers used for a local ID */
      &numImport,      /* Number of vertices to be sent to me */
      &importGlobalGids,  /* Global IDs of vertices to be sent to me */
      &importLocalGids,   /* Local IDs of vertices to be sent to me */
      &importProcs,    /* Process rank for source of each incoming vertex */
      &importToPart,   /* New partition for each incoming vertex */
      &numExport,      /* Number of vertices I must send to other processes*
      &exportGlobalGids,  /* Global IDs of the vertices I must send */
      &exportLocalGids,   /* Local IDs of the vertices I must send */
      &exportProcs,    /* Process to which I send each of the vertices */
      &exportToPart);  /* Partition to which each vertex will belong */

if (rc != ZOLTAN_OK){
  printf("sorry...\n");
  MPI_Finalize();
  Zoltan_Destroy(&zz);
  exit(0);
}

/*****************************************************************
** In a real application, you would rebalance the problem now by
** sending the objects to their new partitions.  Your query
** functions would need to reflect the new partitioning.
*****************************************************************/
```

```
  /******************************************************************
  ** Free the arrays allocated by Zoltan_LB_Partition, and free
  ** the storage allocated for the Zoltan structure.
  ******************************************************************/

  Zoltan_LB_Free_Part(&importGlobalGids, &importLocalGids,
                      &importProcs, &importToPart);
  Zoltan_LB_Free_Part(&exportGlobalGids, &exportLocalGids,
                      &exportProcs, &exportToPart);

  Zoltan_Destroy(&zz);

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

  MPI_Finalize();

  return 0;
}
```

4.3 A C++ example

In this section we show a C++ example that solves the same recursive coordinate bisection problem that the C example in Section 4.2.1 solves.

We created a C++ class called **rectangularMesh**, which can represent the rectangular mesh shown in Figure 4.2. The query functions required by Zoltan are implemented as static methods of the **rectangularMesh** class. They take as an argument an object of type **rectangularMesh**. We will not list the entire class due to space constraints. If you obtain the Zoltan source you will find the class in **examples/CPP/rectangularMesh.h**. But just to give you an idea, here is the method that returns the number of objects owned by the process:

```
static int get_number_of_objects(void *data, int *ierr)
  {
  // Prototype: ZOLTAN_NUM_OBJ_FN
  // Return the number of objects I own.

  rectangularMesh *mesh = (rectangularMesh *)data;
  *ierr = ZOLTAN_OK;
  return mesh->get_num_my_IDs();
  }
```

We use static class methods, but your query functions could also be global C or C++ functions.

The example below can be found in the Zoltan source in **examples/CPP/simpleRCB.cp**

```
#include <rectangularGraph.h>
int main(int argc, char *argv[])
{
  // Initialize MPI and Zoltan

  int rank, size;
  float version;

  MPI::Init(argc, argv);
  rank = MPI::COMM_WORLD.Get_rank();
  size = MPI::COMM_WORLD.Get_size();

  Zoltan_Initialize(argc, argv, &version);

  // Create a Zoltan object

  Zoltan *zz = new Zoltan(MPI::COMM_WORLD);

  if (zz == NULL){
    MPI::Finalize();
    exit(0);
  }

  // Create a simple rectangular mesh.  It's vertices are the
  // objects to be partitioned.

  rectangularMesh *mesh = new rectangularMesh();

  mesh->set_x_dim(10);
  mesh->set_y_dim(8);
  mesh->set_x_stride(1);
  mesh->set_y_stride(4);

  // General parameters:

  zz->Set_Param("DEBUG_LEVEL", "0");     // amount of debug messages desired
  zz->Set_Param("LB_METHOD", "RCB");     // recursive coordinate bisection
```

```
zz->Set_Param("NUM_GID_ENTRIES", "1"); // number of integers in a global ID
zz->Set_Param("NUM_LID_ENTRIES", "1"); // number of integers in a local ID
zz->Set_Param("OBJ_WEIGHT_DIM", "1");  // dimension of a vertex weight
zz->Set_Param("RETURN_LISTS", "ALL");  // return all lists in LB_Partition

// RCB parameters:

zz->Set_Param("KEEP_CUTS", "1");              // save decomposition
zz->Set_Param("RCB_OUTPUT_LEVEL", "0");       // amount of output desired
zz->Set_Param("RCB_RECTILINEAR_BLOCKS", "1"); // create rectilinear regions

// Query functions:

zz->Set_Num_Obj_Fn(rectangularMesh::get_number_of_objects, (void *)mesh);
zz->Set_Obj_List_Fn(rectangularMesh::get_object_list, (void *)mesh);
zz->Set_Num_Geom_Fn(rectangularMesh::get_num_geometry, NULL);
zz->Set_Geom_Multi_Fn(rectangularMesh::get_geometry_list, (void *)mesh);

// Zoltan can now partition the vertices in the simple mesh.

int changes;
int numGidEntries;
int numLidEntries;
int numImport;
ZOLTAN_ID_PTR importGlobalIds;
ZOLTAN_ID_PTR importLocalIds;
int *importProcs;
int *importToPart;
int numExport;
ZOLTAN_ID_PTR exportGlobalIds;
ZOLTAN_ID_PTR exportLocalIds;
int *exportProcs;
int *exportToPart;

int rc = zz->LB_Partition(changes, numGidEntries, numLidEntries,
  numImport, importGlobalIds, importLocalIds, importProcs, importToPart,
  numExport, exportGlobalIds, exportLocalIds, exportProcs, exportToPart);
  if (rc != ZOLTAN_OK){
    printf("Partitioning failed on process %d\n",rank);
    MPI::Finalize();
    delete zz;
    delete mesh;
    exit(0);
  }

  // In a real application, you would rebalance the problem now by
  // sending the objects to their new partitions.

  // Free the arrays allocated by LB_Partition, and free
  // the storage allocated for the Zoltan structure and the mesh.

  zz->LB_Free_Part(&importGlobalIds, &importLocalIds, &importProcs,
                   &importToPart);
  zz->LB_Free_Part(&exportGlobalIds, &exportLocalIds, &exportProcs,
                   &exportToPart);

  delete zz;
  delete mesh;

  // all done ////////////////////////////////////////////////////

  MPI::Finalize();

  return 0;
}
```