Sandia National Laboratories

About Sandia
Capabilities
Programs
Contacting Us
News and Events
Search
Home

Zoltan Home Page
Zoltan User's Guide
Zoltan Developer's Guide
Zoltan Project Description
Papers and Presentations
How to Cite Zoltan
Download Zoltan
Report a Zoltan Bug
Contact Zoltan Developers

Zoltan:
Parallel Partitioning, Load Balancing and Data-Management Services

Developer's Guide

The Zoltan Team
Sandia National Laboratories:
Erik Boman
Karen Devine
Robert Heaphy
Bruce Hendrickson
Vitus Leung
Lee Ann Riesen
Courtenay Vaughan

Ohio State University
Umit Catalyurek
Doruk Bozdag

National Institute of Standards and Technology
William F. Mitchell

Zoltan Developer's Guide, Version 3.0

DOWNLOAD PDF VERSION HERE.

[Zoltan Home Page | Next: Introduction and General Principles]

Introduction and General Principles

The goal of the Zoltan project is to design a general-purpose tool for parallel data management for unstructured, dynamic applications. This tool includes a suite of load-balancing algorithms, an unstructured communication package, distributed data directories, and dynamic debugging tools that can be used by a variety of applications. It will, thus, be used by many application developers and be added to by many algorithm developers. Software projects of this scale need general guidelines and principles so that the code produced is easily maintained and added to. We have tried to keep restrictions on developers to a minimum. However, we do require that a few coding practices be followed. These guidelines are described in the following sections:

[Table of Contents | Next: Philosophy of Zoltan | Previous: Table of Contents]

Philosophy of Zoltan

The Zoltan library is designed to be a general-purpose tool-kit providing a variety of parallel data management services to a wide range of scientific applications (see the Zoltan User's Guide). To enable general use of the library, the library does not directly access the data structures of an application. Instead, the library obtains information it needs through an object-oriented interface between Zoltan and the application. This interface uses call-back query functions to gather information. An application developer must write and register these query functions before using Zoltan. The intent, however, is that the number and complexity of these query functions are low, allowing applications to easily interface with the library. In addition, new algorithm development would use the same query functions as previous algorithms, enabling applications to use new algorithms without changes to the query functions.

In developing new algorithms for Zoltan, the developer must write the code that calls the query functions to build the needed data structures for the algorithm. However, the application need not change its query functions. Thus, new algorithms can be added to the library and used by an application with minimal effort on the part of the application developer.

[Table of Contents | Next: Coding Principles | Previous: Introduction]

Coding Principles in Zoltan

Include files

Include files should be used for function prototypes, macro definitions, and data structure definitions. The convention used is that external function prototypes and data structure definitions required by more than one module are stored in include files named **_const.h* (e.g., *zz/zz_const.h*). Include files with static function prototypes or static data structure definitions (i.e., files that are included in only one module) are named **.h* (e.g., *rcb/rcb.h*).

The include file *include/zoltan.h* contains the Zoltan interface; it should be included by C application source files that call Zoltan. C++ applications that use the C++ interface should include *include/zoltan_cpp.h* instead.

The include file *zz/zz_const.h* describes the principle Zoltan data structures. As these data structures are used heavily by the algorithms in Zoltan, *zz/zz_const.h* should be included in most source files of Zoltan.

Every Zoltan C language header file should be surrounded with an **extern "C" {}** declaration. The declaration must occur after every other **#include** statement, and before all function declarations. This declaration tells a C++ compiler not to mangle the names of functions declared in that header file.

```
#ifndef __EXAMPLE_H
#define __EXAMPLE_H

#include "mpi.h"
#include "zoltan_types.h"
#include "zoltan_align.h"

#ifdef __cplusplus
extern "C" {
#endif

int func1(int a, int b);
double dfunc(int a, int b, int c);

#ifdef __cplusplus
} /* closing bracket for extern "C" */
#endif

#endif /* __EXAMPLE_H */
```

*Example of C language header file with **extern "C"***

If an **#include** statement appears after the opening of the **extern "C" {}** declaration, the included file may cause **mpi.h** or some other system header file to be processed. When compiling with a C++ compiler, this usually leads to compile errors because the function names in some of those headers are supposed to be mangled.

It should not be necessary to use the declaration in all header files, but rather only in header files that are used in C++ applications. But experience has taught us that you never know what header files will end up being included, and that one that is not included now, may be included in the future when someone adds an **#include** statement to a file. To save someone the effort later on of figuring out why their C++ compilation is failing, please include the **extern "C" {}** declaration in every header file, even if at this point in time you do not believe it will ever be included in the compilation of a C++ application.

Global variables

The use of global variables is highly discouraged in Zoltan. In limited cases, static global variables can be tolerated within a source file of an algorithm. However, developers should keep in mind that several Zoltan structures may be used by an application, with each structure using the same algorithm. Thus, global variables set by one invocation of a routine may be reset by other invocations, causing errors in the algorithms. Global variable names may also conflict with variables used elsewhere in the library or application, causing unintended side-effects and complicating debugging. For greatest robustness, developers are asked NOT to use global variables in their algorithms. See Data Structures for ideas on avoiding the use of global variables.

Function Names

In order to avoid name conflicts with applications and other libraries, all non-static functions should be prepended with **Zoltan_**. Moreover, function names should, in general, include their module names; e.g., **Zoltan_HSFC_Box_Assign** is part of the HSFC module of Zoltan. As a general rule, each new word in a function name should be capitalized (for example, **Zoltan_Invert_Lists**). Static Zoltan functions do not have to follow these rules.

Parallel Communication

All communication in the Zoltan library should be performed through MPI communication routines. The MPI interface was chosen to enable portability to many different platforms. It will be especially important as the code is extended to heterogeneous computing systems.

Some useful communication utilities are provided within the library to perform unstructured communication and synchronization. See Unstructured Communication Utilities and Parallel Computing.

Memory Management

It is strongly suggested that all memory allocation in the library is handled using the functions supplied in *Utilities/Memory*. Use of these functions will make debugging and maintenance of the library much easier as the library gets larger. See Memory Management Utilities for more information on these utilities.

For memory that is returned by Zoltan to an application, however, special memory allocation functions must be used to maintain compatibility with both C and Fortran90 applications. See Memory Management in Zoltan Algorithms for more information.

One of the few data types specified for use in the Zoltan interface is the **ZOLTAN_ID_PTR** type used for global and local object identifiers (IDs). Macros simplifying and providing error checking for ID allocation and manipulation are provided.

Errors, Warnings, and Return Codes

If an error or warning occurs in the Zoltan library, a message should be printed to *stderr* (using one of the printing macros below), all memory

allocated in the current function should be freed, and an error code should be returned. The Zoltan library should never "exit"; control should always be returned to the application with an error code. The error codes are defined in *include/zoltan_types.h*.

Currently, this philosophy is not strictly followed in all portions of Zoltan. Efforts are underway to bring existing code up-to-date, and to follow this rule in all future development.

ZOLTAN_PRINT_ERROR(int *processor_number*, char **function_name*, char **message*)
ZOLTAN_PRINT_WARN(int *processor_number*, char **function_name*, char **message*)

Macros for printing error and warning messages in Zoltan. The macros are defined in *Utilities/shared/zoltan_util.h*.

Arguments:

processor_number	The processor's rank in the Zoltan communicator. The value -1 can be used if the rank is not available.
function_name	A string containing the name of the function in which the error or warning occurred.
message	A string containing the error or warning message.

[Table of Contents | Next: Zoltan Quality Assurance | Previous: Philosophy]

Zoltan Developer's Guide | *Next* | *Previous*

Zoltan Quality Assurance

This document describes the Software Quality Assurance (SQA) policies and procedures used in the Zoltan project. Zoltan developers at Sandia or under contract to Sandia are required to follow these software development policies.

Quality Policy

Sandia´s ASC Quality Management Council (AQMC) developed and manages the Quality Assurance Program (QAP) for Sandia´s ASC program. The AQMC chartered the development of the *Sandia National Laboratories Advanced Simulation and Computing (ASC) Software Quality Plan, Part 1: ASC Software Quality Engineering Practices, Version 2.0* document (SAND 2006-5998) as the practical SQA guidance for projects like Zoltan. A companion document, *Sandia National Laboratories Advanced Simulation and Computing (ASC) Software Quality Plan, Part 2: Mappings for the ASC Software Quality Practices* (SAND 2006-5997), shows how these practices satisfy corporate policies including CPR001.3.6, Corporate Software Engineering Excellence, and DOE/NNSA orders 414.1C and QC-1 rev 10.

The Zoltan project is committed to a program of quality improvement in compliance with the *ASC Software Quality Engineering Practices* document. The Zoltan Team Leader is the owner of the Zoltan quality system. Zoltan developers at Sandia or under contract to Sandia are required to follow these software development practices. The Zoltan team shall participate in all reporting processes, audits, and assessments as directed by the AQMC.

Quality Definition

QC-1 rev 10 defines quality as "the degree to which customer requirements are met."

The Zoltan project accepts the following definition of quality: "the totality of characteristics of a product or service that bear on its ability to satisfy stated or implied needs." This is Juran´s "fitness for use" definition of quality (ANSI/ASQC A8402-1994.) This superior definition of quality fully satisfies the QC-1 rev 10 definition. This definition is also more useful in a research environment where the requirements are derived from a research proposal rather than directly from customers and end users.

Classification of Defects

The Zoltan project accepts the following system of classification of defects:

Critical: A defect that could lead to loss of life, significant environmental damage, or substantial financial loss.
Major: A non critical defect that significantly impacts Zoltan's fitness for use.
Minor: A (non critical, non major) defect that reasonably impacts Zoltan's fitness for use.
Incidental: Any other defect which does not reasonably reduce Zoltan's fitness for use.

Release Policy

Only the Zoltan team leader may authorize (certify) a release. The Zoltan team leader shall not release software with any known critical or major defects. User registration shall allow the Zoltan team to notify all Sandia and ASC users and to recall their defective software if a critical or major defect is discovered after release. The Zoltan team leader may determine that it is acceptable to release software with known minor or incidental defects.

Software Quality Tools

Because of the small scale of the Zoltan Project, only a few, simple tools are required for use by Zoltan developers:

CVS: maintains code, documentation, meeting notes, emails, and QA program artifacts;
Purify, PureCoverage, Quantify (Rational), Valgrind, gdb: for dynamic code testing, coverage measurements, and performance analysis;
Bugzilla: tracks bugs, requests for changes, and enhancements;
Mailman: creates email lists to automatically notify users by area(s) of interest;
Makefiles: ensures proper compilation and linking for all supported platforms; and
Zoltan Test Script: runs integration, regression, release and acceptance testing.

Software Quality Processes

Bug Reporting, Issue Tracking, Enhancement Requests: All of these items are now directly entered into Bugzilla by developers and users. This "process" is built into the tool. Detailed instructions for using Bugzilla are found on the Zoltan web page. Bugzilla also provides query and report features for tracking the status of entered items;.

A process is defined as a sequence of steps performed for a given purpose (IEEE Std. 610.12.) Zoltan´s other processes are defined as checklists because checklists are one of the seven fundamental quality tools. These checklists are also the primary artifact created when following a process. Currently the following processes are defined:

Development: (not currently used) defines the software development process including requirements, design, implementation, testing, reviews, and approvals;
Release: defines the release process including testing requirements and creation of the release product;
Request: defines the process of capturing user requests for new features;
Note: this process is now obsolete. Request processes in progress may continue until complete but new requests should use Bugzilla;
Requirement: the process of capturing user comments that may become requirements after review and approval;
Note: this process is now obsolete. Requirement processes in progress may continue until complete but new requirements should use Bugzilla;
Review: defines the materials reviewed prior to acceptance for Zoltan release;
Note: Developers are encouraged to use Bugzilla to enter the specific review process rather than use the Review checklist. At this time this is an trial effort and either method may be used.
Third Party Software:defines the steps required to obtain, manage, use, and test for software created outside of Zoltan and the ASC program; and
Training: defines the material a new developer must read, required skills to demonstrate and computer accounts that must be obtained.

Zoltan's software quality process checklists define how work may be performed, including process ownership, authorization to perform, activities and their sequence (when sequencing is required), process instructions, metrics, and identification of who performed each activity.

The only allowed source for process checklists is Zoltan's CVS repository in the SQA_templates directory (under Zoltan_Internal.) A Zoltan developer initiates a process by obtaining the current CVS version of the process, renaming it, and committing the renamed process checklist back into CVS in an appropriate directory on the same day. The process may continue under this committed version even if its original process is later superseded unless specifically requested by the Team Leader. After one or more activities are completed, the process checklist is updated to reflect the results and committed back to CVS (with appropriate comments.) A process is completed when all required activities are completed including reviews and approvals (as necessary), and committed to CVS. The final CVS comment should indicate that the process is complete.

Zoltan´s implementation of the *ASC Software Quality Engineering Practices*

The following is brief description **for Zoltan developers** about the Zoltan project´s implementation of the *ASC Software Quality Engineering Practices* (SAND 2006-5998):

PR1. Document and maintain a strategic plan.
The Zoltan web page has a direct hyperlink to the Zoltan Project Description defining its mission and philosophy. The Zoltan project has a strong association with the Trilinos project to share in the development of common software engineering practices and sharing of appropriate tools and experience.

PR2. Perform a risk-based assessment, determine level of formality and applicable practices, and obtain approvals.
The Zoltan project has an approved level of formality (medium) for its deliverable software. Its biggest technical risk results from providing parallel solutions to NP hard partitioning problems. Technical risks are mitigated by collaborations within Sandia and internationally. The most significant non-technical risk is the conflicting priorities of Zoltan developers working on many other projects simultaneously.

PR3. Document lifecycle processes and their interdependencies, and obtain approvals.
The Zoltan project follows the *Trilinos Software Lifecycle Model* (SAND 2006-6929). It also follows the ANSI/ASQ Z1.13-1999 standard *Quality Guidelines for Research* which is compatible with the research phase in the Trilinos Lifecycle model.

PR4. Define, collect, and monitor appropriate process metrics.
The Zoltan project is committed to comply fully with the new and evolving AQMC requirements for collecting and reporting "defect" metrics. Other metrics determined by Zoltan´s continual process improvement process (**PR 5**) will be implemented.

PR5. Periodically evaluate quality problems and implement process improvements.
The Zoltan project has built the Deming/Shewhart process improvement cycle PDCA (Plan, Do, Check, Act) into all of its process checklists. This is the most effective process improvement technique known. It is recommended by ISO 9001:2000.

PR6. Identify stakeholders and other requirements sources.
The Zoltan project´s primary stakeholders are the ASC applications using Zoltan including SIERRA, ACME, ALEGRA/NEVADA, XYCE, and Trilinos.

PR7. Gather and manage stakeholders´ expectations and requirements.
The Zoltan project´s primary input from ASC applications´ expectations and requirements are via their communication of Zoltan´s role in meeting their ASC milestones. Since Zoltan is an "enabling technology," these requirements are broadly stated performance improvement needs. The Zoltan team actively anticipates and develops load balancing software for the future needs of the Sandia research community before they actually become formal requirements.

PR8. Derive, negotiate, manage, and trace requirements.
Zoltan project requirements normally derive from its funded research proposals which state research goals. This is a normal procedure in a research environment (see ANSI/ASQ Z1.13-1999). Periodic and final reports document the success in meeting these research goals.

PR9. Identify and analyze risk events.
All Zoltan developers should report any new or changed risks via the zoltan-dev email target for evaluation by the Team Lead.

PR10. Define, monitor, and implement the risk response.
The Zoltan team will create a corrective action plan whenever any condition threatens to adversely impact the Zoltan project resources or schedule.

PR11. Create and manage the project plan.
ANSI/ASQ Z1.13-1999 states that the research proposal is equivalent to a project plan in a research environment. The Team Leader assigns responsibilities, deliverables, resources, and schedules in order to manage the project.

PR12. Track project performance versus project plan and implement needed (corrective) actions.
The Team Leader periodically tracks responsibilities, deliverables, resources, and schedules in order to manage the project.

PR13. Communicate and review design.
The Zoltan architecture is fully documented in the Zoltan Developer´s Guide. New features are originally documented and reviewed in team discussions to the zoltan-dev email target. Prior to release, the design documentation is finalized in both the Zoltan Developer´s Guide and the Zoltan User´s Guide.

PR14. Create required software and product documentation.
Developers will follow the Zoltan Development Process Checklist.

PR15. Identify and track third party software products and follow applicable agreements.
Developers will follow the Zoltan Third Party Software Process Checklist.

PR16. Identify, accept ownership, and manage the assimilation of other software products.
Not applicable since Zoltan does not "assimilate" third party software.

PR17. Perform version control of identified software product artifacts.
All software and process artifact are under maintained CVS as early as reasonable after their creation.

PR18. Record and track issues associated with the software product.
Developers will use Bugzilla to record and track issues.

PR19. Ensure backup and disaster recovery of software product artifacts.
Nightly backups, periodic offsite backups, and disaster recovery are services provided by the CSRI computer support staff. Disaster recovery has been successfully performed from real problems.

PR20. Plan and generate the release package.
Developers will follow the Zoltan Release Process Checklist.

PR21. Certify that the software product (code and its related artifacts) is ready for release and distribution.
The Zoltan Team Leader will certify any version of Zoltan for release via an email to zoltan-dev target.

PR22. Distribute release to customers.
Zoltan files are released via a download from the Zoltan web site. The Zoltan Team Leader will make the download available after certification. (Research versions of the Zoltan software are directly available to collaborators for development.)

PR23. Define and implement a customer support plan.
(See **PR 6** for a list of ASC stakeholders.) The Zoltan team provides one-on-one training whenever requested and quickly responds to any user complaint.

PR24. Implement the training identified in the customer support plan.
See **PR 23** above. If additional training is ever requested, the Zoltan project will piggy back on the annual Trilinos Users Group meeting with a training session on using Zoltan.

PR25. Evaluate customer feedback to determine customer satisfaction.

PR 26 Develop and maintain a software verification plan.
Developers are expected to create new tests for the Zoltan test suite when new features are added to Zoltan.

Currently, a new test framework based on FAST/EXACT is being implemented. Documentation about this test framework is under preparation. A process checklist will be developed around the steps required to add new tests to the suite and to run the suite.

PR27. Conduct tests to demonstrate that acceptance criteria are met and to ensure that previously tested capabilities continue to perform as expected.
This practice is a subset of the Zoltan Release Process Checklist.

PR28. Conduct independent technical reviews to evaluate adequacy with respect to requirements.
Developers will follow the Zoltan Review Process Checklist. ANSI/ASQ Z1.13-1999 states that the peer reviewed publications and conference presentations are a normal form of technical review in the research environment.

PR29. Determine project team training needed to fulfill assigned roles and responsibilities.
New developers will follow the Zoltan Training Process for new team members.

PR30. Track training undertaken by project teams.
Zoltan developers are encouraged to participate in the annual Trilios Users Group (TUG) meeting which provides sessions for SQA/SQE training to developers. Attendance records are kept for this event and for any Zoltan team meetings that provide training. Sandia provides many other opportunities for training including formal courses and periodic internal software developers conferences. External conferences (e. g., IPDPS and SIAM) are counted as technical training.

[Table of Contents | Next: Zoltan Distribution | Previous: Coding Principles in Zoltan]

Zoltan Distribution

The organization of the Zoltan software distribution is described in the following sections. Full pathnames are specific to Sandia's 980 SON LAN.

[Table of Contents | Next: CVS | Previous: Zoltan Quality Assurance]

CVS

The source code and documentation for the Zoltan library is maintained under the Concurrent Versions System (CVS) software. CVS allows multiple developers to edit their own copies of the software and merges updated versions with the developers' own versions.

On Sandia's 980 SON LAN, CVS is accessed through the following path:

    /Net/local/gnu/bin/cvs for Sun workstations running Solaris.

Developers must set the CVSROOT environment variable to the repository directory:

    setenv CVSROOT *username*@software.sandia.gov:/space/CVS-Zoltan

where *username* is the developer's username on the CVS server software.sandia.gov. To get a working copy of the Zoltan software, the CVS check-out facility is used:

    cvs checkout -P Zoltan

Other useful CVS commands update a developer's working directory, merging the developer's changes with those in the repository:

    cvs update

and check into the repository a developer's changes:

    cvs commit

The UNIX man page for cvs contains information on these and other useful CVS commands.

[Table of Contents | Next: Layout of Directories | Previous: Zoltan Distribution]

Zoltan Developer's Guide | *Next* | *Previous*

Layout of Directories

The source code is organized into several subdirectories within the Zoltan main directory. General interface routines are stored in a single directory. Communication and memory allocation utilities available to all algorithms are in separate directories. Each load-balancing method (or family of methods) should be stored in its own directory. In addition, a courtesy copy of the ParMETIS graph-partitioning package is included in the top-level directory *ParMETIS*.

In the following table, the source-code directories currently in the Zoltan directory are listed and described.

Directory	Description
zz	General Interface definitions, Zoltan data structure definitions, interface functions and functions related to the interface See Interface Functions, ID Data Types, and Data Structures.
lb	Load-Balancing interface routines, and load-balancing data structure definitions.
all	Special memory allocation functions for memory returned by Zoltan to an application.
par	Parallel computing routines.
param	Routines for changing parameter values at runtime.
parmetis	Routines to access the ParMETIS and Jostle partitioning libraries.
rcb	Recursive Coordinate Bisection (RCB) and Recursive Inertial Bisection (RIB) algorithms.
hsfc	Hilbert Space-Filling Curve partitioning algorithm.
bsfc	Binned Space-Filling Curve partitioning algorithm.
oct	Rensselaer Polytechnic Institute's octree partitioning algorithms.
reftree	William Mitchell's Refinement Tree Partitioning algorithm and refinement tree data structure.
timer	Timing routines.
ch	Routines to read Chaco input files and build graphs for the driver program *zdrive.*
ha	Routines to support heterogeneous architectures.
fort	Fortran (F90) interface for Zoltan.
Utilities/shared	Simple functions and utilities shared by Zoltan and other Zoltan Utilities.
Utilities/Memory	Memory management utilities
Utilities/Communication	Unstructured communication utilities
Utilities/DDirectory	Distributed Data Directory utilities
Utilities/Config	Platform-specific makefile definitions for compiler, library and include-file paths.
driver	Test driver programs, *zdrive* and *zCPPdrive*.
fdriver	Fortran90 version of the test driver program.
examples	Simple examples written in C and C++ that use Zoltan.
docs/Zoltan_html	Zoltan documentation and home page.
docs/Zoltan_html/ug_html	User's guide in HTML format.

docs/Zoltan_html/dev_html	Developer's guide in HTML format.
docs/Zoltan_pdf	PDF versions of the Zoltan User's Guide and Developer's Guide.
docs/internal	SQA documents for the Zoltan project.

The directory structure of the Zoltan distribution.

[Table of Contents | Next: Compilation | Previous: CVS]

Compilation and Makefiles

The Zoltan distribution includes a main (top-level) Makefile with targets for the Zoltan library, the test driver programs, and some graphical tools. When the library is compiled for a specific target platform, *A*, the top-level Makefile obtains platform-specific values for platform *A* from the configuration file *Utilities/Config/Config.A*. This file should be edited to reflect the environment of the target platform *A*. A subdirectory, *Obj_A*, is created, and Makefile_sub is copied into that directory for use by gmake.

New source code files are added to the Zoltan Makefiles in two ways. Files added to existing directories are added to the source files listed in the "<*directory_name*>_CSRC" and "<*directory_name*>_INC" variables in *Zoltan/Makefile*, where <*directory_name*> corresponds to the existing Zoltan directory name; the files will then be included in the compilation of Zoltan. For new source code files in new directories, new variables "<*directory_name*>_CSRC" and "<*directory_name*>_INC" should be added to *Zoltan/Makefile*. These variables should also be included in the "ZOLTAN_CSRC" variable and in the zscript target. The variables "ALL_CSRC" and "ALL_INC" can be used as examples.

New algorithms can be added as separate libraries with which Zoltan may link. The implementation of the ParMETIS interface in Zoltan can serve as an example. Within the *Utilities/Config* files, pathnames for the new libraries and their include files can be specified. Within *Zoltan/Makefile*, tests should be added for the definition of these paths. If they are defined, appropriate information should be added to the THIRD_PARTY_LIBS, THIRD_PARTY_LIBPATH, and THIRD_PARTY_INCPATH variables in *Zoltan/Makefile*.

[Table of Contents | Next: Zoltan Interface and Data Structures | Previous: Layout of Directories]

Zoltan Interface and Data Structures

The interface functions, data types and data structures for the Zoltan library are described in the following sections:

[Table of Contents | Next: Interface Functions | Previous: Compilation]

Interface Functions

The interface to the Zoltan library is defined in the file *include/zoltan.h*. This file should be included in application programs that use Zoltan. It is also included in *zz/zz_const.h*, which should be included by most Zoltan files to provide access to the Zoltan data structures described below.

In *include/zoltan.h*, the enumerated type **ZOLTAN_FN_TYPE** defines the application query function types (e.g., **ZOLTAN_NUM_OBJ_FN_TYPE** and **ZOLTAN_OBJ_LIST_FN_TYPE**). The interface query routines (e.g., **ZOLTAN_NUM_OBJ_FN** and **ZOLTAN_OBJ_LIST_FN**) and their argument lists are defined as C type definitions (typedef). These type definitions are used by the application developer to implement the query functions needed for the application to use Zoltan.

Prototypes for the Zoltan interface functions (e.g., **Zoltan_LB_Partition** and **Zoltan_Migrate**) are also included in *include/zoltan.h*. Interface functions are called by the application to register functions, select a load-balancing method, invoke load balancing and migrate data.

The interface to the C++ version of the Zoltan library is in the file *include/zoltan_cpp.h*. This file defines the **Zoltan** class, representing a **Zoltan_Struct** data structure and the functions which operate upon it. The conventions used to wrap C library functions as C++ library functions are described in the chapter **C++ Interface**. A C++ program that uses Zoltan includes *include/zoltan_cpp.h* instead of *include/zoltan.h*.

For more detailed information on Zoltan's query and interface functions, please see the Zoltan User's Guide.

[Table of Contents | Next: ID Data Types | Previous: Zoltan Interface and Data Structures]

Zoltan Developer's Guide | *Next* | *Previous*

ID Data Types

Within Zoltan, objects are identified by a global identification (ID) value provided by the application. This global ID must be unique across all processors. The application may also provide a local ID value that it can use for faster location of objects within its own data structure. For example, local array indices to objects' data may be provided as the local IDs; these indices can then be used to directly access data in the query functions. Zoltan does not use these local IDs, but since it must pass them to the application in the interface query functions, it must store them with the objects' data. ID data types and macros for manipulating IDs are described below.

IDs and Arrays of IDs

Zoltan stores each global and local ID as an array of unsigned integers. Arrays of IDs are passed to the application as a one-dimensional array of unsigned integers with size *number_of_IDs* * *number_of_entries_per_ID*. A type definition **ZOLTAN_ID_PTR** (in *include/zoltan_types.h*) points to an ID or array of IDs. The number of array entries per ID can be set by the application using the NUM_GID_ENTRIES and NUM_LID_ENTRIES parameters.

Allocating IDs

Macros that simplify the allocation of global and local IDs are described in the table below. These macros provide consistent, easy-to-use memory allocation with error checking and, thus, their use is highly recommended. Each macro returns NULL if either a memory error occurs or the number of IDs requested is zero.

ZOLTAN_ID_PTR ZOLTAN_MALLOC_GID(struct Zoltan_Struct *zz);	Allocates and returns a pointer to a single global ID.
ZOLTAN_ID_PTR ZOLTAN_MALLOC_LID(struct Zoltan_Struct *zz);	Allocates and returns a pointer to a single local ID.
ZOLTAN_ID_PTR ZOLTAN_MALLOC_GID_ARRAY(struct Zoltan_Struct *zz, int *n*);	Allocates and returns a pointer to an array of *n* global IDs, where the index into the array for the i^{th} global ID is *i**NUM_GID_ENTRIES.
ZOLTAN_ID_PTR ZOLTAN_MALLOC_LID_ARRAY(struct Zoltan_Struct *zz, int *n*);	Allocates and returns a pointer to an array of *n* local IDs, where the index into the array for the i^{th} local ID is *i**NUM_LID_ENTRIES.
ZOLTAN_ID_PTR ZOLTAN_REALLOC_GID_ARRAY(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *ptr*, int *n*);	Reallocates and returns a pointer to an array of *n* global IDs, replacing the current array pointed to by *ptr*.
ZOLTAN_ID_PTR ZOLTAN_REALLOC_LID_ARRAY(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *ptr*, int *n*);	Reallocates and returns a pointer to an array of *n* local IDs, replacing the current array pointed to by *ptr*.

Common Operations on IDs

In addition, macros are defined for common operations on global and local IDs. These macros include error checking when appropriate and account for different values of NUM_GID_ENTRIES and NUM_LID_ENTRIES. Use of these macros improves code robustness and simplifies code maintenance; their use is highly recommended.

void **ZOLTAN_INIT_GID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *id*);	Initializes all entries of the global ID *id* to zero; *id* must be allocated before calling **ZOLTAN_INIT_GID**.
void **ZOLTAN_INIT_LID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *id*);	Initializes all entries of the local ID *id* to zero; *id* must be allocated before calling **ZOLTAN_INIT_LID**.
void **ZOLTAN_SET_GID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *tgt*, **ZOLTAN_ID_PTR** *src*);	Copies the global ID *src* into the global ID *tgt*. Both *src* and *tgt* must be allocated before calling **ZOLTAN_SET_LID**.
void **ZOLTAN_SET_LID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *tgt*, **ZOLTAN_ID_PTR** *src*);	Copies the local ID *src* into the local ID *tgt*. Both *src* and *tgt* must be allocated before calling **ZOLTAN_SET_LID**.
int **ZOLTAN_EQ_GID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *a*, **ZOLTAN_ID_PTR** *b*);	Returns TRUE if global ID *a* is equal to global ID *b*.
void **ZOLTAN_PRINT_GID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *id*);	Prints all entries of a single global ID *id*.
void **ZOLTAN_PRINT_LID**(struct Zoltan_Struct *zz, **ZOLTAN_ID_PTR** *id*);	Prints all entries of a single local ID *id*.

[Table of Contents | Next: Data Structures | Previous: Interface Functions]

Zoltan Developer's Guide | *Next* | *Previous*

Data Structures

The **Zoltan_Struct** data structure is the main data structure for interfacing between Zoltan and the application. The application creates an **Zoltan_Struct** data structure through a call to **Zoltan_Create**. Fields of the data structure are then set through calls from the application to interface routines such as **Zoltan_Set_Param** and **Zoltan_Set_Fn**. The fields of the **Zoltan_Struct** data structure are listed and described in the table below. See the Zoltan User's Guide for descriptions of the function types used in the **Zoltan_Struct**.

A **Zoltan_Struct** data structure *zz* is passed from the application to Zoltan in the call to **Zoltan_LB_Partition**. This data structure is passed to the individual load-balancing routines. The *zz->LB.Data_Structure* pointer field should point to the main data structures of the particular load-balancing algorithm so that the data structures may be preserved for future calls to **Zoltan_LB_Partition** and so that separate instances of the same load-balancing algorithm (with different **Zoltan_Struct** structures) can be used by the application.

Fields of Zoltan_Struct	Description
MPI_Comm *Communicator*	The MPI communicator to be used by the Zoltan structure; set by **Zoltan_Create**.
int *Proc*	The rank of the processor within *Communicator*; set in **Zoltan_Create**.
int *Num_Proc*	The number of processors in *Communicator*; set in **Zoltan_Create**.
int *Num_GID*	The number of array entries used to represent a global ID. Set via a call to **Zoltan_Set_Param** for NUM_GID_ENTRIES.
int *Num_LID*	The number of array entries used to represent a local ID. Set via a call to **Zoltan_Set_Param** for NUM_LID_ENTRIES.
int *Debug_Level*	A flag indicating the amount of debugging information that should be printed by Zoltan.
int *Fortran*	A flag indicating whether or not the structure was created by a call from Fortran.
PARAM_LIST * *Params*	A linked list of string pairs. The first item in each pair is the name of a modifiable parameter. The second string is the new value the parameter should adopt. These string pairs are read upon invocation of a Zoltan algorithm and the appropriate parameter changes are made. This design allows for different Zoltan structures to have different parameter settings.
int *Deterministic*	Flag indicating whether algorithms used should be forced to be deterministic; used to obtain completely reproducible results. Set via a call to **Zoltan_Set_Param** for DETERMINISTIC.
int *Obj_Weight_Dim*	Number of weights per object. Set via a call to **Zoltan_Set_Param** for OBJ_WEIGHT_DIM.
int *Edge_Weight_Dim*	For graph algorithms, number of weights per edge. Set via a call to **Zoltan_Set_Param** for EDGE_WEIGHT_DIM.

int *Timer*	Timer type that is currently active. Set via a call to **Zoltan_Set_Param** for TIMER.
ZOLTAN_NUM_EDGES_FN * *Get_Num_Edges*	A pointer to an application-registered function that returns the number of edges associated with a given object. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Edges_Fn**.
void **Get_Num_Edges_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Num_Edges*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Edges_Fn**.
ZOLTAN_EDGE_LIST_FN * *Get_Edge_List*	A pointer to an application-registered function that returns a given object's neighbors along its edges. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Edge_List_Fn**.
void **Get_Edge_List_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Edge_List*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Edge_List_Fn**.
ZOLTAN_NUM_GEOM_FN * *Get_Num_Geom*	A pointer to an application-registered function that returns the number of geometry values needed to describe the positions of objects. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Geom_Fn**.
void **Get_Num_Geom_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Num_Geom*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Geom_Fn**.
ZOLTAN_GEOM_FN * *Get_Geom*	A pointer to an application-registered function that returns a given object's geometry information (e.g., coordinates). Set in **Zoltan_Set_Fn** or **Zoltan_Set_Geom_Fn**.
void **Get_Geom_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Geom*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Geom_Fn**.
ZOLTAN_NUM_OBJ_FN * *Get_Num_Obj*	A pointer to an application-registered function that returns the number of objects assigned to the processor before load balancing. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Obj_Fn**.
void **Get_Num_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Num_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Obj_Fn**.
ZOLTAN_OBJ_LIST_FN * *Get_Obj_List*	A pointer to an application-registered function that returns arrays of objects assigned to the processor before load balancing. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Obj_List_Fn**.
void **Get_Obj_List_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Obj_List*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Obj_List_Fn**.
ZOLTAN_FIRST_OBJ_FN * *Get_First_Obj*	A pointer to an application-registered function that returns the first object assigned to the processor before load balancing. Used with *Get_Next_Obj* as an iterator over all objects. Set in **Zoltan_Set_Fn** or **Zoltan_Set_First_Obj_Fn**.
void **Get_First_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_First_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_First_Obj_Fn**.

ZOLTAN_NEXT_OBJ_FN * *Get_Next_Obj*	A pointer to an application-registered function that, given an object assigned to the processor, returns the next object assigned to the processor before load balancing. Used with *Get_First_Obj* as an iterator over all objects. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Next_Obj_Fn**.
void **Get_Next_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Next_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Next_Obj_Fn**.
ZOLTAN_NUM_BORDER_OBJ_FN * *Get_Num_Border_Obj*	A pointer to an application-registered function that returns the number of objects sharing a subdomain border with a given processor. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Border_Obj_Fn**.
void **Get_Num_Border_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Num_Border_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Border_Obj_Fn**.
ZOLTAN_BORDER_OBJ_LIST_FN * *Get_Border_Obj_List*	A pointer to an application-registered function that returns arrays of objects that share a subdomain border with a given processor. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Border_Obj_List_Fn**.
void **Get_Border_Obj_List_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Border_Obj_List*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Border_Obj_List_Fn**.
ZOLTAN_FIRST_BORDER_OBJ_FN * *Get_First_Border_Obj*	A pointer to an application-registered function that returns the first object sharing a subdomain border with a given processor. Used with *Get_Next_Border_Obj* as an iterator over objects along borders. Set in **Zoltan_Set_Fn** or **Zoltan_Set_First_Border_Obj_Fn**.
void **Get_First_Border_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_First_Border_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_First_Border_Obj_Fn**.
ZOLTAN_NEXT_BORDER_OBJ_FN * *Get_Next_Border_Obj*	A pointer to an application-registered function that, given an object, returns the next object sharing a subdomain border with a given processor. Used with *Get_First_Border_Obj* as an iterator over objects along borders. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Next_Border_Obj_Fn**.
void **Get_Next_Border_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Next_Border_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Next_Border_Obj_Fn**.
ZOLTAN_NUM_COARSE_OBJ_FN * *Get_Num_Coarse_Obj*	A pointer to an application-registered function that returns the number of objects in the initial coarse grid. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Coarse_Obj_Fn**.
void **Get_Num_Coarse_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Num_Coarse_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Coarse_Obj_Fn**.
ZOLTAN_COARSE_OBJ_LIST_FN * *Get_Coarse_Obj_List*	A pointer to an application-registered function that returns arrays of objects in the initial coarse grid. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Coarse_Obj_List_Fn**.

void *Get_Coarse_Obj_List_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Coarse_Obj_List*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Coarse_Obj_List_Fn**.
ZOLTAN_FIRST_COARSE_OBJ_FN * *Get_First_Coarse_Obj*	A pointer to an application-registered function that returns the first object of the initial coarse grid. Used with *Get_Next_Coarse_Obj* as an iterator over all objects in the coarse grid. Set in **Zoltan_Set_Fn** or **Zoltan_Set_First_Coarse_Obj_Fn**.
void *Get_First_Coarse_Obj_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_First_Coarse_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_First_Coarse_Obj_Fn**.
ZOLTAN_NEXT_COARSE_OBJ_FN * *Get_Next_Coarse_Obj*	A pointer to an application-registered function that, given an object in the initial coarse grid, returns the next object in the coarse grid. Used with *Get_First_Coarse_Obj* as an iterator over all objects in the coarse grid. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Next_Coarse_Obj_Fn**.
void *Get_Next_Coarse_Obj_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Next_Coarse_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Next_Coarse_Obj_Fn**.
ZOLTAN_NUM_CHILD_FN * *Get_Num_Child*	A pointer to an application-registered function that returns the number of refinement children of an object. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Child_Fn**.
void *Get_Num_Child_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Num_Child*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Num_Child_Fn**.
ZOLTAN_CHILD_LIST_FN * *Get_Child_List*	A pointer to an application-registered function that returns arrays of objects that are refinement children of a given object. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Child_List_Fn**.
void *Get_Child_List_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Child_List*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Child_List_Fn**.
ZOLTAN_CHILD_WEIGHT_FN * *Get_Child_Weight*	A pointer to an application-registered function that returns the weight of an object. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Child_Weight_Fn**.
void *Get_Child_Weight_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Child_Weight*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Child_Weight_Fn**.
ZOLTAN_OBJ_SIZE_FN * *Get_Obj_Size*	A pointer to an application-registered function that returns the size (in bytes) of data objects to be migrated. Called by **Zoltan_Migrate**. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Obj_Size_Fn**.
void *Get_Obj_Size_Data	A pointer to data provided by the user that will be passed to the function pointed to by *Get_Obj_Size*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Obj_Size_Fn**.

ZOLTAN_PACK_OBJ_FN * *Pack_Obj*	A pointer to an application-registered function that packs all data for a given object into a communication buffer provided by the migration tools in preparation for data-migration communication. Called by **Zoltan_Migrate** for each object to be exported. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Pack_Obj_Fn**.
void **Pack_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Pack_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Pack_Obj_Fn**.
ZOLTAN_UNPACK_OBJ_FN * *Unpack_Obj*	A pointer to an application-registered function that unpacks all data for a given object from a communication buffer after the communication for data migration is completed. Called by **Zoltan_Migrate** for each imported object. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Unpack_Obj_Fn**.
void **Unpack_Obj_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Unpack_Obj*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Unpack_Obj_Fn**.
ZOLTAN_LB *LB*	A structure with data used by the load-balancing tools. See the table below.
ZOLTAN_MIGRATE *Migrate*	A structure with data used by the migration tools. See the table below.

Fields of the **Zoltan_Struct** *data structure.*

Each **Zoltan_Struct** data structure has a **ZOLTAN_LB** sub-structure. The **ZOLTAN_LB** structure contains data used by the load-balancing tools, including pointers to specific load-balancing methods and load-balancing data structures. The fields of the **ZOLTAN_LB** structure are listed and described in in the following table.

Fields of ZOLTAN_LB	Description
void * *Data_Structure*	The data structure used by the selected load-balancing algorithm; this pointer is cast by the algorithm to the appropriate data type.
double *Imbalance_Tol*	The degree of load balance which is considered acceptable. Set via a call to **Zoltan_Set_Param** for IMBALANCE_TOL.
int *Num_Global_Parts*	The total number of partitions to be generated. Set via a call to **Zoltan_Set_Param** for NUM_GLOBAL_PARTITIONS or through summation of NUM_LOCAL_PARTITIONS parameters.
int *Num_Local_Parts*	The number of partitions to be generated on this processor. Set via a call to **Zoltan_Set_Param** for NUM_LOCAL_PARTITIONS or (roughly) through division of the NUM_GLOBAL_PARTITIONS parameter by the number of processors.
int *Return_Lists*	A flag indicating whether the application wants import and/or export lists returned by **Zoltan_LB_Partition**. Set via a call to **Zoltan_Set_Param** for RETURN_LISTS.
ZOLTAN_LB_METHOD *Method*	An enumerated type designating which load-balancing algorithm should be used with this Zoltan structure; set via a call to **Zoltan_Set_Param** for LB_METHOD.
LB_FN * *LB_Fn*	A pointer to the load-balancing function specified by *Method*.

ZOLTAN_LB_FREE_DATA_FN *Free_Structure*	Pointer to a function that frees the Data_Structure memory.
ZOLTAN_LB_POINT_ASSIGN_FN *Point_Assign*	Pointer to the function that performs Zoltan_LB_Point_Assign for the particular load-balancing method.
ZOLTAN_LB_BOX_ASSIGN_FN *Box_Assign*	Pointer to the function that performs Zoltan_LB_Box_Assign for the particular load-balancing method.

Fields of the **ZOLTAN_LB** *data structure.*

Each **Zoltan_Struct** data structure has a **ZOLTAN_MIGRATE** sub-structure. The **ZOLTAN_MIGRATE** structure contains data used by the migration tools, including pointers to pre- and post-processing routines. These pointers are set through the interface routine **Zoltan_Set_Fn** and are used in **Zoltan_Migrate**. The fields of the **ZOLTAN_MIGRATE** structure are listed and described in in the following table.

Fields of ZOLTAN_MIGRATE	Description
int *Auto_Migrate*	A flag indicating whether Zoltan should perform auto-migration for the application. If true, Zoltan calls **Zoltan_Migrate** to move objects to their new processors; if false, data migration is left to the user. Set in **Zoltan_Set_Param** for AUTO_MIGRATE.
ZOLTAN_PRE_MIGRATE_PP_FN * *Pre_Migrate_PP*	A pointer to an application-registered function that performs pre-processing for data migration. The function is called by **Zoltan_Migrate** before data migration is performed. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Pre_Migrate_PP_Fn**.
void **Pre_Migrate_PP_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Pre_Migrate_PP*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Pre_Migrate_PP_Fn**.
ZOLTAN_MID_MIGRATE_PP_FN * *Mid_Migrate_PP*	A pointer to an application-registered function that performs processing between the packing and unpacking operations in **Zoltan_Migrate**. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Mid_Migrate_PP_Fn**.
void **Mid_Migrate_PP_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Mid_Migrate_PP*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Mid_Migrate_PP_Fn**.
ZOLTAN_POST_MIGRATE_PP_FN **Post_Migrate_PP*	A pointer to an application-registered function that performs post-processing for data migration. The function is called by **Zoltan_Migrate** after data migration is performed. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Post_Migrate_PP_Fn**.
void **Post_Migrate_PP_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Post_Migrate_PP*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Post_Migrate_PP_Fn**.
ZOLTAN_PRE_MIGRATE_FN * *Pre_Migrate*	A pointer to an application-registered function that performs pre-processing for data migration. The function is called by **Zoltan_Help_Migrate** before data migration is performed. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Pre_Migrate_Fn**. Maintained for backward compatibility with Zoltan v1.3 interface.
void **Pre_Migrate_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Pre_Migrate*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Pre_Migrate_Fn**.

ZOLTAN_MID_MIGRATE_FN * *Mid_Migrate*	A pointer to an application-registered function that performs processing between the packing and unpacking operations in **Zoltan_Help_Migrate**. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Mid_Migrate_Fn**. Maintained for backward compatibility with Zoltan v1.3 interface.
void **Mid_Migrate_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Mid_Migrate*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Mid_Migrate_Fn**.
ZOLTAN_POST_MIGRATE_FN **Post_Migrate*	A pointer to an application-registered function that performs post-processing for data migration. The function is called by **Zoltan_Help_Migrate** after data migration is performed. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Post_Migrate_Fn**. Maintained for backward compatibility with Zoltan v1.3 interface.
void **Post_Migrate_Data*	A pointer to data provided by the user that will be passed to the function pointed to by *Post_Migrate*. Set in **Zoltan_Set_Fn** or **Zoltan_Set_Post_Migrate_Fn**.

Fields of the **ZOLTAN_MIGRATE** *data structure.*

For each pointer to an application registered function in the **Zoltan_Struct** and **ZOLTAN_MIGRATE** data structures there is also a pointer to a Fortran application registered function, of the form ZOLTAN_FUNCNAME_FORT_FN **Get_Funcname_Fort*. These are for use within the Fortran interface. The Zoltan routines should invoke the usual application registered function regardless of whether the Zoltan structure was created from C or Fortran.

[Table of Contents | Next: Services | Previous: ID Data Types]

Services

Within Zoltan, several services are provided to simplify development of new algorithms in the library. Each service consists of a routine or set of routines that is compiled directly into Zoltan. Use of these services makes debugging easier and provides a uniform look to the algorithms in the library. The services available are listed below.

[Table of Contents | Next: Parameter Setting Routines | Previous: Data Structures]

Zoltan Developer's Guide | *Next* | *Previous*

Parameter Setting Routines

Zoltan allows applications to change a number of parameter settings at runtime. This facility supports debugging by, for instance, allowing control over the type and quantity of output. It also allows users to modify some of the parameters that characterize the partitioning algorithms. The design of the parameter setting routines was driven by several considerations. First, we wanted to keep the user interface as simple as possible. Second, we wanted to allow different Zoltan structures to have different parameter settings associated with them. This second consideration precluded the use of C's static global variables (except in a few special places). The parameter routines described below allow developers to provide runtime access to any appropriate variables. In some cases, it is appropriate to allow developers to tinker with parameters that will never be documented for users.

Our solution to parameter setting is to have a single interface routine **Zoltan_Set_Param**. This function calls a set of more domain-specific parameter setting routines, each of which is responsible for a domain-specific set of parameters. Assuming there are no errors, the parameter name and new value are placed in a linked list of new parameters which is maintained by the Zoltan structure. When a partitioning method is invoked on a Zoltan structure, it scans through this linked list using the **Zoltan_Assign_Param_Vals** function, resetting parameter values that are appropriate to the method.

In addition to the method-specific parameters, Zoltan also has a set of so-called **key parameters**. These are normally stored in the Zoltan structure and may be accessed by any part of the Zoltan code (including all the methods). A list of the key parameters currently used in Zoltan can be found in the User's Guide.

The routines that control parameter setting are listed below. Note that these routines have been written to be as independent of Zoltan as possible. Only a few minor changes would be required to use these routines as a separate library.

Zoltan_Set_Param: User interface function that calls a set of method-specific routines.
Zoltan_Set_Param_Vec: Same as Zoltan_Set_Param, but for vector parameters.
Zoltan_Check_Param: Routine to check if parameter name and value are OK.
Zoltan_Bind_Param: Routine to associate a parameter name with a variable.
Zoltan_Bind_Param_Vec: Same as Zoltan_Bind_Param, but for vector parameters.
Zoltan_Assign_Param_Vals: Scans list of parameter names & values, setting relevant parameters accordingly.
Zoltan_Free_Params: Frees a parameter list.

See also: Adding new parameters in Zoltan.

---

int **Zoltan_Set_Param**(struct **Zoltan_Struct** *zz, char *param_name, char *new_val);

The **Zoltan_Set_Param** function is the user interface for parameter setting. Its principle purpose is to call a sequence of more domain-specific routines for setting domain-specific parameters (e.g., **Zoltan_RCB_Set_Param**). If you are adding algorithms to Zoltan, you must write one of these domain-specific parameter routines and modify **Zoltan_Set_Param** to call it. **Zoltan_RCB_Set_Param** can serve as a template for this task. The arguments to this routine are two strings *param_name* and *new_val*. The domain-specific routines return an integer value with the following meaning.

0 - The parameter name was found, and the value passed all error checks.
1 - The parameter name was not found among the parameters known by the domain-specific routine.
2 - The parameter name was found, but the value failed the error checking.
3 - Same as 0, but do not add parameter and value to linked list.

Other - More serious error; value is an error code.

If one of the domain-specific parameter routines returns with a 0, **Zoltan_Set_Param** adds the parameter and the value (both strings) to a linked list of such pairs that is pointed to by the *Params* field of the *zz* structure.

Arguments:

zz	The Zoltan structure whose parameter value is being modified.
param_name	A string containing the name of the parameter being modified. It is automatically converted to all upper-case letters.
new_val	The new value desired for the parameter, expressed as a string.

Returned Value:

int	Error code.

---

int **Zoltan_Set_Param_Vec**(struct **Zoltan_Struct** *zz, char *param_name, char *new_val, int index);

This routine works the same way as Zoltan_Set_Param, but is used for vector parameters. A vector parameter is a parameter that in addition to a name also has a set of indices, usually starting at 0. Each entry (component) may have a different value. This routine sets a single entry (component) of a vector parameter. If you want all entries (components) of a vector parameter to have the same value, set the parameter using Zoltan_Set_Param as if it were a scalar parameter.

---

int **Zoltan_Check_Param**(char *param_name, char *new_val, **PARAM_VARS** *params, **PARAM_UTYPE** *result, int *matched_index);

The **Zoltan_Check_Param** routine simplifies the task of writing your own domain-specific parameter setting function. **Zoltan_Check_Param** compares the *param_name* string against a list of strings that you provide, and if a match is found it extracts the new value from the *new_val* string. See **Zoltan_RCB_Set_Param** for an example of how to use this routine.

Arguments:

param_name	A capitalized string containing the name of the parameter being modified.
new_val	The new value desired for the parameter, expressed as a string.
params	The data structure (defined in *params/params_const.h*) describing the domain-specific parameters to be matched against. The data structure is an array of items, each of which consists of four fields. The first field is a string that is a capitalized name of a parameter. The second field is an address that is unused in **Zoltan_Check_Param**, but is used in **Zoltan_Assign_Param_Vals**. The third field is another capitalized string that indicates the type of the parameter from the first field. Currently supported types are "INT", "INTEGER", "FLOAT", "REAL", "DOUBLE", "LONG", "STRING" and "CHAR". It is easy to add additional types by simple modifications to **Zoltan_Check_Param** and **Zoltan_Assign_Param_Vals**. The fourth field is an integer that gives the dimension (length) of the parameter, if it is a vector parameter. Scalar parameters have dimension 0. The array is terminated by an item consisting of four NULL fields. See **Zoltan_RCB_Set_Param** for an example of how to set up this data structure.
result	Structure of information returned by **Zoltan_Check_Param** (defined in *params/params_const.h*). If *param_name* matches any of the parameter names from the first field of the *params* data structure, **Zoltan_Check_Param** attempts to decode the value in *new_val*. The type of the value is determined by the third field in the *params* data structure. If the value decodes properly, it is returned in *result*.
matched_index	If *param_name* matches, then *matched_index* returns the index into the *params* array that corresponds to the matched parameter name. The *matched_index* and *result* values allow the developer to check that values being assigned to parameters are valid.

Returned Value:

int	0 - *param_name* found in *params* data structure and *new_val* decodes OK. 1 - *param_name* not found in *params* data structure. 2 - *param_name* found in *params* data structure but *new_val* doesn't decode properly.

---

int **Zoltan_Bind_Param** (**PARAM_VARS** **params*, char **name*, void **var*);

This routine is used to associate the name of a parameter in the parameter array *params* with a variable pointed to by *var*.
Note that since the variable to be bound can be of an arbitrary type, the pointer should be cast to a void pointer. **Zoltan_Bind_Param** must be called before **Zoltan_Assign_Param_Vals**, where the actual assignment of values takes place.

Arguments:

params	The data structure describing the domain-specific parameters to be matched against. The data structure is an array of items, each of which consists of four fields. The first field is a string that is a capitalized name of a parameter. The second field is an address that is unused in **Zoltan_Check_Param**, but is used in **Zoltan_Assign_Param_Vals**. The third field is another capitalized string that indicates the type of the parameter from the first field. Currently supported types are "INT", "INTEGER", "FLOAT", "REAL", "DOUBLE", "LONG", "STRING" and "CHAR". It is easy to add additional types by simple modifications to **Zoltan_Check_Param** and **Zoltan_Assign_Param_Vals**. The fourth field is an integer that gives the dimension (length) of the parameter, if it is a vector parameter. Scalar parameters have dimension 0. The array is terminated by an item consisting of four NULL fields.
name	A capitalized string containing the name of the parameter being modified.
var	A pointer to the variable you wish to associate with the parameter name *name*. The pointer should be type cast to a void pointer. The user is responsible for ensuring that the pointer really points to a variable of appropriate type. A NULL pointer may be used to "unbind" a variable such that it will not be assigned a value upon future calls to **Zoltan_Assign_Param_Vals.**

Returned Value:

int	Error code.

---

int **Zoltan_Bind_Param_Vec**(**PARAM_VARS** **params*, char **name*, void **var*, int *dim*);

Same as Zoltan_Bind_Param, but for vector parameters. The additional parameter *dim* gives the dimension or length of the vector parameter.

---

int **Zoltan_Assign_Param_Vals**(**PARAM_LIST** **change_list*, **PARAM_VARS** **params*, int *debug_level*, int *my_proc*, int *debug_proc*);

This routine changes parameter values as specified by the list of names and new values which is associated with a Zoltan structure. To use this routine, parameter values should first be set to their defaults, and then **Zoltan_Assign_Param_Vals** should be called to alter the values as appropriate. See **Zoltan_RCB** for a template.

Arguments:

change_list	The linked list of parameter names and values which is constructed by **Zoltan_Set_Param** and is a field of an **Zoltan_Struct** data structure (defined in *params/param_const.h*).

params	The data structure (defined in *params/params_const.h*) describing the domain-specific parameters to be matched against. The data structure is an array of items, each of which consists of three fields. The first field is a string which is a capitalized name of a parameter. The second field is an address of the parameter which should be altered. The third field is another capitalized string which indicates the type of the parameter being altered. Currently supported types are "INT", "INTEGER", "FLOAT", "REAL", "DOUBLE", "LONG", "STRING" and "CHAR". It is easy to add additional types by simple modifications to **Zoltan_Check_Param** and **Zoltan_Assign_Param_Vals**. The array is terminated by an item consisting of three NULL fields.
debug_level	Zoltan debug level. (Normally this is *zz->Debug_Level*.)
my_proc	Processor number. (Normally this is *zz->Proc*.)
debug_proc	Processor number for debugging. (Normally this is *zz->Debug_Proc*.)

Returned Value:

int	Error code.

The last three input parameters may seem strange. They are present to support Zoltan's debugging features. If the parameter utility code is used outside of Zoltan, these parameters may be removed or simply set these input values to zero in the function call.

---

void **Zoltan_Free_Params** (**PARAM_LIST** ***param_list*);

This routine frees the parameters in the list pointed to by *param_list*.

Arguments:

param_list	A pointer to a list (array) of parameters to be freed. **PARAM_LIST** is defined in *params/param_const.h*.

[Table of Contents | Next: Parallel Computing Routines | Previous: Services]

Zoltan Developer's Guide | *Next* | *Previous*

Parallel Computing Routines

Parallel computing utilities are described in this section.

Zoltan_Print_Sync_Start / **Zoltan_Print_Sync_End**: provide synchronization of processors for I/O (with example).
Zoltan_Print_Stats : print statistics about a parallel variable.

---

void **Zoltan_Print_Sync_Start**(MPI_Comm *communicator*, int *do_print_line*);

The **Zoltan_Print_Sync_Start** function is adapted from work of John Shadid for the MPSalsa project at Sandia National Laboratories. With **Zoltan_Print_Sync_End**, it provides synchronization so that one processor in the Zoltan communicator can complete its I/O before the next processor begins its I/O. This synchronization utility is useful for debugging algorithms, as it allows the output from processors to be produced in an organized manner. It is, however, a serializing process, and thus, does not scale well to large number of processors.

Zoltan_Print_Sync_Start should called by each processor in the MPI communicator before the desired I/O is performed. **Zoltan_Print_Sync_End** is called by each processor after the I/O is performed. No communication can be performed between calls to **Zoltan_Print_Sync_Start** and **Zoltan_Print_Sync_End**. See the example below for usage of **Zoltan_Print_Sync_Start**.

Arguments:

communicator	The MPI communicator containing all processors to participate in the synchronization.
do_print_line	A flag indicating whether to print a line of "#" characters before and after the synchronization block. If *do_print_line* is TRUE, a line is printed; no line is printed otherwise.

---

void **Zoltan_Print_Sync_End**(MPI_Comm *communicator*, int *do_print_line*);

The **Zoltan_Print_Sync_End** function is adapted from work of John Shadid for the MPSalsa project at Sandia National Laboratories. With **Zoltan_Print_Sync_Start**, it provides synchronization so that one processor in the Zoltan communicator can complete its I/O before the next processor begins its I/O. This synchronization utility is useful for debugging algorithms, as it allows the output from processors to be produced in an organized manner. It is, however, a serializing process, and thus, does not scale well to large number of processors.

Zoltan_Print_Sync_Start should called by each processor in the MPI communicator before the desired I/O is performed. **Zoltan_Print_Sync_End** is called by each processor after the I/O is performed. No communication can be performed between calls to **Zoltan_Print_Sync_Start** and **Zoltan_Print_Sync_End**. See the example below for usage of **Zoltan_Print_Sync_End**.

Arguments:

communicator	The MPI communicator containing all processors to participate in the synchronization.

do_print_line	A flag indicating whether to print a line of "#" characters before and after the synchronization block. If *do_print_line* is TRUE, a line is printed; no line is printed otherwise.

---

void **Zoltan_Print_Stats**(MPI_Comm *comm*, int *debug_proc*, double *x*, char **msg*);

Zoltan_Print_Stats is a very simple routine that computes the maximum and sum of the variable *x* over all processors associated with the MPI communicator *comm*. It also computes and prints the imbalance of *x*, that is, the maximum value divided by the average minus one. If *x* has the same value on all processors, the imbalance is zero.

Arguments:

comm	The MPI Communicator to be used in maximum and sum operations.
debug_proc	The processor from which output should be printed.
x	The variable of which one wishes to display statistics.
msg	A string that typically describes the meaning of *x*.

Example Using Zoltan_Print_Sync_Start/Zoltan_Print_Sync_End

```
...
if (zz->Debug_Level >= ZOLTAN_DEBUG_ALL) {
    int i;
    Zoltan_Print_Sync_Start(zz->Communicator, TRUE);
    printf("Zoltan: Objects to be exported from Proc %d\n", zz->Proc);
    for (i = 0; i < *num_export_objs; i++) {
        printf("    Obj: ");
        ZOLTAN_PRINT_GID(zz, &((*export_global_ids)[i*zz->Num_GID]));
        printf(" Destination: %4d\n",
              (*export_procs)[i]);
    }
    Zoltan_Print_Sync_End(zz->Communicator, TRUE);
}
```

Example usage of Zoltan_Print_Sync_Start and Zoltan_Print_Sync_End to synchronize output among processors. (Taken from Zoltan_LB_Partition in lb/lb_balance.c.)

[Table of Contents | Next: Object List Function | Previous: Parameter Setting Routines]

Zoltan Developer's Guide | *Next* | *Previous*

Common Functions for Querying Applications

Many Zoltan algorithms need to query applications for similar data. The following functions provide simple, uniform query functionality for algorithm developers:

Zoltan_Get_Obj_List
Zoltan_Get_Coordinates

These functions provide a uniform method of calling the query functions registered by an application. Their use simplifies new algorithm development and code maintenance. Usage examples are in *rcb/shared.c*.

Zoltan_Get_Obj_List can be called from any Zoltan algorithm to obtain a list of object IDs, weights, and partition assignments.

Given a list of object IDs, **Zoltan_Get_Coordinates** can be called from any Zoltan algorithm to obtain a list of coordinates for those IDs.

Note that, contrary to most Zoltan functions, these functions allocate memory for their return lists.

---

int **Zoltan_Get_Obj_List**(
  struct **Zoltan_Struct** *zz,
  int *num_obj,
  ZOLTAN_ID_PTR *global_ids,
  ZOLTAN_ID_PTR *local_ids,
  int wdim,
  float **objwgts,
  int **parts);

Zoltan_Get_Obj_List returns arrays of global and local IDs, partition assignments, and object weights (if **OBJ_WEIGHT_DIM** is not zero) for all objects on a processor. It is a convenient function that frees algorithm developers from calling **ZOLTAN_OBJ_LIST_FN**, **ZOLTAN_FIRST_OBJ_FN**, **ZOLTAN_NEXT_OBJ_FN**, and **ZOLTAN_PARTITION_FN** query functions directly.

Arguments:

zz	A pointer to the Zoltan structure created by **Zoltan_Create**.
num_obj	Upon return, the number of objects.
global_ids	Upon return, an array of global IDs of objects on the current processor.
local_ids	Upon return, an array of local IDs of objects on the current processor. NULL is returned when NUM_LID_ENTRIES is zero.
wdim	The number of weights associated with an object (typically 1), or 0 if weights are not requested.
objwgts	Upon return, an array of object weights. Weights for object *i* are stored in *objwgts*[*i***wdim*:(*i*+1)**wdim*-1], for *i*=0,1,...,*num_obj*-1. If *wdim* is zero, the return value of objwgts is undefined and may be NULL.
parts	Upon return, an array of partition assignments. Object *i* is currently in partition *parts[i]*.

Returned value:

Error code.

Required Query Functions:

ZOLTAN_NUM_OBJ_FN

ZOLTAN_OBJ_LIST_FN or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair

Optional Query Functions:

ZOLTAN_PARTITION_FN

---

int **Zoltan_Get_Coordinates**(
  struct **Zoltan_Struct** *zz*,
  int *num_obj*,
  ZOLTAN_ID_PTR *global_ids*,
  ZOLTAN_ID_PTR *local_ids*,
  int **num_dim*,
  double ***coords*);

Given lists of object IDs, **Zoltan_Get_Coordinates** returns the dimensionality of the problem and an array of coordinates of the objects. It is a convenient function that frees algorithm developers from calling **ZOLTAN_NUM_GEOM_FN**, **ZOLTAN_GEOM_MULTI_FN**, and **ZOLTAN_GEOM_FN** query functions directly.

Arguments:

zz	A pointer to the Zoltan structure created by **Zoltan_Create**.
num_obj	The number of objects.
global_ids	An array of global IDs of objects on the current processor.
local_ids	An array of local IDs of objects on the current processor. *local_ids* is NULL when NUM_LID_ENTRIES is zero.
num_dim	Upon return, the number of coordinates for each object (typically 1, 2 or 3).
coords	Upon return, an array of coordinates for the objects. Coordinates for object *i* are stored in *coords*[*i***num_dim*:(*i*+1)**num_dim*-1], for *i*=0,1,...,*num_obj*-1.

Returned value:

Error code.

Required Query Functions:

ZOLTAN_NUM_GEOM_FN

ZOLTAN_GEOM_MULTI_FN or **ZOLTAN_GEOM_FN**

[Table of Contents | Next: Hash Function | Previous: Parallel Routines]

Zoltan Developer's Guide | *Next* | *Previous*

Hash Function

Zoltan provides a hash function for global and local IDs. The hash function computes a non-negative integer value in a certain range from an ID.

Zoltan_Hash : hash a global or local ID into non-negative integers

---

---

unsigned int **Zoltan_Hash**(**ZOLTAN_ID_PTR** *key*, int *num_id_entries*, unsigned int *n*);

Zoltan_Hash computes a hash value for a global or local ID. Note that this hash function has been optimized for 32-bit integer systems, but should work on any machine. The current implementation uses a simple multiplicative hash function based on Don Knuth's golden ratio method; see *The Art of Computer Programming*, vol. 3.

Arguments:

key	A pointer to the ID to be hashed.
num_id_entries	The length of the ID (as given by **NUM_GID_ENTRIES** or **NUM_LID_ENTRIES**).
n	The computed hash value will be between 0 and *n*-1.

Return Value:

unsigned int	The hash value (between 0 and *n*-1).

[Table of Contents | Next: Timing Routines | Previous: Object List Function]

Zoltan Developer's Guide | *Next* | *Previous*

Timing Routines

To assist in performance measurements and profiling, several timing routines are included in the Zoltan library. The main timer function, **Zoltan_Time**, provides access to at least two portable timers: one CPU clock and one wall clock. On most systems, user time can also be measured. A higher-level timing capability built using **Zoltan_Time** is also available; see **ZOLTAN_TIMER** for more details.

The routines included in the utility are listed below.

Zoltan_Time: Returns the time (in seconds) after some fixed reference point in time.
Zoltan_Time_Resolution: The resolution of the specified timer.

Currently, the following timers are supported:

- *ZOLTAN_TIME_WALL* : wall-clock time.

  On most systems, this timer calls MPI_Wtime.
- *ZOLTAN_TIME_CPU* : cpu time.

  On most systems, this timer calls the ANSI C function clock(). Note that this timer may roll over at just 71 minutes. **Zoltan_Time**attempts to keep track of the number of roll-overs but this feature will work only if **Zoltan_Time**is called at least once during every period between roll-overs.
- *ZOLTAN_TIME_USER* : user time.

  On most systems, this timer calls times(). Note that times() is required by POSIX and is widely available, but it is not required by ANSI C so may be unavailable on some systems. Compile Zoltan with -DNO_TIMES in this case.

Within Zoltan, it is recommended to select which timer to use by setting the TIMER general parameter via **Zoltan_Set_Param**. The default value of TIMER is *wall*. Zoltan stores an integer representation of the selected timing method in *zz->Timer*. This value should be passed to **Zoltan_Time**, as in **Zoltan_Time**(*zz->Timer*).

---

double **Zoltan_Time**(int *timer*);

Zoltan_Time returns the time in seconds, measured from some fixed reference time. Note that the time is *not* synchronized among different processors or processes. The time may be either CPU time or wall-clock time. The timer is selected through **Zoltan_Set_Param**.

Arguments:

timer	The timer type (e.g., wall or cpu) represented as an integer. See top of page for a list of valid values.

Returned Value:

double	The time in seconds. The time is always positive; a negative value indicates an error.

---

double **Zoltan_Time_Resolution**(int *timer*) ;

Zoltan_Time_Resolution returns the resolution of the current timer. The returned resolution is a lower bound on the actual resolution.

Arguments:

timer	The timer type (e.g., wall or cpu) represented as an integer. See top of page for a list of valid values.

Returned Value:

double	The timer resolution in seconds. If the resolution is unknown, -1 is returned.

Example:
Here is a simple example for how to use the timer routines:

```
double t0, t1, t2;
Zoltan_Set_Param(zz, "TIMER", "wall");
t0 = Zoltan_Time(zz->Timer);
/* code segment 1  */
t1 = Zoltan_Time(zz->Timer);
/* code segment 2  */
t2 = Zoltan_Time(zz->Timer);
/* Print timing results */
Zoltan_Print_Stats(zz->Communicator, zz->Debug_Proc, t1-t0, "Time for part 1:");
Zoltan_Print_Stats(zz->Communicator, zz->Debug_Proc, t2-t1, "Time for part 2:");
Zoltan_Print_Stats(zz->Communicator, zz->Debug_Proc, t2-t0, "Total time :");
```

[Table of Contents | Next: Debugging Services | Previous: Hash Function]

Zoltan Developer's Guide | *Next* | *Previous*

High-Level Timing Services: ZOLTAN_TIMER

The ZOLTAN_TIMER provides high-level capability for instrumenting code with timers and reporting the execution times measured. The ZOLTAN_TIMER can store many separate timers within a single ZOLTAN_TIMER object and associate a name with each timer for ease of reporting. It can perform parallel synchronization among processors using a time to ensure that all time is attributed to the correct section of the timed code. Its output contains the maximum, minimum and average times over sets of processors.

The ZOLTAN_TIMER uses **Zoltan_Time** to obtain the current clock time.

NOTE: The current implementation of ZOLTAN_TIMER relies on two assumptions to work correctly.

1. ZOLTAN_TIMER assumes that individual timers within a single ZOLTAN_TIMER object are initialized in the same order on all processors. Times over multiple processors are accrued based on the value of the *timer_idx* returned by **Zoltan_Timer_Init**, so for accurate timing, each processor should associate the same value of *timer_idx* with the same section of code.

2. ZOLTAN_TIMER uses synchronization in **Zoltan_Timer_Print** and **Zoltan_Timer_PrintAll**, and optionally in **ZOLTAN_TIMER_START** and **ZOLTAN_TIMER_STOP**. For these routines to work properly, then, all processors must call them at the same place in the code to satisfy the synchronization. A possible workaround is to provide MPI_COMM_SELF as an argument to these functions for single-processor timing.

In future work, these constraints may be weakened so that, for instance, different processors may have different numbers of timers or skip synchronization points.

Source code location:	*Utilities/Timer*
Function prototypes file:	*Utilities/Timer/zoltan_timer.h* or *include/zoltan_timer.h Utilities/Timer/zoltan_timer_cpp.h* or *include/zoltan_timer_cpp.h*
Library name:	libzoltan_timer.a
Other libraries used by this library:	libmpi.a and libzoltan_mem.a.
Routines:	

Zoltan_Timer_Create: Creates a ZOLTAN_TIMER object to store timers.
Zoltan_Timer_Init: Initializes a new timer.
ZOLTAN_TIMER_START: Starts a single timer.
ZOLTAN_TIMER_STOP: Stops a single timer.
Zoltan_Timer_Print: Prints the values of a single timer.
Zoltan_Timer_PrintAll: Prints the values of all timers.
Zoltan_Timer_Reset: Resets a single timer.
Zoltan_Timer_Copy: Copies a ZOLTAN_TIMER object to newly allocated memory.
Zoltan_Timer_Copy_To: Copies a ZOLTAN_TIMER object to existing memory.
Zoltan_Timer_Destroy: Frees a ZOLTAN_TIMER object.

Use in Zoltan:

The ZOLTAN_TIMER utilities are used in Zoltan's graph and hypergraph algorithms. It is activated by setting parameter *use_timers* to a positive integer value.

C:	struct Zoltan_Timer ***Zoltan_Timer_Create**(int *timer_flag*);
C++:	**Zoltan_Timer_Object(int *timer_flag*);**

Zoltan_Timer_Create allocates memory for storing information to be used by the Zoltan_Timer. The pointer returned by this function is passed to many subsequent functions. An application or Zoltan itself may allocate more than one Zoltan_Timer data structure; for example, a separate Zoltan_Timer may be used in different partitioning algorithms or in different routines.

In the C++ interface, the Zoltan_Timer_Object class represents a Zoltan_Timer data structure and the functions that operate on it. It is the constructor that allocates an instance of a Zoltan_Timer_Object. It has no return value.

Input Arguments:

timer_flag	A flag indicating the type of timer to be used; it is passed to calls to **Zoltan_Time**. Valid values are *ZOLTAN_TIME_WALL*, *ZOLTAN_TIME_CPU*, *ZOLTAN_TIME_USER* and *ZOLTAN_TIME_DEF* (the default timer). See the timing routines for more information about the timer_flag values.

Returned Value:

struct Zoltan_Timer *	Pointer to memory for storage of Zoltan_Timer information. If an error occurs, NULL will be returned in C.

C:	int **Zoltan_Timer_Init**(struct Zoltan_Timer **zt*, int *use_barrier*, const char **timer_name*);
C++:	int **Zoltan_Timer_Object::Init**(const int *use_barrier*, const std::string & timer_name);

Zoltan_Timer_Init initializes a single timer within a Zoltan_Timer object. Each timer in the Zoltan_Timer object is assigned a unique integer, which is returned by **Zoltan_Timer_Init** and is later used to indicate which timer to start or stop. It is also used to accrue times across multiple processors. **Zoltan_Timer_Init** may be called several times with the same Zoltan_Timer object to create many different times within the object.

Note that processors must initialize multiple timers within a Zoltan_Timer object in the same order to ensure that the returned timer index value is the same on each processor.

Input Arguments:

zt	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**.
use_barrier	Flag indicating whether to synchronize processors before starting or stopping a timer. A value of 1 causes MPI_Barrier to be invoked before the timer is started or stopped; a value of 0 provides no synchronization.
timer_name	A character string associated with the timer; it is printed as the timer name in **Zoltan_Timer_Print** and **Zoltan_Timer_PrintAll**.

Returned Value:

int	The unique integer identifier for this timer.

C:	int **ZOLTAN_TIMER_START**(struct Zoltan_Timer *_zt_, int _timer_idx_, MPI_COMM _communicator_);
C++:	int **Zoltan_Timer_Object::Start**(const int _timer_idx_, const MPI_COMM & communicator);

ZOLTAN_TIMER_START starts the timer with index *timer_idx* associated with the Zoltan_Timer struct *zt*. Error checking ensures that the timer is not already running before it is started. If the timer timer_idx was initialized with *use_barrier*=1, all processors should start the timer at the same point in the code.

Input Arguments:

zt	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**.
timer_idx	The integer timer index (returned by **Zoltan_Timer_Init**) of the timer to be started.
communicator	The MPI communicator to be used for synchronization is the timer was initialized with *use_barrier*=1.

Returned Value:

int	Error code indicating whether the timer started successfully.

C:	int **ZOLTAN_TIMER_STOP**(struct Zoltan_Timer *_zt_, int _timer_idx_, MPI_COMM _communicator_);
C++:	int **Zoltan_Timer_Object::Stop**(const int _timer_idx_, const MPI_COMM & communicator);

ZOLTAN_TIMER_STOP stops the timer with index *timer_idx* associated with the Zoltan_Timer struct *zt*. Error checking ensures that the timer is already running before it is stopped. If the timer timer_idx was initialized with *use_barrier*=1, all processors should stop the timer at the same point in the code.

Input Arguments:

zt	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**.
timer_idx	The integer timer index (returned by **Zoltan_Timer_Init**) of the timer to be stopped.
communicator	The MPI communicator to be used for synchronization is the timer was initialized with *use_barrier*=1.

Returned Value:

int	Error code indicating whether the timer stopped successfully.

C:	int **Zoltan_Timer_Print**(struct Zoltan_Timer *_zt_, int _timer_idx_, int _proc_, MPI_Comm _comm_, FILE *_fp_);
C++:	int **Zoltan_Timer_Object::Print**(const int _timer_idx_, const int _proc_, const MPI_Comm &_comm_, FILE *_fp_);

Zoltan_Timer_Print accrues accumulated times for a single timer *timer_idx* across a communicator and computes the minimum, maximum and average values across all processors of the MPI communicator *comm*. These values, as well as the timer index *timer_idx* and timer name, are then printed by processor *proc*. Because of the synchronization needed to compute the minimum, maximum and average values, **Zoltan_Timer_Print** must be called by all processors in the communicator *comm*. Communicator MPI_COMM_SELF can be used to print a single processor's timer values.

Input Arguments:

zt	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**.
timer_idx	The integer timer index (returned by **Zoltan_Timer_Init**) of the timer to be printed.
proc	The rank (within MPI communicator *comm*) of the processor that should print the timer's values.
comm	The MPI communicator across which minimum, maximum and average values of the timer should be computed.

fp	The file pointer to a open, writable file to which timer values should be printed. Special files *stdout* and *stderr* are also legal values for this argument.

Returned Value:

int	Error code.

C:	int **Zoltan_Timer_PrintAll**(struct Zoltan_Timer *zt, int *proc*, MPI_Comm *comm*, FILE **fp*);
C++:	int **Zoltan_Timer_Object::PrintAll**(const int *proc*, const MPI_Comm &*comm*, FILE **fp*);

Zoltan_Timer_PrintAll accrues accumulated times for all timers in *zt* across a communicator and computes the minimum, maximum and average values across all processors of the MPI communicator *comm*. The timer values for each timer, as well as its timer index and timer name, are then printed by processor *proc*. Because of the synchronization needed to compute the minimum, maximum and average values, **Zoltan_Timer_PrintAll** must be called by all processors in the communicator *comm*. Communicator MPI_COMM_SELF can be used to print a single processor's timer values.

Input Arguments:

zt	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**.
proc	The rank (within MPI communicator *comm*) of the processor that should print the timer's values.
comm	The MPI communicator across which minimum, maximum and average values of the timer should be computed.
fp	The file pointer to a open, writable file to which timer values should be printed. Special files *stdout* and *stderr* are also legal values for this argument.

Returned Value:

int	Error code.

C:	int **Zoltan_Timer_Reset**(struct Zoltan_Timer *zt, int *timer_idx*, int *use_barrier*, const char **timer_name*);
C++:	int **Zoltan_Timer_Object::Reset**(const int *timer_idx*, const int *use_barrier*, const std::string & timer_name);

Zoltan_Timer_Reset resets the single timer represented by *timer_idx* within a Zoltan_Timer object. The accumulated time within the timer is reset to zero. The timer's name *timer_name* and the flag *use_barrier* indicating whether the timer should be started and stopped synchronously across processors may be changed as well.

Input Arguments:

zt	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**.
timer_idx	The integer timer index (returned by **Zoltan_Timer_Init**) of the timer to be reset.
use_barrier	Flag indicating whether to synchronize processors before starting or stopping a timer. A value of 1 causes MPI_Barrier to be invoked before the timer is started or stopped; a value of 0 provides no synchronization.
timer_name	A character string associated with the timer; it is printed as the timer name in **Zoltan_Timer_Print** and **Zoltan_Timer_PrintAll**.

Returned Value:

int	Error code indicating whether or not the timer was reset correctly.

C:	struct Zoltan_Timer ***Zoltan_Timer_Copy**(struct Zoltan_Timer **from*);
C++:	**Zoltan_Timer_Object**(const Zoltan_Timer_Object &*from*);

Zoltan_Timer_Copy creates a new ZOLTAN_TIMER object and copies the state of the existing ZOLTAN_TIMER object *from* to the new object. It returns the new ZOLTAN_TIMER object.

In C++, there is no direct interface to **Zoltan_Timer_Copy**. Instead, the Zoltan_Timer_Object copy constructor invokes the C library function **Zoltan_Timer_Copy**.

Input Arguments:

from	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create** whose state is to be copied to new memory.

Returned Value:

struct Zoltan_Timer *	Pointer to memory for storage of the copied Zoltan_Timer information. If an error occurs, NULL will be returned in C.

C:	int **Zoltan_Timer_Copy_To**(struct Zoltan_Timer ***to*, struct Zoltan_Timer **from*);
C++:	**Zoltan_Timer_Object** & operator =(const Zoltan_Timer_Object &*from*);

Zoltan_Timer_Copy_To copies one ZOLTAN_TIMER object to another, after first freeing any memory used by the targe ZOLTAN_TIMER object and re-initilizing it.

The C++ interface to **Zoltan_Timer_Copy_To** is through the Zoltan_Timer_Object copy operator which invokes the C library function **Zoltan_Timer_Copy_To**.

Arguments:

to	Pointer to the Zoltan_Timer struct whose state is to be overwritten with the state of *from*.
from	Pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create** whose state is to be copied to *to*.

Returned Value:

int	Error code.

C:	void **Zoltan_Timer_Destroy**(struct Zoltan_Timer ***zt*);
C++:	**~Zoltan_Timer_Object();**

Zoltan_Timer_Destroy frees memory allocated by **Zoltan_Timer_Create** and in C, sets the timer pointer *zt* to NULL. **Zoltan_Timer_Destroy** should be called when an application is finished using a timer object.

In C++, the Zoltan_Timer_Object class represents a Zoltan_Timer data structure and the functions that operate on it. **Zoltan_Timer_Destroy** is called by the destructor for the Zoltan_Timer_Object.

Input Arguments:

zt	Pointer to the pointer to the Zoltan_Timer struct returned by **Zoltan_Timer_Create**. Upon return, zt is set to NULL.

Returned Value:

None.

[Table of Contents | Next: Debugging Services | Previous: Timing Routines]

Zoltan Developer's Guide | *Next* | *Previous*

Debugging Services

Execution of code for debugging can be controlled by algorithm specific parameters or by the Zoltan key parameter DEBUG_LEVEL. The value of the *Debug_Level* field of the **Zoltan_Struct** structure can be tested to determine whether the user desires debugging information. Several constants (**ZOLTAN_DEBUG_***) are defined in *zz/zz_const.h*; the *Debug_Level* field should be compared to these values so that future changes to the debugging levels can be made easily. An example is included below.

Several macros for common debugging operations are provided. The macros can be used to generate function trace information, such as when control enters or exits a function or reaches a certain point in the execution of a function.

ZOLTAN_TRACE_ENTER
ZOLTAN_TRACE_EXIT
ZOLTAN_TRACE_DETAIL

These macros produce output depending upon the value of the DEBUG_LEVEL parameter set in Zoltan by a call to **Zoltan_Set_Param**. The macros are defined in *zz/zz_const.h.*

Examples of the use of these macros can be found below and in *lb/lb_balance.c* and *rcb/rcb.c.*

ZOLTAN_TRACE_ENTER(struct **Zoltan_Struct** **zz*, char **function_name*);

ZOLTAN_TRACE_ENTER prints to *stdout* a message stating that a given processor is entering a function. The call to the macro should be included at the beginning of major functions for which debugging information is desired. Output includes the processor number and the function name passed as an argument to the macro. The amount of output produced is controlled by the value of the DEBUG_LEVEL parameter set in Zoltan by a call to **Zoltan_Set_Param**.

Arguments:

zz	Pointer to a Zoltan structure.
function_name	Character string containing the function's name.

Output:

ZOLTAN (Processor #) Entering *function_name*

ZOLTAN_TRACE_EXIT(struct **Zoltan_Struct** **zz*, char **function_name*);

ZOLTAN_TRACE_EXIT prints to *stdout* a message stating that a given processor is exiting a function. The call to the macro should be included at the end of major functions (and before return statements) for which debugging information is desired. Output includes the processor number and the function name passed as an argument to the macro. The amount of output produced is controlled by the value of the DEBUG_LEVEL parameter set in Zoltan by a call to **Zoltan_Set_Param**.

Arguments:

zz	Pointer to a Zoltan structure.

function_name	Character string containing the function's name.

Output:

ZOLTAN (Processor #) Leaving *function_name*

ZOLTAN_TRACE_DETAIL(struct **Zoltan_Struct** *zz, char *function_name, char *message);

ZOLTAN_TRACE_DETAIL prints to *stdout* a message specified by the developer. It can be used to indicate how far execution has progressed through a routine. It can also be used to print values of variables. See the example below. Output includes the processor number, the function name passed as an argument to the macro, and a user-defined message passed to the macro. The amount of output produced is controlled by the value of the DEBUG_LEVEL parameter set in Zoltan by a call to **Zoltan_Set_Param**.

Arguments:

zz	Pointer to a Zoltan structure.
function_name	Character string containing the function's name.
message	Character string containing a message defined by the developer.

Output:

ZOLTAN (Processor #) *function_name: message*

Example:
An example using the debugging macros in shown below.

```
#include "zoltan.h"
void example(struct Zoltan_Struct *zz)
{
char *yo = "example";
char tmp[80];
int a, b;

       ZOLTAN_TRACE_ENTER(zz, yo);
       a = function_one(zz);
       ZOLTAN_TRACE_DETAIL(zz, yo, "After function_one");
       b = function_two(zz);
       sprintf(tmp, "b = %d a = %d", b, a);
       ZOLTAN_TRACE_DETAIL(zz, yo, tmp);
       if (zz->Debug_Level >= ZOLTAN_DEBUG_ALL)
         printf("Total = %d\n", a+b);
       ZOLTAN_TRACE_EXIT(zz, yo);
}
```

[Table of Contents | Next: Adding New Load-Balancing Algorithms | Previous: ZOLTAN_TIMER]

Adding New Load-Balancing Algorithms to Zoltan

The Zoltan library is designed so that adding new load-balancing algorithms to the library is simple. In many cases, existing code can be easily modified to use the interface query functions to build the data structures needed for the algorithm. The process for adding new algorithms to the library is described below; more detail is provided at each link.

1. Make sure you follow the Philosophy of Zoltan and the Coding Principles in Zoltan.
2. Use the Data Structures provided by Zoltan.
3. Implement a Load-Balancing Function front-end to the algorithm. Note that Zoltan load-balance methods should assign objects both to processors and partitions, which may be different. The recommended strategy is to assign objects to partitions first, then use Zoltan_LB_Part_To_Proc to generate the corresponding processors.
4. Add the algorithm to the Load-Balancing Interface Routines.
5. Add the Parameters needed by the algorithm. Also make sure that the algorithm uses the General Parameters in Zoltan properly, in particular **Imbalance_Tol** and **Debug_Level.**
6. If necessary, write a routine to free your dynamically allocated data structures. See tips on memory management in Zoltan.
7. If your algorithm uses persistent data structures, like the RCB tree with KEEP_CUTS, write a routine to copy your load balancing data structure.
8. We recommend you add partition remapping to your algorithm using **Zoltan_LB_Remap**.
9. Update the Fortran and C++ interfaces, if necessary.
10. Document your new method. The documentation should be written in a format that can easily be converted into HTML and PDF. Consider adding a simple application to the *examples* directory demonstrating the use of your method.
11. Please contact the Zoltan team if you would like your method to be distributed with future versions of Zoltan.

[Table of Contents | Next: Load-Balancing Interface Routines | Previous: Debugging Services]

Load-Balancing Interface Routines

Any new method that you wish to add to the Zoltan library must have an interface that conforms to the prototype **LB_FN**. Note that the load balancing function may return either import lists, export lists, or both. All processes must return the same type of list. If import (export) lists are not computed, then the variable *num_import* (*num_export*) must be set to a negative number (typically -1) upon return. Full support of the RETURN_LISTS parameter is not required. If RETURN_LISTS is not set to NONE, the new algorithm may return either import or export lists; the Zoltan interface will then build the lists requested by RETURN_LISTS.

A new algorithm must be added to the load-balancing interface for use with parameter LB_METHOD. An entry for the new algorithm must be added to the enumerated type **Zoltan_LB_Method** in *lb/lb_const.h*. An external **LB_FN** prototype for the load-balancing function must also be added to *lb/lb_const.h*; see the prototype for function *Zoltan_RCB* as an example. A character string describing the new algorithm should be chosen to be used as the parameter value for LB_METHOD. In function **Zoltan_LB_Set_LB_Method**, a test for this string should be added and the *Method* and *LB_Fn* fields of the **Zoltan_Struct** should be set to the new enumerated type value and new load-balancing function pointer.

[Table of Contents | Next: Load-Balancing Function Implementation | Previous: Adding New Algorithms]

Zoltan Developer's Guide | *Next* | *Previous*

Load-Balancing Function Implementation

The new load-balancing algorithm should be implemented as an **ZOLTAN_LB_FN**. The type definition for an **ZOLTAN_LB_FN** is in *lb/lb_const.h* and is described below. When the new algorithm is selected, the *LB_Fn* field of the **Zoltan_Struct** is set to point to the **ZOLTAN_LB_FN** function for the new algorithm. This pointer is then used in invoking load balancing in **Zoltan_LB_Partition**.

---

typedef int **ZOLTAN_LB_FN** (struct **Zoltan_Struct** **zz*, float **part_sizes*, int **num_import*, **ZOLTAN_ID_PTR** **import_global_ids*, **ZOLTAN_ID_PTR** **import_local_ids*, int ***import_procs*, int ***import_to_parts*, int **num_export*, **ZOLTAN_ID_PTR** **export_global_ids*, **ZOLTAN_ID_PTR** **export_local_ids*, int ***export_procs*, int ***export_to_parts*);

The **ZOLTAN_LB_FN** function type describes the arguments passed to a load-balancing function. The input to the function is a **Zoltan_Struct** containing pointers to application-registered functions to be used in the load-balancing algorithm. The remaining arguments are output parameters listing the objects to be imported or exported to the processor in the new decomposition. The arrays for global and local IDs and source processors must be allocated by the load-balancing function. The load-balancing function may return either the import arrays, the export arrays, or both. If no import data is returned, **num_import* must be set to a negative number, and similarly with **num_export*. Full support of the RETURN_LISTS parameter is not required. If RETURN_LISTS is not set to NONE, the new algorithm may return either import or export lists; the Zoltan interface will then build the lists requested by RETURN_LISTS.

Arguments:

zz	A pointer to the **Zoltan_Struct** to be used in the load-balancing algorithm.
part_sizes	Input: an array of partition sizes for each weight component. Entry *part_sizes[i*obj_weight_dim+j]* contains the user-requested partition size for partition *i* with respect to object weight *j* for *i*=0,1,...,number of partitions-1, and *j*=0,1,...,*obj_weight_dim*-1. If the application sets parameter *OBJ_WEIGHT_DIM*, *obj_weight_dim* is the set value of *OBJ_WEIGHT_DIM*; otherwise, *obj_weight_dim* is one.
num_import	Upon return, the number of objects to be imported to the processor for the new decomposition. A negative number indicates that no import data has been computed and the import arrays should be ignored.
import_global_ids	Upon return, an array of *num_import* global IDs of objects to be imported to the processor for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.
import_local_ids	Upon return, an array of *num_import* local IDs of objects to be imported to the processor for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.
import_procs	Upon return, an array of size *num_import* containing the processor IDs of processors owning (in the old decomposition) the objects to be imported for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.
import_to_parts	Upon return, an array of size *num_import* containing the partition IDs of partitions to which objects will be imported *in the NEW decomposition*. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.
num_export	Upon return, the number of objects to be exported from the processor for the new decomposition. A negative number indicates that no export data has been computed and the export arrays should be ignored.
export_global_ids	Upon return, an array of *num_export* global IDs of objects to be exported from the processor for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.
export_local_ids	Upon return, an array of *num_export* local IDs of objects to be exported from the processor for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.

export_procs	Upon return, an array of size *num_export* containing the processor IDs of processors owning (in the old decomposition) the objects to be exported for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.
export_to_parts	Upon return, an array of size *num_export* containing the partition IDs of partitions to which the objects will be exported for the new decomposition. If this array is non-null, it must be allocated by **Zoltan_Special_Malloc**.

Returned Value:

int	Error code.

[Table of Contents | Next: Data Structures | Previous: Load-Balancing Interface Routines]

Data Structures

The main data structures for the algorithm should be pointed to by the *LB.Data_Structure* field of the **Zoltan_Struct**. This requirement allows reuse of data structures from one invocation of the new load-balancing algorithm to the next. It also prevents the use of global data structures for the algorithm so that multiple instances of the algorithm may be used (i.e., the same algorithm can be used for multiple **Zoltan_Struct** structures). An example showing the construction of data structures for the Recursive Coordinate Bisection (RCB) algorithm is included in the figure below.

```
/* Allocate RCB data structure for this Zoltan structure.
 * If the previous data structure still exists, free the Dots first;
 * the other fields can be reused.
 */
if (zz->LB.Data_Structure == NULL) {
   rcb = (RCB_STRUCT *) ZOLTAN_MALLOC(sizeof(RCB_STRUCT));
   zz->LB.Data_Structure = (void *) rcb;
   rcb->Tree_Ptr = (struct rcb_tree *)
                   ZOLTAN_MALLOC(zz->Num_Proc*sizeof(struct rcb_tree));
   rcb->Box = (struct rcb_box *) ZOLTAN_MALLOC(sizeof(struct rcb_box));
}
else {
   rcb = (RCB_STRUCT *) zz->LB.Data_Structure;
   ZOLTAN_FREE(&(rcb->Dots));
}
```

Example demonstrating allocation of data structures for the RCB algorithm. (Taken from rcb/rcb_util.c.)

The data needed for the algorithm is collected through calls to the query functions registered by the application. Algorithms should test the query function pointers for NULL and report errors when needed functions are not registered. The appropriate query functions can be called to build the algorithm's data structures up front, or they can be called during the algorithm's execution to gather data only as it is needed. The figure below shows how the *Dots* data structure needed by RCB is built. The call to *zz->Get_Num_Obj* invokes an **ZOLTAN_NUM_OBJ_FN** query function to determine the number of objects on the processor. Space for the *Dots* data structure is allocated through calls to **ZOLTAN_MALLOC**, **ZOLTAN_MALLOC_GID_ARRAY**, and **ZOLTAN_MALLOC_LID_ARRAY**. The *Dots* information is obtained through a call to the Zoltan service function **Zoltan_Get_Obj_List**; this function calls either an **ZOLTAN_OBJ_LIST_FN** or an **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair to get the object IDs and weights. The data for each *Dot* is set in the function *initialize_dot*, which includes calls to *zz->Get_Geom*, an **ZOLTAN_GEOM_FN** query function.

```
/*
 * Allocate space for objects.  Allow extra space
 * for objects that are imported to the processor.
 */

*num_obj = zz->Get_Num_Obj(zz->Get_Num_Obj_Data, &ierr);
if (ierr) {
  ZOLTAN_PRINT_ERROR(zz->Proc, yo,
                "Error returned from Get_Num_Obj.");
  return(ierr);
}

*max_obj = (int)(1.5 * *num_obj) + 1;
*global_ids = ZOLTAN_MALLOC_GID_ARRAY(zz, (*max_obj));
*local_ids  = ZOLTAN_MALLOC_LID_ARRAY(zz, (*max_obj));
*dots = (struct Dot_Struct *)
         ZOLTAN_MALLOC((*max_obj)*sizeof(struct Dot_Struct));

if (!(*global_ids) || (zz->Num_LID && !(*local_ids)) || !(*dots)) {
  ZOLTAN_PRINT_ERROR(zz->Proc, yo, "Insufficient memory.");
  return(ZOLTAN_MEMERR);
}

if (*num_obj > 0) {

  if (wgtflag) {

    /*
     *  Allocate space for object weights.
     */

    objs_wgt = (float *) ZOLTAN_MALLOC((*num_obj)*sizeof(float));
    if (!objs_wgt) {
      ZOLTAN_PRINT_ERROR(zz->Proc, yo, "Insufficient memory.");
      return(ZOLTAN_MEMERR);
    }
    for (i = 0; i < *num_obj; i++) objs_wgt[i] = 0.;
  }

  /*
   *  Get list of objects' IDs and weights.
   */

  Zoltan_Get_Obj_List(zz, *global_ids, *local_ids, wgtflag,
                 objs_wgt, &ierr);
  if (ierr) {
    ZOLTAN_PRINT_ERROR(zz->Proc, yo,
                  "Error returned from Zoltan_Get_Obj_List.");
    ZOLTAN_FREE(&objs_wgt);
    return(ierr);
  }
```

```
  ierr = initialize_dot(zz, *global_ids, *local_ids, *dots,
                        *num_obj, wgtflag, objs_wgt);
  if (ierr == ZOLTAN_FATAL || ierr == ZOLTAN_MEMERR) {
    ZOLTAN_PRINT_ERROR(zz->Proc, yo,
                  "Error returned from initialize_dot.");
    ZOLTAN_FREE(&objs_wgt);
    return(ierr);
  }

  ZOLTAN_FREE(&objs_wgt);
}
```

Example demonstrating how data structures are built for the RCB algorithm. (Taken from rcb/shared.c.)

The data structures pointed to by *zz->LB.Data_Structure* will be freed at some point, and may be copied.

A function that frees these structures and resets *zz->LB.Data_Structure* to NULL should be written. The function should be called when the load-balancing algorithm exits, either normally or due to an error condition. The function **Zoltan_RCB_Free_Structure** in *rcb/rcb_util.c* may be used as an example.

If your algorithm uses the KEEP_CUTS parameter, a function that copies one *zz->LB.Data_Structure* to another is required. This is particularly important for C++, which may create temporary objects at runtime by invoking a copy operator (which will call your copy function). It is a convenience for C applications, which may wish to copy one Zoltan_Struct to another. See the function **Zoltan_RCB_Copy_Structure** in *rcb/rcb_util.c* for an example.

[Table of Contents | Next: Memory Management | Previous: Load-Balancing Function Implementation]

Zoltan Developer's Guide | *Next* | *Previous*

Memory Management in Zoltan Algorithms

Zoltan uses a memory management package to simplify debugging of memory problems. It is strongly recommended that algorithm developers use the routines in this package, such as **ZOLTAN_MALLOC** , **ZOLTAN_CALLOC** and **ZOLTAN_FREE**, instead of the standard C routines for most memory management.

Macros that simplify the allocation of global and local identifiers (IDs) are defined in *zz/zz_id_const.h*. These macros are described in the ID Data Types section. The macros include error checking for the allocations and, thus, their use is highly recommended.

When a dynamic structure needs to be returned to the application, special memory allocation routines are needed. For example, the import and export lists of data to migrate are returned to an application from **Zoltan_LB_Partition** and **Zoltan_Invert_Lists**. There are two special routines for managing memory for such situations, called **Zoltan_Special_Malloc** and **Zoltan_Special_Free**. Algorithms must use these functions to maintain compatibility with both C and Fortran90 applications; these special routines manage memory in a way that is compatible with both languages.

Some load-balancing algorithms may contain persistent data structures, that is, data structures that are preserved between calls to the load-balancing routine. The **Zoltan_Struct** structure contains a field *LB.Data_Structure* for this purpose, allowing multiple Zoltan structures to preserve their own decomposition data. The developer should write a function that frees this data structure. Use **Zoltan_RCB_Free_Structure** as an example.

---

int **Zoltan_Special_Malloc**(struct **Zoltan_Struct** **zz*, void ***array*, int *size*, **ZOLTAN_SPECIAL_MALLOC_TYPE***type*);

The **Zoltan_Special_Malloc** routine allocates memory to be returned to the application by Zoltan (e.g., the result arrays of **Zoltan_LB_Partition** and **Zoltan_Invert_Lists**). Returned memory must be allocated by **Zoltan_Special_Malloc** to insure it is allocated by the same language as the application. Memory allocated by **Zoltan_Special_Malloc** must be deallocated by **Zoltan_Special_Free**.

Arguments:

zz	The Zoltan structure currently in use.
array	Upon return, a pointer to the allocated space. Usually of type int** or **ZOLTAN_ID_PTR***.
size	The number of elements (not bytes) to be allocated.
type	The type of array to allocate. Must be one of **ZOLTAN_SPECIAL_MALLOC_INT**, **ZOLTAN_SPECIAL_MALLOC_GID** or **ZOLTAN_SPECIAL_MALLOC_LID** for processor numbers, global IDs and local IDs, respectively.

Returned Value:

int	1 if the allocation succeeded; 0 if it failed.

Example:

```
ierr = Zoltan_Special_Malloc(zz, (void **)import_gid,
                             num_import,
                             ZOLTAN_SPECIAL_MALLOC_GID);
```

Allocates an array with *num_import* global IDs and returns a pointer to the allocated space in *import_gid*.

---

int **Zoltan_Special_Free**(struct **Zoltan_Struct** *zz, void **array, **ZOLTAN_SPECIAL_MALLOC_TYPE** *type*);

Zoltan_Special_Free frees memory allocated by **Zoltan_Special_Malloc**. The array pointer is set to NULL upon return.

Arguments:

zz	The Zoltan structure currently in use.
array	The array to be freed. Upon return, the pointer is set to NULL.
type	The type of the array. Must be one of **ZOLTAN_SPECIAL_MALLOC_INT**, **ZOLTAN_SPECIAL_MALLOC_GID** or **ZOLTAN_SPECIAL_MALLOC_LID** for processor numbers, global IDs and local IDs, respectively.

Returned Value:

int	1 if the deallocation succeeded; 0 if it failed.

Example:

```
ierr = Zoltan_Special_Free(zz, (void **)import_gid,
                           ZOLTAN_SPECIAL_MALLOC_GID);
```

Frees the **global IDs** array *import_gid* and sets it to NULL.

[Table of Contents | Next: Parameters | Previous: Data Structures]

Adding new parameters

All parameters in Zoltan should be set and accessed through the parameter setting routines. To add a new parameter to an existing method, you need to do the following:

- In the source code for the desired method, search for the place where the static array of parameters is defined. It will look something like: *static PARAM_VARS Method_params[] = { ... }*. Add a line with the name of the new parameter, a pointer to the variable you want to associate (usually NULL), and its type.
- In the method source code, bind the parameter to a local variable through **Zoltan_Bind_Param**. Make sure you do this before **Zoltan_Assign_Param_Vals** is invoked.
- Update the parameter function for the method in question. This routine is typically called **Zoltan_***Method***_Set_Param**. This routine checks if a given string is a valid parameter for that method. It may also verify the values.

When you add a new method to Zoltan, you also need to:

- Write a parameter function for your method that checks whether a given string and value is a valid parameter pair for your method. See **Zoltan_RCB_Set_Param** in *rcb/rcb.c* for an example.
- Let your method access the parameters via **Zoltan_Bind_Param** and **Zoltan_Assign_Param_Vals**.
- Change the parameter function array in *params/set_params.c* to include your parameter function. Simply add a new entry to the static array that looks like: *static ZOLTAN_SET_PARAM_FN * Param_func[] = {...}*.
- Make sure your method uses the key parameters in Zoltan correctly.

One useful convention is to put your method routine and your corresponding parameter function in the same source file. This way you can define the parameters in a static array. This convention eliminates the risk of bugs caused by duplicate declarations (that are, incorrectly, not identical).

[Table of Contents | Next: Partition Remapping | Previous: Memory Management]

Partition Remapping

Partition remapping can be incorporated into load-balancing algorithms. The partition remapping algorithm works as follows:

- After partitioning within an **ZOLTAN_LB_FN** but before import or export lists are built, the partitioning algorithm calls **Zoltan_LB_Remap**.
- **Zoltan_LB_Remap** builds a bipartite graph based on local import or export information (depending on which is available in the partitioning algorithm). Vertices of the graph are processor or partition numbers used in the old (input to the partitioner) and new (computed by the partitioner) decompositions. Edges connect old and new vertices; edge weight for edge e_{ij} is the number of objects in old partition *i* that are also in new partition *j*. The bipartite graph is stored as a hypergraph, so that Zoltan's hypergraph matching routines may be applied.
- **Zoltan_LB_Remap** gathers the local hypergraph edges onto each processor and performs a serial matching of the vertices. This matching defines the remapping.
- **Zoltan_LB_Remap** remaps the input processor and partition information to reflect the remapping and returns the result to the application. It also builds array *zz->LB.Remap* that is used in other functions (e.g., **Zoltan_LB_Box_PP_Assign** and **Zoltan_LB_Point_PP_Assign**).
- Using the remapping information returned from **Zoltan_LB_Remap**, the partitioning algorithm builds the import or export lists to return to the application. Note: if the partition algorithm builds import lists, data may have to be moved to appropriate processors before building import lists to reflect the remapping; see *rcb/shared.c* for an example.

---

int **Zoltan_LB_Remap** (struct **Zoltan_Struct** *zz, int *new_map, int num_obj, int *procs, int *old_parts, int *new_parts, int export_list_flag);

Zoltan_LB_Remap remaps computed partition (or processor) numbers in an attempt to maximize the amount of data that does not have to be migrated to the new decomposition. It is incorporated directly into partitioning algorithms, and should be called after the new decomposition is computed but before return lists (import or export lists) are created. **Zoltan_LB_Remap** should be invoked when Zoltan parameter *REMAP* is one. Even when *REMAP* is one, remapping is not done under a number of conditions; these conditions are listed with the description of *REMAP*.

Arguments:

zz	A pointer to the **Zoltan_Struct** used in the partitioning algorithm.
new_map	Upon return, a flag indicating whether remapping was actually done. Remapping is not done under a number of conditions (described with parameter *REMAP*) or when the computed remap gives a worse or equivalent result than the decomposition computed by the partitioning algorithm.
num_obj	Input: the number of objects the processor knows about after computing the decomposition. If the partitioning algorithm computes export lists, *num_obj* is the number of objects stored on the processor; if it computes import lists, *num_obj* is the number of objects that will be stored on the processor in the new decomposition.
procs	Upon input: an array of size *num_obj* containing processor assignments for the objects; if *export_list_flag* == 1, *procs* contains processor assignments in the NEW decomposition (computed by the partitioner); otherwise, *procs* contains processor assignments in the OLD decomposition (input by the application). Upon return, *procs* contains remapped processor assignments for the NEW decomposition, regardless of the value of *export_list_flag*.
old_parts	Upon input: an array of size *num_obj* containing partition assignments for the objects in the OLD decomposition (input by the application).

new_parts	Upon input: an array of size *num_obj* containing partition assignments for the objects in the NEW decomposition (computed by the partitioning algorithm). Upon return: *new_parts* contains remapped partition assignments in the NEW decomposition.
export_list_flag	Flag indicating whether the partitioning algorithm computes export lists or import lists. The procedure for building the bipartite graph depends on whether the partitioning algorithm knows export or import information.

Returned Value:

int	Error code.

[Table of Contents | Next: Migration Tools | Previous: Adding new parameters]

Migration Tools

The migration tools in the Zoltan library perform communication necessary for data migration in the application. The routine **Zoltan_Migrate** calls application-registered packing routines to gather data to be sent to other processors. It sends the data using the unstructured communication package. It then calls application-registered unpacking routines for each imported object to add received data to the processor's data structures. See the Zoltan User's Guide for more details on the use of and interface to the migration tools.

In future releases, the migration tools will be updated to use MPI data types to support heterogeneous computing architectures.

[Table of Contents | Next: FORTRAN Interface | Previous: Partition Remapping]

Zoltan Developer's Guide | *Next* | *Previous*

FORTRAN Interface

With any change to the user API of Zoltan, the Fortran interface should be modified to reflect the change. This section contains information about the implementation of the Fortran interface which should cover most situations.

If you have questions or need assistance, contact william.mitchell@nist.gov.

If changes are made to functions that are called by *zdrive*, then the changes should also be made to *zfdrive*. Changes to the Fortran interface can be tested by building and running *zfdrive*, if the changes are in functions called by *zfdrive*. The *zfdrive* program works the same way as *zdrive* except that it is restricted to the Chaco examples and simpler input files.

Any changes in the interface should also be reflected in the Fortran API definitions in the Zoltan User's Guide.

Structures

All structures in the API have a corresponding user-defined type in the Fortran interface. If a new structure is added, then modifications will be required to *fort/fwrap.fpp* and *fort/cwrap.c*. In these files, search for **Zoltan_Struct** and "do like it does."

An explanation of how structures are handled may help. The Fortran user-defined type for the structure simply contains the address of the structure, i.e., the C pointer returned by a call to create the structure. Note that the user does not have access to the components of the structure, and can only pass the structure to functions. Within the Fortran structure, the address is stored in a variable of type(**Zoltan_PTR**), which is a character string containing one character for each byte of the address. This gives the best guarantee of portability under the Fortran and C standards. Also, to insure portability of passing character strings, the character string is converted to an array of integers before passing it between Fortran and C. The process of doing this is most easily seen by looking at **Zoltan_Destroy**, which has little else to clutter the code.

Modifications to an existing Zoltan interface function

If the argument list or return type of a user-callable function in Zoltan changes, the same changes must be made in the Fortran interface routines. This involves changes in two files: *fort/fwrap.fpp* and *fort/cwrap.c*. In these files, search for the function name with the prefix **Zoltan_** omitted, and modify the argument list, argument declarations, return type, and call to the C library function as appropriate. When adding a new argument, if there is not already an argument of the same type, look at another function that does have an argument of that type for guidance.

Removing a Zoltan interface function

If a user callable function is removed from the Zoltan library, edit *fort/fwrap.fpp* and *fort/cwrap.c* to remove all references to that function.

Adding a new Zoltan interface function

Adding a new function involves changes to the two files *fort/fwrap.fpp* and *fort/cwrap.c*. Perhaps the easiest way to add a new function to these files is to pick some existing function, search for all occurrences of it, and use that code as a guide for the implementation of the interface for the new function. **Zoltan_LB_Point_Assign** is a nice minimal function to use as an example. Use a case insensitive search on the name of the function without the **Zoltan_LB_** prefix, for example **point_assign**.

Here are the items in *fwrap.fpp*:

- public statement: The name of the function should be included in the list of public entities.
- interface for the C wrapper: Copy one of these and modify the function name, argument list and declarations for the new function. The name is of the form **Zfw_LB_Point_Assign** (fw stands for Fortran wrapper).
- generic interface: This assigns the function name to be a generic name for one or more module procedures.
- module procedure(s): These are the Fortran-side wrapper functions. Usually there is one module procedure of the form **Zf90_LB_Point_Assign**. If one argument can have more than one type passed to it (for example, it is type void in the C interface), then there must be one module procedure for each type that can be passed. These are distinguished by appending a digit to the end of the module procedure name. If n arguments can have more than one type, then n digits are appended. See **Zoltan_LB_Free_Part** for example. Generally the module procedure just calls the C-side wrapper function, but in some cases it may need to coerce data to a different type (e.g., **Zoltan_Struct**), or may actually do real work (e.g., **Zoltan_LB_Free_Part**).

Here are the items in *cwrap.c*:

- name mangling: These are macros to convert the function name from the case sensitive C name (for example, **Zfw_LB_Point_Assign**) to the mangled name produced by the Fortran compiler. There are four of these for each function:
  - lowercase (**zfw_lb_point_assign**),
  - uppercase (**ZFW_LB_POINT_ASSIGN**),
  - lowercase with underscore (**zfw_lb_point_assign_**), and
  - lower case with double underscore (**zfw_point_assign__** but the second underscore is appended only if the name already contains an underscore, which will always be the case for names starting with **Zfw_**).
- C-side wrapper function: Usually this just calls the Zoltan library function after coercing the form of the data (for example, constructing the pointer to **Zoltan_Struct** and call-by-reference to call-by-value conversions).

Query functions

If a query function is added, deleted or changed, modifications must be made to *fort/fwrap.fpp* and *fort/cwrap.c*, similar to the modifications for interface functions, and possibly also *include/zoltan.h* and *zz/zz_const.h*.

Here are the places query functions appear in *fwrap.fpp*:

- public statement for the **ZOLTAN_FN_TYPE** argument: These are identical to the C enumerated type.
- definition of the **ZOLTAN_FN_TYPE** arguments: There are two groups of these, one containing subroutines (void functions) and one containing functions (int functions). Put the new symbol in the right category. The value assigned to the new symbol must agree exactly with where the symbol appears in the order of the enumerated type.

Here are the places query functions appear in *cwrap.c*:

- reverse wrappers: These are the query functions that are actually called by the Zoltan library routines when the query function was registered from Fortran. They are just wrappers to call the registered Fortran routine, coercing argument types as necessary. Look at **Zoltan_Num_Edges_Fort_Wrapper** for an example.
- **Zfw_Set_Fn**: This has a switch based on the value of the **ZOLTAN_FN_TYPE** argument to set the Fortran query function and wrapper in the **Zoltan_Struct**.

In *zz/zz_const.h*, if a new field is added to the structures for a new query function, it should be added in both C and Fortran forms. In *include/zoltan.h*, if a new typedef for a query function is added, it should be added in both C and Fortran forms. See these files for examples.

Enumerated types and defined constants

Enumerated types and defined constants that the application uses as the value for an arguments must be placed in *fwrap.fpp* with the same value. See **ZOLTAN_OK** for an example.

[Table of Contents | Next: C++ Interface | Previous: Migration Tools]

Zoltan Developer's Guide | *Next* | *Previous*

C++ Interface

As with the Fortran interface just described, any change to the user API of Zoltan should be reflected in the C++ interface. This section explains the conventions used in the C++ interface, which you will want to follow when you modify or expand it.

Classes

The C language Zoltan library already observes the principles of object oriented program design. Each sub function of Zoltan (load balancing, timing, etc.) has a data structure associated with it. This data structure maintains all the state required for one instance of that sub function. Each request of the library for some operation requires that data structure.

The classes in the Zoltan C++ library follow the structure just described. Each class is defined in a header file and encapsulates a Zoltan data structure and the functions that operate on that structure. A C++ application wishing to use a feature of Zoltan, would include the feature's header file in it's source, and link with the Zoltan C library.

The C language load balancing data stucture (**Zoltan_Struct**) and the C functions that operate on it are accessed through the C++ **Zoltan** class, defined in *zoltan_cpp.h*.

The communication package is encapsulated the **Zoltan_Comm** class defined in *zoltan_comm_cpp.h*. Again, to use the communication utility of Zoltan from a C++ program, include *zoltan_comm_cpp.h* and use the C++ methods defined there.

The C++ Zoltan timer class is called **Zoltan_Timer_Object** and is defined in *zoltan_timer_cpp.h*.

The distributed directory utility of Zoltan is encapsulated in the class **Zoltan_DD** defined in *zoltan_dd_cpp.h*

Programming Conventions

When modifying the interface to Zoltan , you will want to modify the appropriate C++ header file accordingly. This section describes the conventions to follow to maintain a consistent and correct library interface for the C++ user of Zoltan.

Namespaces

In order to maintain portability across platforms, there is no Zoltan namespace. Many C++ compilers do not support namespaces at this time. The name of each Zoltan class begins with **Zoltan_**, and hopefully this will never clash with another namespace.

Class names

Class names are **Zoltan_** followed by text indicating the sub function of Zoltan that is encapsulated by the class.

Method names

Method names are derived from the C library function names in such a way that the name will be obvious to a person familiar with the C library. We remove the beginning of the C library name, the part that identifies the subset of the Zoltan library that the function is part of, and keep the last part of the C library name, the part that describes what the function does. For example the C function **Zoltan_LB_Partition** becomes the C++ method **LB_Partition** in the class **Zoltan** and C function **Zoltan_Comm_Create** becomes the C++ method **Create** in the class **Zoltan_Comm**.

Const methods

All class methods which can be declared **const**, because they do not modify the object, should be declared **const**. This allows C++ programmers to call these methods on their **const** objects.

Declaration of method parameters

Parameters that are not changed in the method should be declared **const**. This can get complicated, but it helps to read declarations from right to left. **const int * & p** says *p is a reference to a pointer to a const int* and means the method will not change the value pointed to by *p*. On the other hand **int * const & p** says that *p is a reference to a const pointer to int* so the method will not change the pointer.

Variables that are passed by value in a C function will be passed by const reference in the C++ method. This is semantically the same, but it is more efficient, and it will work with temporary variables created by a compiler.

If a C function takes a pointer to a single built-in type (not an aggregate type), the associated C++ method will take a reference variable. If a C function takes a pointer to a pointer, the C++ function will take a pointer reference. The references are more efficient, and it is the behavior a C ++ programmer expects. A pointer to an array remains a pointer to an array.

C function parameter	C++ method parameter	method's const behavior
int val	const int &val	won't change value
int *singlep	int &singlep	may change value
	const int &singlep	won't change value
int **singlep	int *&singlep	may change pointer or value
	const int * &p	won't change value
	int *const &p	won't change pointer to value
	const int * const &p	won't change anything
int *arrayp	int *arrayp	may change array contents
	const int * arrayp	won't change array contents

If a C function takes a pointer to an array of *char*, the associated C++ method will take a *string* object.

C function parameter	C++ method parameter
char *fname	std::string &fname

In all honesty, it is tedious to carefully apply const'ness in parameter declarations, and we did not do it consistently throughout the C++ wrapping of Zoltan. Please feel free to add *const* declarations where they belong, and try to use them correctly if you add or modify Zoltan C++ methods.

Copy constructor, copy operator

Each class should have a copy constructor and a copy operator.

Keeping the C++ interface up-to-date

Here we provide a checklist of things to be done when the C interface to the Zoltan library is changed:

- If a new major component is added to Zoltan, create a C++ class for that component in a new header file, using the programming conventions described above.
- If functions are added or removed, or their parameter lists are changed, then update the header file defining the class that contains those functions.
- When Zoltan data structures are changed, be sure to change the C functions that copy the data structure. (They contain **Copy** in their name.) Correct copying is more important in C++, where the compiler may generate new temporary objects, than it is in C.
- If you change the C++ API, be sure to change:
    - **zCPPdrive**, the test program for the Zoltan C++ library
    - the C++ examples in the **Examples** directory
    - the method prototypes in the Zoltan User's Guide.

[Table of Contents | Next: References | Previous: FORTRAN Interface]

References

1. M. J. Berger and S. H. Bokhari. "A partitioning strategy for nonuniform problems on multiprocessors." *IEEE Trans. Computers*, C-36 (1987), 570-580.
2. K. Devine, B. Hendrickson, M. St.John, E. Boman, and C. Vaughan. "Zoltan: A Dynamic Load-Balancing Library for Parallel Applications, User's Guide." Sandia National Laboratories Tech. Rep. SAND99-1377, Albuquerque, NM, 1999.
3. H. C. Edwards. *A Parallel Infrastructure For Scalable Adaptive Finite Element Methods and Its Application To Least Squares C^ (inf) Collocation*. Ph.D. Dissertation, University of Texas at Austin, May, 1997.
4. B. Hendrickson and K. Devine. "Dynamic Load Balancing in Computational Mechanics." *Comp. Meth. Appl. Mech. Engrg.*, **184** (2000) 484-500.
5. B. Hendrickson and R. Leland. ``The Chaco User's Guide, version 2.0." Sandia National Laboratories Tech. Rep. SAND94-2692, Albuquerque, NM, 1994. http://www.cs.sandia.gov/CRF/chac.html
6. 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/
7. C. Walshaw. "Parallel Jostle Library Interface: Version 1.1.7." Tech. Rep., Univ. of Greenwich, London, 1995. http://www.gre.ac.uk/jostle

---

Appendix: Using the Test Drivers: *zdrive*, *zCPPdrive* and *zfdrive*

Introduction

In order to facilitate development and testing of the Zoltan library, simple driver programs, *zdrive* (C), *zCPPdrive* (C++) and *zfdrive* (Fortran90), are included with the library distribution. The concept behind the drivers is to read in mesh or graph information from files, run Zoltan, and then output the new assignments for each processor to another file. The test drivers *zdrive* and *zCPPdrive* read ExodusII/NemesisI parallel FEM files and Chaco input files. Parallel NemesisI files can be created from ExodusII or Genesis file using the NemesisI utilities *nem_slice* and *nem_spread*. The Fortran90 program *zfdrive* reads only Chaco input files.

Source code for *zdrive* is in the *driver* and *ch* directories of the Zoltan distribution. Source code for *zfdrive* is in the *fdriver* directory. The source code for *zCPPdrive* is also in *driver*, and uses some of the same C source files (in *driver* and *ch*) that *zdrive* uses.

To compile the test drivers, use the following commands:

gmake [options] zdrive

gmake [options] zCPPdrive

gmake YES_FORTRAN=1 [options] zfdrive

where the options are described below.

Options to gmake:

ZOLTAN_ARCH=<platform> Specify the target architecture. A corresponding file, *Utilities/Config/Config.<platform>*, containing environment definitions for <*platform*>, must be created in the *Utilities/Config* directory.

The drivers are placed in the *Obj_<platform>* directory.

Running the Test Drivers

The test drivers are run using an input command file. A fully commented example of this file and the possible options can be found in *zdrive.inp*. The default name for the command file is *zdrive.inp*, and the drivers will look for this file in the execution directory if an alternate name is not given on the command line. If another filename is being used for the command file, it should be specified as the first argument on the command line. (Note: *zfdrive* cannot read a command line argument; its input file must be named *zdrive.inp*.)

For an example of a simple input file, see the figure below. In this problem, the method being used for dynamic load balancing is RCB. Input data is read from Chaco input files *simple.graph* and *simple.coords*. Zoltan's DEBUG_LEVEL parameter is set to 3; default values of all other parameters are used. (Note: *zfdrive* can read only a simplified version of the input file. See the *zfdrive* notes in *zdrive.inp* for more details.)

```
Decomposition method  = rcb
Zoltan Parameters     = Debug_Level=3
File Type             = Chaco
File Name             = simple
Parallel Disk Info    = number=0
```

Example zdrive.inp file

The *zdrive* programs creates ascii files named "*file_name*.out.*p*.*n*", where *file_name* is the file name specified in *zdrive.inp*, *p* is the number of processors on which *zdrive* was run, and *n*=0,1,...,*p*-1 is the processor by which the file was created. (For *zfdrive*, the files are named "*file_name*.fout.*p*.*n*".) These files are in the same directory where the input graph file was located for that processor. Each file contains a list of global ids for the elements that are assigned to that processor after running Zoltan. The input decomposition can also be written in this format to files "*file_name*.in.*p*.*n*"; see "zdrive debug level" in *zdrive.inp* for more details.

Decompositions for 2D problems can be written to files that can be plotted by gnuplot. See "gnuplot output" in *zdrive.inp* for more information. Decompositions for 3D problems can be viewed after the test driver has finished by running the graphical tools *vtk_view* or *vtk_write* described next.

Adding New Algorithms

The driver has been set up in such a way that testing new algorithms that have been added to Zoltan is relatively simple. The method that is in the input file is passed directly to Zoltan. Thus, this string must be the same string that the parameter *LB_METHOD* is expecting.

[Table of Contents | Next: Visualizing Geometric Partitions | Previous: References]

</html

```
##############################################################################
# Zoltan Library for Parallel Applications                                   #
# Copyright (c) 2000,2001,2002, Sandia National Laboratories.                #
# This document is released under the GNU Lesser General Public License.     #
# For more info, see the README file in the top-level Zoltan directory.      #
##############################################################################
##############################################################################
# CVS File Information
#    $RCSfile: zdrive.inp,v $
#    $Author: lafisk $
#    $Date: 2006/08/28 17:04:01 $
#    $Revision: 1.31 $
##############################################################################
#
# EXAMPLE OF zdrive.inp INPUT FILE FOR zdrive AND zfdrive.
#
##############################################################################
# GENERAL NOTES
#
# 1) Any line beginning with a "#" is considered a comment and will be
# ignored by the file parser.
#
# 2) The order of the lines IS NOT significant.
#
# 3) Any lines that are optional are marked as such in this file. Unless
# otherwise noted a line is required to exist in any input file.
#
# 4) The case of words IS NOT significant, e.g., "file" IS equivalent
# to "FILE" or "File", etc.
#
# 5) The amount of blank space in between words IS significant. Each
# word should only be separated by a single space.
#
# 6) Blank lines are ignored.
#
#
##############################################################################


#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Decomposition Method = <method>
#
# This line is used to specify the algorithm that Zoltan will use
# for load balancing. Currently, the following methods that are acceptable:
#       rcb - Reverse Coordinate Bisection
#       octpart - Octree/Space Filling Curve
#       parmetis - ParMETIS graph partitioning
#       jostle - Jostle graph partitioning
#       reftree - Refinement tree partitioning
#
#------------------------------------------------------------------------
Decomposition Method    = rcb

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Zoltan Parameters = <options>
#
# This line is OPTIONAL. If it is not included, no user-defined parameters
# will be passed to Zoltan.
```

```
#
# This line is used to to specify parameter values to overwrite the default
# parameter values used in Zoltan.  These parameters will be passed to Zoltan
# through calls to Zoltan_Set_Param(). Parameters are set by entries consisting
# of pairs of strings "<parameter string>=<value string>".
# The <parameter string> should be a string that is recognized by the
# particular load-balancing method being used.
# The parameter entries should be separated by commas.
# When many parameters must be specified, multiple
# "Zoltan Parameters" lines may be included in the input file.
# NOTE:  The Fortran90 driver zfdrive can read only one parameter per line.
#-------------------------------------------------------------------------
Zoltan Parameters       = DEBUG_LEVEL=3
Zoltan Parameters       = RCB_REUSE=0

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# File Type = <file type><,chaco or Matrix Market options>
#
# This line is OPTIONAL. If it is not included, then it is assumed that
# the file type is parallel nemesis.
#
# This line indicates which format the file is in. The current
# file types for this line are:
#       NemesisI        - parallel ExodusII/NemesisI files (1 per processor)
#       Chaco           - Chaco graph and/or geometry file(s)
#       hypergraph      - format documented in driver/dr_hg_readfile.c,
#                         suffix .hg
#       matrixmarket    - Matrix Market exchange format, suffix .mtx
#       matrixmarket+   - our enhanced Matrix Market format, documented in
#                         driver/dr_hg_io.c, includes vertex and edge weights,
#                         and process ownership of matrix data for
#                         a distributed matrix, suffix .mtxp
#
# For NemesisI input, the initial distribution of data is given in the
# Nemesis files.  For Chaco input, however, an initial decomposition is
# imposed by the zdrive.  Four initial distribution methods are provided.
# The method to be used can be specified in the chaco options:
#       initial distribution = <option>
# where <option> is
#       linear  -- gives the first n/p objects to proc 0, the
#                  next n/p objects to proc 1, etc.
#       cyclic  -- assigns the objects to processors as one would
#                  deal cards; i.e., gives the first object to proc 0,
#                  the second object to proc 1, ..., the pth object to
#                  proc (p-1),the (p+1)th object to proc 0, the (p+2)th
#                  object to proc 1, etc.
#       file    -- reads an initial distribution from the input file
#                  <filename>.assign, where File Name is specified by
#                  the "File Name" command line below.
#       owner   -- for vertices, same as "linear."  For hyperedge, send a
#                  copy of a hyperedge to each processor owning one of its
#                  vertices.  (Multiple processors may then store each
#                  hyperedge.)
# If an initial distribution is not specified, the default is linear.
#
# A second Chaco option is to distribute the objects over a subset
# of the processors, not all processors. The syntax for this is:
#       initial procs = k
# where k is an integer between 1 and the number of processors.
# The objects will be evenly distributed among the k first
```

```
# processors, using the distribution method optionally specified by
# the "initial distribution" option.
#
# Example:
#     File Type = chaco, initial distribution = cyclic, initial procs = 2
# will give proc 0 objects 1, 3, 5, ... and proc 1 objects 2, 4, 6, ...
# while procs 2 and higher get no objects.
#
# For hypergraph, matrixmarket and matrixmarket+ files, there are three
# options that determine how zdrive divides the hypergraph or matrix data
# across the processes initially.  (The Zoltan library will redistribute
# these elements yet again before the parallel hypergraph methods begins.)
#
#  initial_distribution = {val}      initial vertex (object) distribution
#
#    linear (default) - First n/p vertices supplied by first process,
#      next n/p vertices supplied by next process, and so on.
#    cyclic - Deal out the vertex ownership in round robin fashion.
#    file - Use process vertex assignment found in the file (.mtxp only)
#
#  initial_pins =  {val}       initial pin (matrix non-zero) distribution
#
#    row (default) - Each zdrive process supplies entire rows of the matrix,
#      in compressed row storage format
#    column - Each zdrive process supplies entire columns of the matrix, in
#      compressed column storage format
#    linear - First n/p pins (matrix non-zeroes) supplied by first process,
#      next n/p pins supplied by next process, and so on.
#    cyclic - Deal out the pin ownership in round robin fashion.
#    file - Use process pin assignment found in the file (.mtxp only)
#    zero - Process zero initially has all pins
#
#  initial_procs = {n}
#     This has the same meaning that it has for Chaco files.  The initial
#     vertices, pins and weights are all provided by only {n} processes.

# NOTE:  The Fortran90 driver zfdrive does not read NemesisI files.
# NOTE:  The Fortran90 driver zfdrive does not accept any Chaco options.
#-------------------------------------------------------------------------
File Type              = NemesisI

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# File Name = <filename>
#
# This line contains the filename for the input finite element mesh.
#
# If the file type is NemesisI then this name refers to the base name
# of the parallel ExodusII files that contain the results. The base
# name is the parallel filename without the trailing .<# proc>.<file #>
# on it. This file must contain the Nemesis global information.
#
# If the file type is Chaco, this name refers to the base name of the
# Chaco files containing graph and/or coordinates information.  The
# file <filename>.graph will be read for the Chaco graph information;
# The file <filename>.coords will be read for Chaco geometry information.
# The optional file <filename>.assign may be read for an initial decomposition
# by specifying "initial distribution=file" on the "File Type" input line.
# For more information about the format of these files, see
# the Chaco user's guide.
#-------------------------------------------------------------------------
```

```
File Name       = testa.par

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Parallel Disk Info = <options>
#
# This line is OPTIONAL. If this line is left blank, then it is assumed
# that there is no parallel disk information, and all of the files are
# in a single directory.  This line is used only for Nemesis files.
#
# This line gives all of the information about the parallel file system
# being used. There are a number of options that can be used with it,
# although for most cases only a couple will be needed. The options are:
#
#      number=<integer> - this is the number of parallel disks that the
#                         results files are spread over. This number must
#                         be specified, and must be first in the options
#                         list. If zero (0) is specified, then all of the
#                         files should be in the root directory specified
#                         below.
#      list={list}      - OPTIONAL, If the disks are not sequential, then a
#                         list of disk numbers can be given. This list should
#                         be enclosed in brackets "{}", and the disk numbers
#                         can be seperated by any of the following comma,
#                         blank space, tab, or semicolon.
#      offset=<integer> - OPTIONAL, This is the offset from zero that the
#                         disk numbers begin with. If no number is specified,
#                         this defaults to 1. This option is ignored if
#                         "list" is specified.
#      zeros            - OPTIONAL, This specifies that leading zeros are
#                         used in the parallel file naming convention. For
#                         example, on the Paragon, the file name for the
#                         first pfs disk is "/pfs/tmp/io_01/". If this is
#                         specified, then the default is not to have leading
#                         zeros in the path name, such as on the teraflop
#                         machine "/pfs/tmp_1/".
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#-------------------------------------------------------------------------
Parallel Disk Info      = number=4,zeros

#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Parallel file location = <options>
#
# This line is OPTIONAL, only if the above line is excluded as well, or
# the number of raids is specified as zero (0). If this line is excluded,
# then the root directory is set to the execution directory, ".", and all
# files should be in that directory.  This line is used only for Nemesis
# files.
#
# This line gives all of the information about where the parallel files are
# located. There are only two options for this line, and both must be
# specified. The options are:
#      root=<root directory name>
#        This line is used to specify what the name of the root directory is
#        on the target machine. This can be any valid root directory
#        name. For example, if one is running on an SGI workstation and
#        using the "tflop" numbering scheme then you could use something
#        similar to "/usr/tmp/pio_" in this field so that files would be
#        written to root directories named:
#             /usr/tmp/pio_1
```

```
#             /usr/tmp/pio_2
#                   .
#                   .
#                   .
#             /usr/tmp/pio_<Parallel Disk Info, number>
#
#       subdir=<subdirectory name>
#         This line specifies the name of the subdirectory, under the root
#         directory, where files are to be written. This is tacked onto
#         the end of the "root" after an appropriate integer is added to
#         "root". Continuing with the example given for "root", if "subdir"
#         had a value of "run1/input" files would be written to directories
#         named:
#             /usr/tmp/pio_1/run1/input/
#             /usr/tmp/pio_1/run1/input/
#                   .
#                   .
#                   .
#             /usr/tmp/pio_<Parallel Disk Info, number>/run1/input/
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#-------------------------------------------------------------------------
Parallel File Location  = root=/pfs/io_, subdir=mmstjohn

#-------------------------------------------------------------------------
# Zdrive debug level = <integer>
#
# This line is optional.  It sets a debug level within zdrive (not within
# Zoltan) that determines what output is written to stdout at runtime.
# The currently defined values are listed below.  For a given debug level
# value i, all debug output for levels <= i is printed.
#
#       0  --  No debug output is produced.
#       1  --  Evaluation of the initial and final partition is done
#              through calls to driver_eval and Zoltan_LB_Eval.
#       2  --  Function call traces through major driver functions are
#              printed.
#       3  --  Generate output files of initial distribution.
#              Debug Chaco input files.
#       4  --  Entire distributed mesh (elements, adjacencies, communication
#              maps, etc.) is printed.  This output is done serially and can
#              be big and slow.
#
# Default value is 1.
#-------------------------------------------------------------------------
Zdrive debug level = 1

#-------------------------------------------------------------------------
# text output = <integer>
#
# This line is optional.  If the integer specified is greater than zero,
# zdrive produces files listing the partition and processor assignment of
# each object.  When "text output = 1," P files are generated, where P is
# the number of processors used for the run.  Files have suffix ".out.P.N",
# where P is the number of processors and N = 0,...,P-1 is the processor that
# generated the particular file.
#
# Default value is 1.
#-------------------------------------------------------------------------
text output = 1
```

```
#---------------------------------------------------------------------------
# gnuplot output = <integer>
#
# This line is optional.  If the integer specified is greater than zero,
# zdrive produces files that can be plotted using gnuplot.  Each processor
# generates files containing its decomposition; these files are named
# similarly to the standard output filenames generated by zdrive but they
# include a "gnu" field.  A file containing the gnuplot commands to actually
# plot the decomposition is also generated; this file has a ".gnuload" suffix.
# To plot the results, start gnuplot; then type
#   load "filename.gnuload"
#
# The decomposition can be based on processor assignment or partition
# assignment.  See zdrive input line "plot partitions".
#
# For Chaco input files, edges are not drawn between neighboring subdomains (
# as Chaco input is balanced with respect to graph nodes).  Data style
# "linespoints" is used; this style can be changed using gnuplot's
# "set data style ..." command.
#
# In addition, processor assignments are written to the parallel Nemesis files
# to be viewed by other graphics packages (avs, mustafa, blot, etc.).  Note
# that the parallel Nemesis files must have space allocated for at least one
# elemental variable; this allocation is done by nem_spread.
#
# Gnuplot capability currently works only for 2D problems.
#
# Default value is 0.
#---------------------------------------------------------------------------
gnuplot output = 0

#---------------------------------------------------------------------------
# nemesis output = <integer>
#
# This line is optional.  If the integer specified is greater than zero,
# zdrive writes subdomain assignment information to parallel nemesis files.
# These files match the input nemesis file names, but contain a ".blot" suffix.
# The SEACAS utility nem_join can combine these files into a single Exodus file
# for plotting by blot, avs, mustafa, etc.  Note that the input parallel
# Nemesis files must have space allocated for at least one
# elemental variable; this allocation is done by nem_spread.
#
# The decomposition can be based on processor assignment or partition
# assignment.  See zdrive input line "plot partitions".
#
# This option does nothing for Chaco input files.
#
# Default value is 0.
#---------------------------------------------------------------------------
nemesis output = 0

#---------------------------------------------------------------------------
# plot partitions = <integer>
#
# This line is optional.  If the integer specified is greater than zero,
# zdrive writes partition assignments to the gnuplot or nemesis output files;
# one file per partition is generated.
# Otherwise, zdrive writes processor assignments to the gnuplot or nemesis
# output files, with one file per processor generated.
```

```
#
# See zdrive input lines "gnuplot output" and "nemesis output".
#
# Default value is 0 (processor assignments written).
#---------------------------------------------------------------------------
plot partitions = 0

#---------------------------------------------------------------------------
# print mesh info file = <integer>
#
# This line is optional.  If the integer specified is greater than zero,
# zdrive produces files describing the mesh connectivity.  Each processor
# generates a file containing its vertices (with coordinates) and elements
# (with vertex connectivity); these files are named
# similarly to the standard output filenames generated by zdrive but they
# include a ".mesh" suffix.
#
# Default value is 0.
#---------------------------------------------------------------------------
print mesh info file = 0

#---------------------------------------------------------------------------
# Chaco input assignment inverse = <integer>
#
# This line is optional.  It sets the IN_ASSIGN_INV flag, indicating that
# the "inverse" Chaco assignment format should be used if a Chaco assignment
# file is read for the initial decomposition.  If this flag is 0, the assignment
# file lists, for each vertex, the processor to which it is assigned.  If this
# flag is 1, the assignment file includes, for each processor, the number of
# vertices assigned to the processor followed by a list of those vertices.
# See the Chaco User's guide for a more detailed description of this parameter.
#
# Default value is 0.
#---------------------------------------------------------------------------
Chaco input assignment inverse = 0


#---------------------------------------------------------------------------
# Number of Iterations = <integer>
#
# This line is optional.  It indicates the number of time the load-balancing
# method should be run on the input data.  The original input data is passed
# to the method for each invocation.
# Multiple iterations are useful primarily for testing the RCB_REUSE parameter.
#
# Default value is 1.
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#---------------------------------------------------------------------------
Number of Iterations = 1

#---------------------------------------------------------------------------
# zdrive action = <integer>
#
# This line is optional.  It indicates the action the driver should take,
# typically load-balancing or ordering. Valid values are:
#
#       0  --  No action.
#       1  --  Load balance.
#       2  --  Order.
```

```
#       3   --  First load balance, then order.
#
# Default value is 1 (load balance).
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#-------------------------------------------------------------------------
zdrive action = 1


#-------------------------------------------------------------------------
# Test Drops = <integer>
#
# This line signals that zdrive should exercise the box- and point-assign
# capability of Zoltan.  Note that the partitioning method must support
# box- and point-drop, and appropriate parameters (e.g., Keep_Cuts) must also
# be passed to Zoltan; otherwise, an error is returned from the box- and
# point-assign functions.
#
# Default value is 0.
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#-------------------------------------------------------------------------
Test Drops = 0

#-------------------------------------------------------------------------
# Test DDirectory = <integer>
#
# This line signals that zdrive should exercise the Distributed Directory
# utility of Zoltan.  Comparisons between zdrive-generated communication maps
# and DDirectory-generated communication maps are done.  If a difference is
# found, a diagnostic message containing "DDirectory Test" is printed as
# output from zdrive.
#
# Default value is 0.
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#-------------------------------------------------------------------------
Test DDirectory = 0

#-------------------------------------------------------------------------
# Test Null Import Lists = <integer>
#
# This line signals that zdrive should test Zoltan's capability to accept
# NULL import lists to Zoltan_Help_Migrate.  It allows the driver to pass NULL
# import lists.  This flag's value should not affect the output of zdrive.
#
# Default value is 0.
#
# NOTE:  The Fortran90 driver zfdrive ignores this input line.
#-------------------------------------------------------------------------
Test Null Import Lists = 0

#-------------------------------------------------------------------------
# Test Multi Callbacks = <integer>
#
# This line signals that zdrive should test the list-based (MULTI) callback
# functions.  If this line is set to 1, zdrive registers list-based callback
# functions.  Otherwise, callbacks on individual functions are registered.
# This flag's value should not affect the output of zdrive.
#
```

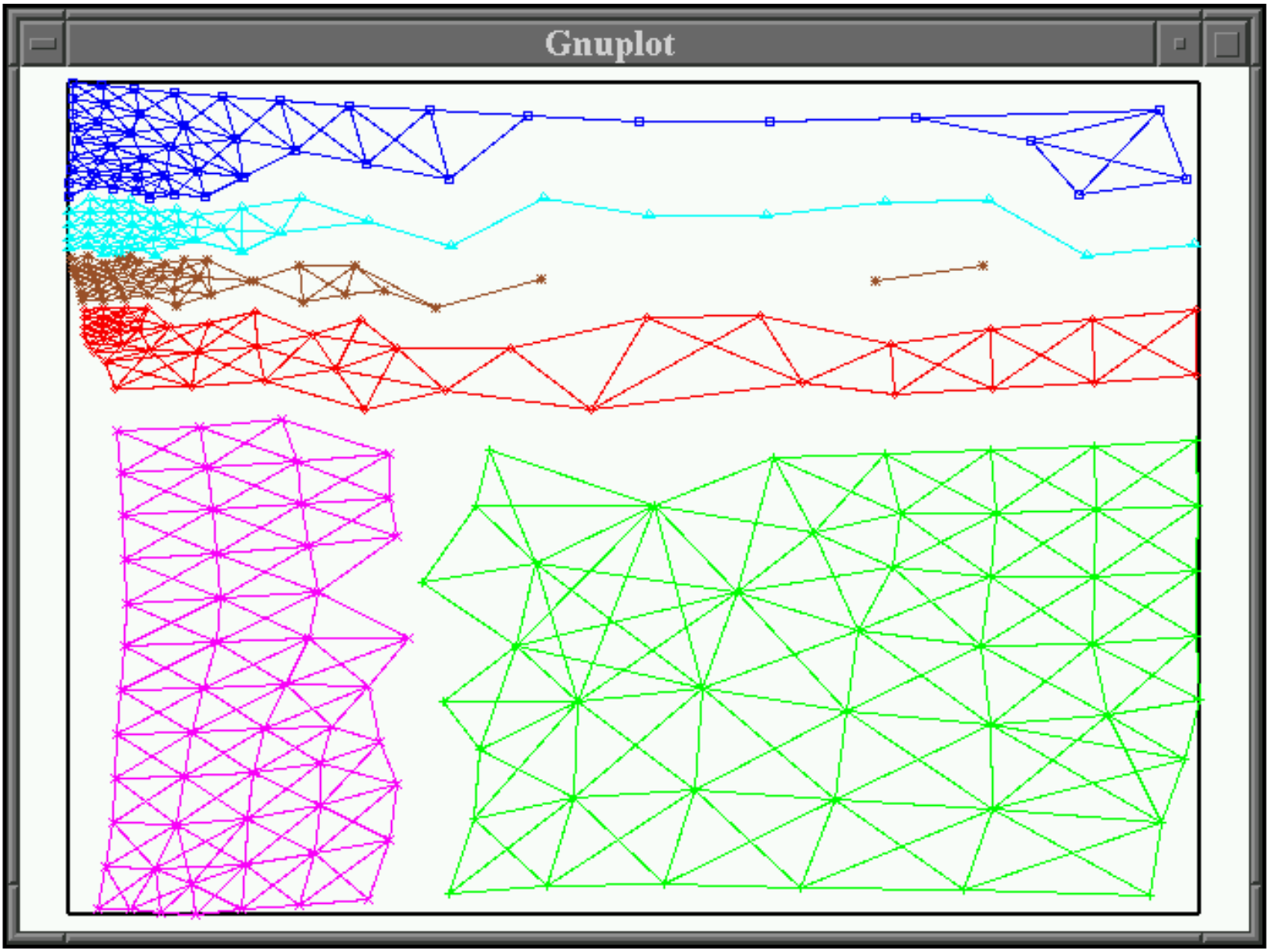
```
# Default value is 0.
#------------------------------------------------------------------------
Test Multi Callbacks = 0

#------------------------------------------------------------------------
# Test Local Partitions = <integer>
#
# This line signals that zdrive should test Zoltan using various values
# of the NUM_LOCAL_PARTITIONS parameter and/or nonuniform partition sizes.
# While setting NUM_LOCAL_PARTITIONS using a "Zoltan Parameter" above
# would make all processors have the same number of local partitions,
# this flag allows different processors to have different values for
# NUM_LOCAL_PARTITIONS.
# Valid values are integers from 0 to 7.
#    0:  NUM_LOCAL_PARTITIONS is not set (unless specified as a
#        "Zoltan Parameter" above).
#    1:  Each processor sets NUM_LOCAL_PARTITIONS to its processor number;
#        e.g., processor 0 requests zero local partitions; processor 1
#        requests 1 local partition, etc.
#    2:  Each odd-numbered processor sets NUM_LOCAL_PARTITIONS to its
#        processor number; even-numbered processors do not set
#        NUM_LOCAL_PARTITIONS.
#    3:  One partition per proc, but variable partition sizes.
#        Only set partition sizes for upper half of procs
#        (using Zoltan_LB_Set_Part_Sizes and global partition numbers).
#    4:  Variable number of partitions per proc, and variable
#        partition sizes. Proc i requests i partitions, each
#        of size 1/i.
#    5:  One partition per proc, but variable partition sizes.
#        Same as case 3, except all sizes are increased by one to
#        avoid possible zero-sized partitions.
#    6:  One partition per proc, but variable partition sizes.
#        When nprocs >= 6, zero-sized partitions on processors >= 2.
#        (This case is of particular interest for HSFC.)
#    7:  One partition per proc, but variable partition sizes.
#        When nprocs >= 6, zero-sized partitions on processors <= 3.
#        (This case is of particular interest for HSFC.)
#
# Default value is 0.
#------------------------------------------------------------------------
Test Local Partitions = 0

#------------------------------------------------------------------------
# Test Generate Files = <integer>
#
# This line signals that zdrive should test Zoltan using Zoltan_Generate_Files
# to produce output files that describe the geometry, graph, or hypergraph
# used in the load-balancing. Such files may be useful for debugging.
#
#   0: Do not generate files.
#   1: Generate files.
#
# Default value is 0.
#------------------------------------------------------------------------
Test Generate Files = 0
```

Appendix: Visualization of Geometric Partitionings

Graphical images of partitioned meshes can help you to understand the geometric partitioning algorithms of Zoltan and to debug new or existing algorithms. The following sections describe methods for visualizing the partitionings computed by the test drivers.

2D problems with *gnuplot*

To view the result of a 2D decomposition performed by the test driver, use the "gnuplot output" option of the test driver input file, as described in *zdrive.inp*. The test driver will write a file that can be loaded into gnuplot. The result for the test mesh in directory *ch_film2*, partitioned into six regions with RCB, is something like this:



3D problems with *vtk_view*

3D visualization requires downloading and compiling the Visualization Toolkit (VTK) library (version 5.0 or later). You can then use the Zoltan top level Makefile to build the *vtk_view* application found in the *util* directory of Zoltan. Build details can be found in the *Config.generic* file in *Utilities/Config*. Note that you will have to download and build CMake, the makefile generator used by VTK, before you can build VTK.

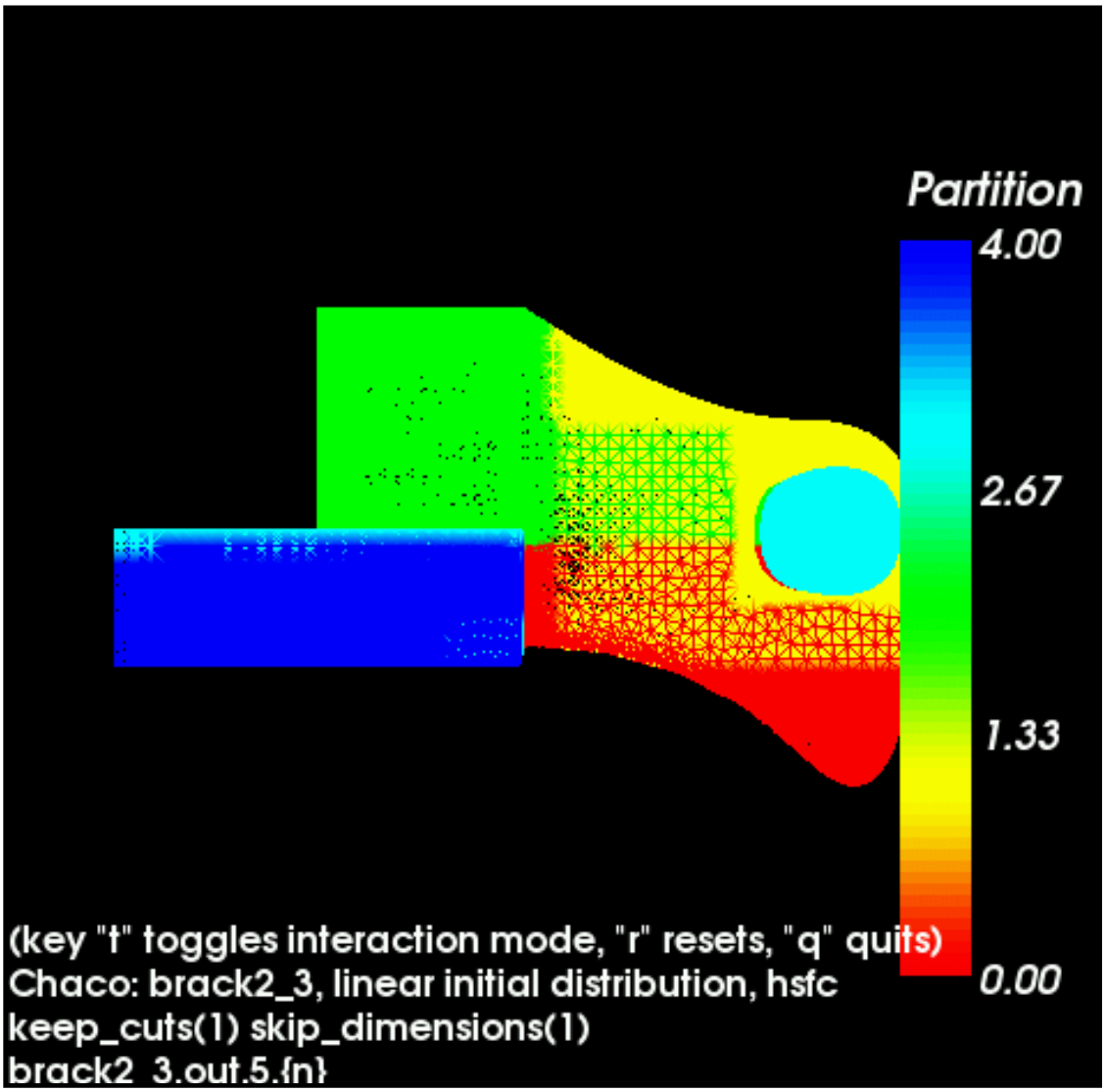
vtk_view is a parallel MPI program. It does not need to be run with the same number of processes with which you ran *zdrive*. You can choose the number of

processes based on the size of the input mesh you will be visualizing, and the computational load of rendering it to an image at interactive rates.

If you run *vtk_view* in the directory in which you ran the test driver, the following will happen:

- *vtk_view* will read *zdrive.inp*, or another input parameter file if you specify a different file on the command line.
- It will read in the same input Chaco or Exodus II mesh that the test driver read in.
- It will read in the *file_name*.out.*p*.*n* files that the test driver wrote listing the partition assigned to every global ID.
- It will open a window on your display, showing the input mesh. For Chaco files, the mesh vertices will be colored by the partition into which Zoltan placed them. For Exodus II files, the mesh elements will be so colored. A scalar bar in the window indicates the mapping from colors to partition numbers. A caption describes the input file name, the decomposition method, the Zoltan parameter settings, and so on. You can use your mouse to rotate the volume, pan and zoom in and out.

The example below shows how *vtk_view* displays the mesh in the test directory *ch_brack2_3* after it has been partitioned with HSFC across 5 processes.



If no test driver output files are found, *vtk_view* will display the mesh without partition IDs.

There are a few additional options that can be added to the test driver input file, that are specifically for *vtk_view*.

zdrive count = <number>	the number of *file_name*.out.*p*.*n* files, also the value of *p*
image height = <number>	number of pixels in height of image (default is *300*)

image width = <number>	number of pixels in width of image (*300*)
omit caption = <1 or 0>	do not print default caption in window if "1" (*0*)
omit scalar bar = <1 or 0>	do not print scalar bar in window if "1" (*0*)
add caption = <text of caption>	display indicated text in the window (*no caption*)

The *zdrive count* option may be required if you have more than one set of test driver output files in the directory. Otherwise, *vtk_view* will look for files of the form *file_name*.out.*p*.*n* for any value *p*. Note that since the window may be resized with the mouse, you may not need *image height* and *image width* unless you must have a very specific window size. Also note that if you ran the Fortan test driver *zfdrive*, you will need to rename the output files from *file_name*.**f**out.*p*.*n* to *file_name*.out.*p*.*n*.

Off-screen rendering with *vtk_write*

In some situations it is not possible or not convenient to open a window on a display. In that case, you can compile *util/vtk_view.cpp* with the flag **OUTPUT_TO_FILE** and it will create a program that renders the image to a file instead of opening a window on a display. (The Zoltan top level Makefile does exactly this when you use the *vtk_write* target.)

Note that while *vtk_view* is built with OpenGL and VTK, *vtk_write* must be built with Mesa GL and a version of the VTK libraries that you have compiled with special Mesa flags and with the Mesa header files. This is because OpenGL implementations are not in general capable of off-screen rendering, and Mesa GL is. The *Config.generic* file in *Utilities/Config* describes in detail how to build Mesa and then VTK for off-screen rendering.

vtk_write goes through the same steps that *vtk_view* does, except at the end it writes one or more image files instead of opening a window on your display. The images begin with a camera focused on the mesh, pointing in the direction of the negative Z-axis. The positive Y-axis is the "up" direction, and we use a right-handed coordinate system. (So the X-axis is pointing to the right.) The camera can revolve around the mesh in 1 degree increments.

The *zdrive count*, *image width*, and *image height* options listed above also apply to *vtk_write*. In addition, you can use these options to govern the output images.

output format = <format name>	choices are tiff, png, jpeg, ps and bmp (default is *tiff*)
output name = <file name>	base name of image file or files (*outfile*)
output frame start = <number>	first frame, between 0 and 360 (*0*)
output frame stop = <number>	last frame, between 0 and 360 (*0*)
output frame stride = <number>	the difference in degrees from one frame to the next (*1*)
output view up = <x y z>	the direction of "up" as camera points at mesh (*0 1 0*)

Other file formats

vtk_view was written to post-process *zdrive* runs, so it only reads Chaco or Exodus II/Nemesis meshes. If you are working with a different mesh-based file format, it is still possible that you could use *vtk_view* or *vtk_write* to view the partitions assigned to your mesh by some application using the Zoltan library. VTK at this point in time has readers for many different file formats. If VTK has a reader for your format, then modify the *read_mesh* function in *util/vtk_view.cpp* to use that reader.

You can then hard-code *vtk_view* to read your file, or you can modify *read_cmd_file* in *driver/dr_input.c* to accept a specification of your file type in addition to Chaco and Nemesis. If you do the latter you can create a zdrive-style input file in which to specify your file name and other visualization parameters.

Finally, you need to create text files listing each global ID you supplied to Zoltan, followed by the partition ID assigned by Zoltan, with only one global ID/partition ID pair per line. Name this file or files using the conventions used by the test drivers.

[Table of Contents | Next: Using the Test Script | Previous: Using the Test Drivers] </html

Zoltan Developer's Guide | *Next* | *Previous*

Appendix: Using the test script *test_zoltan*

The purpose of the Zoltan test script is to run the test driver *zdrive* (or *zfdrive*) on a set of test problems to verify that the Zoltan library works correctly. The script compares the output of actual runs with precomputed output. The assumption is that if the outputs are identical, then the current implementation is is likely to be correct. Small differences may occur depending on the architectures used; developers should examine the output and use their judgement in determining its correctness. It is strongly recommended that developers run *test_zoltan* to verify correctness before committing changes to existing code!

How to run *test_zoltan*

First make sure you have compiled the driver *zdrive* (or *zfdrive*). Then go to the Zoltan directory *Zoltan/tests* and type *test_zoltan* with suitable options as described below. This will run the test script in interactive mode. The output from the driver will be sent to *stdout* and *stderrstdout* and *stderr* with a summary of results. The summary of results is also saved in a log file. If an error occured, look at the log file to find out what went wrong. The script currently assumes that runs are deterministic and reproducible across all architectures, which is not necessarily true. Hence false alarms may occur.

Syntax

test_zoltan [-arch *arch-type*] [-cmd *command*] [other options as listed below]

It is required to use either the -arch or the -cmd option. The other arguments are optional.

Options:

-arch *arch-type*	The architecture on which the driver is to run. For a list of currently supported architectures, type *test_zoltan* with no arguments.
-cmd *command*	The *command* is the command that the script uses to launch the driver. One must include an option to specify the number of processors as part of the command. Use quotes appropriately; for example, *-cmd 'mpirun -np'*. Default settings have been provided for all the supported architectures.
-logfile *filename*	The name of the log file. The default is *test_zoltan.log*. If an old log file exists, it will be moved to *test_zoltan.log.old*.
-no_parmetis	Do not run any ParMETIS methods.
-no_nemesis	Do not run test problems in Nemesis format.
-no_chaco	Do not run test problems in Chaco format.
-yes_fortran	Run the Fortran90 driver *zfdrive* instead of *zdrive*.

The default behavior is to run *zdrive* all methods on all types of input format.

Test problems

The test problems are included in subdirectories of the *Zoltan/test* directory. Problems using Chaco input files are in subdirectories *ch_**; problems using Nemesis input files are in subdirectories *nem_**. Please see the README files located in each test directory for more details on these test problems.

Load balancing methods

Many different load-balancing methods are currently tested in *test_zoltan*. Input files for the methods are found in the test problem subdirectories. The input files are named *zdrive.inp.*<*method*>, where <*method*> indicates which load-balancing method is passed to Zoltan. To run only a subset of the methods, edit the *test_zoltan* script manually; searching for "rcb" shows which lines of the script must be changed.

Number of processors

The script *test_zoltan* runs each test problem on a predetermined number of processors, currently ranging from 3 to 9.

[Table of Contents | Next: RCB | Previous: Visualization of Geometric Partitionings]

Appendix: Recursive Coordinate Bisection (RCB)

Outline of Algorithm

The implementation of Recursive Coordinate Bisection (RCB) in Zoltan is due to Steve Plimpton of Sandia National Laboratories and was modified by Matt St. John and Courtenay Vaughan. In this implementation of RCB, the parallel computer is first divided into two pieces and then the computational domain is divided into two pieces such that the proportion of work in each piece is the same as the proportion of computational power. The division of the parallel machine is done by a subroutine which is part of the support for heterogenous architectures that is being built into the Zoltan library. This process is repeated recursively on each subdomain and its associated part of the computer. Each of these divisions are done with a cutting plane that is orthogonal to one of the coordinate axes.

At each of these stages, each subdomain of processors and the objects that are contained on those processors are divided into two sets based on which side of the cutting plane each object is on. Either or both of these sets may be empty. On each processor, the set of objects which are on the same side of the cut as the processor are retained by the processor, while the other objects are sent to processors on the other side of the cut. In order to minimize the maximum memory usage in each set of processors, the objects that are being sent to each set of processors are distributed such that each each processor in a set has about the same number of objects after the objects from the other set of processors are sent. In the case when a processor has more objects that it will retain than the average number of objects that the rest of the processors have in its set, then that processor will not receive any objects. Thus each processor may send and receive objects from several (or no) processors in the other set. The process of determining which outgoing objects are sent to which processors is determined in the subroutine **Zoltan_Create_Proc_List**. Once this new distribution of objects is determined, the **unstructured communication package** in Zoltan is used to determine which processors are going to receive which objects and actually move the objects.

For applications that wish to add more objects to the decomposition at a later time (e.g., through **Zoltan_LB_Box_Assign** or **Zoltan_LB_Point_Assign**), information to do this can be retained during the decomposition phase. This information is kept if the parameter KEEP_CUTS is set during the decomposition (see the RCB section in the **Zoltan User's Guide**). This information about the decomposition can be thought of as a tree with the nodes which have children representing the cut information and the nodes with no children representing processors. An object is dropped through the tree starting with the root node and uses the cut information at each node it encounters to determine which subtree it traverses. When it reaches a terminal node, the node contains the processor number that the object belongs to. The information to construct the tree is saved during the decomposition. At each step in the decomposition, when each set is divided into two sets, the set with the lowest numbered processor is designated to be the left set and the information about the cut is stored in the lowest numbered processor in the other set of processors which is the right set. As a result of this process, each processor will store information for, at most, one cut, since once a processor stores information about a cut, by being the lowest numbered processor in the right set, it will always be in a left set after each subsequent cut since it will be the lowest numbered processor in the set being cut and the set it is put into will be the left set. The processor which stores the cut information also stores the root node as its parent. After the end of the division process, all of the information is collected onto all of the processors. The parent information is then used to establish the leaf information for the parent. When this information is gathered, the tree structure is stored in arrays with the array position determined by the processor number that was storing the information. There is an array which stores the position of the cut information for the left set and one for the right set as well as arrays for the cut information. Given that the lowest numbered processor after a cut is in the left set, the cut information is stored in the right set, and there is one fewer cut than the total number of processors, processor 0 has no cut information, so the 0 position of the right set array is empty and is used to store the position in the array that the first cut is stored. When this information is used to process an object, array position 0 in the right set array is used to determine the array position of the first cut. From there, which side of the cut the object is on is determined and that information is used to determine which cut to test the object against next. This process is repeated recursively until a terminal node is encountered which contains the processor number that the object belongs to.

When the parameter RCB_REUSE is specified, the RCB algorithm attempts to use information from a previous RCB decomposition to generate an "initial guess" at the new decomposition. For problems that change little between invocations of RCB, using RCB_REUSE can reduce the amount of data movement in RCB, improving the performance of the algorithm. When RCB_REUSE is true,the coordinates of all objects obtained through query functions are passed through **Zoltan_LB_Point_Assign** to determine their processor assignment in the previous RCB decomposition. The information for the objects is then sent to the new processor assignments using the unstructured

communication utilities to generate an initial condition matching the output of the previous RCB decomposition. The normal RCB algorithm is then applied to this new initial condition.

Data Structure Definitions

There are three major data structures in RCB and they are defined in *rcb/rcb.h* and *rcb/shared.h*. The points which are being load balanced are represented as a structure *Dot_Struct* which contains the location of the point, its weight, and its originating processor number. The nodes on the decomposition tree are represented by the structure *rcb_tree* which contains the position of the cut, the dimension that the cut is perpendicular to, and the node's parent and two children (if they exist) in the tree. The structure *RCB_Struct* is the RCB data structure which holds pointers to all of the other data structures needed for RCB. It contains an array of *Dot_Struct* to represent the points being load balanced, global and local IDs for the points, and an array of *rcb_tree* (whose length is the number of processors) which contains the decomposition tree.

Parameters

The parameters used by RCB and their default values are described in the RCB section of the **Zoltan User's Guide**. These can be set by use of the **Zoltan_RCB_Set_Param** subroutine in the file *rcb/rcb.c*.

When the parameter REDUCE_DIMENSIONS is specified, the RCB algorithm will perform a lower dimensional partitioning if the geometry is found to be degenerate. More information on detecting degenerate geometries may be found in another section.

Main Routine

The main routine for RCB is **Zoltan_RCB** in the file *rcb/rcb.c*.

[Table of Contents | Next: Recursive Inertial Bisection (RIB) | Previous: Using the Test Script]

Zoltan Developer's Guide | *Next* | *Previous*

Appendix: Recursive Inertial Bisection (RIB)

Outline of Algorithm

The implementation of Recursive Inertial Bisection (RIB) in Zoltan is due due to Bruce Hendrickson and Robert Leland of Sandia National Laboratories for use in Chaco and was modified by Courtenay Vaughan. RIB is an algorithm similar to RCB (see the **appendix on RCB** for a description of RCB) in that it uses the coordinates of the objects to be balanced to do the load balancing. Similarly to RCB, the domain is recursively divided into two pieces until the number of subdomains needed is reached. In each stage of the division, the direction of the principle axis of the domain to be divided is calculated by determining an eigenvector of the inertial matrix. This direction vector is used to define a normal to a plane which is used to divide the domain into two pieces. This process is repeated until the desired number of subdomains is reached.

The communication of objects being divided is handled by the same routine as is used by **RCB**. For applications which wish to add more objects to the decomposition at a later time (e.g., through **Zoltan_LB_Box_Assign** or **Zoltan_LB_Point_Assign**), information to do this can be retained during the decomposition phase. This information is kept if the parameter KEEP_CUTS is set during the decomposition. The process is similar to that used for RCB, but the information kept is different. For each RIB cut, the center of mass of the subdomain which is cut, the direction vector, and a distance from the center of mass to the cutting plane have to be saved.

Data Structure Definitions

There are three major data structures in RIB and they are defined in *rcb/rib.h* and *rcb/shared.h*. The points which are being load balanced are represented as a structure *Dot_Struct* which contains the location of the point, its weight, and the originating processor's number. The nodes on the decomposition tree are represented by the structure *rib_tree* which contains the position of the cut, the center of mass of the subdomain which is being cut, the direction vector of the principle axis of the subdomain, and the node's parent and two children (if they exist) in the tree. The structure RIB_Struct is the RIB data structure which holds pointers to all of the other data structures needed for RIB. It contains an array of *Dot_Struct* to represent the points being load balanced, global and local IDs of the points, an array of *rib_tree* (whose length is the number of processors) which contains the decomposition tree, and the dimension of the problem.

Parameters

The parameters used by RIB and their default values are described in the RIB section of the **Zoltan User's Guide**. These can be set by use of the **Zoltan_RIB_Set_Param** subroutine in the file *rcb/rib.c*.

When the parameter REDUCE_DIMENSIONS is specified, the RIB algorithm will perform a lower dimensional partitioning if the geometry is found to be degenerate. More information on detecting degenerate geometries may be found in another section.

Main Routine

The main routine for RIB is **Zoltan_RIB** in the file *rcb/rib.c*.

[Table of Contents | Next: ParMETIS and Jostle | Previous: Recursive Coordinate Bisection (RCB)]

Appendix: ParMETIS and Jostle

Overview of structure (algorithm)

This part of Zoltan provides an interface to various graph-based load-balancing algorithms. Currently two libraries are supported: ParMETIS and Jostle. Each of these libraries contain several algorithms.

Interface algorithm

The structure of the code is as follows: Each package (ParMETIS, Jostle) has its own wrapper routine that performs initialization and sets parameters. The main routine is **Zoltan_ParMetis_Jostle,** which constructs an appropriate graph data structure using Zoltan's query functions. After the graph structure has been constructed, the appropriate library is called and the import/export list is created and returned.

Please note that ParMETIS and Jostle are not integral parts of Zoltan. These libraries must be obtained and installed separately. (ParMETIS may be bundled with Zoltan, but it is an independent package developed at Univ. of Minnesota.) Zoltan merely provides an interface to these libraries.

The most complex task in the interface code is the construction of the graph data structure. This structure is described in the next section. The routine uses the Zoltan query functions to get a list of objects and edges on each processor. Each object has a unique global ID which is mapped into a unique (global) number between 1 and *n*, where *n* is the total number of objects. The construction of the local (on-processor) part of the graph is straightforward. When an edge goes between objects that reside on different processors, global communication is required. We use Zoltan's unstructured communication library for this. A hash function (**Zoltan_Hash**) is used to efficiently map global IDs to integers. The graph construction algorithm has parallel complexity $O(max_j \{n_j+m_j+p\})$, where n_j is the number of objects on processor j, m_j is the number of edges on processor j, and p is the number of processors.

One other feature of the interface code should be mentioned. While Zoltan allows objects and edges to have real (float) weights, both ParMETIS and Jostle currently require integer weights. Therefore, Zoltan first checks if the object weights are integers. If not, the weights are automatically scaled and rounded to integers. The scaling is performed such that the weights become large integers, subject to the constraint that the sum of (any component of) the weights is less than a large constant MAX_WGT_SUM < INT_MAX. The scaled weights are rounded up to the nearest integer to ensure that nonzero weights never become zero. Note that for multidimensional weights, each weight component is scaled independently. (The source code is written such that this scaling is simple to change.)

Currently Zoltan constructs and discards the entire graph structure every time a graph-based method (ParMETIS or Jostle) is called. Incremental update of the graph structure may be supported in the future.

The graph construction code in **Zoltan_ParMetis_Jostle** can also be used to interface with other graph-based algorithms. Please contact the Zoltan developers if you have a parallel partitioning or load-balancing code and would like assistance with interfacing it to Zoltan.

Algorithms used in ParMETIS and Jostle libraries

There are two main types of algorithms used in ParMETIS and Jostle. The first is multilevel graph partitioning. The main idea is to take a large graph and construct a sequence of smaller and simpler graphs that in some sense approximate the original graph. When the graph is sufficiently small it is partitioned using some other method. This smallest graph and the corresponding partition is then propagated back through all the levels to the original graph. A popular local refinement strategy known as Kernighan-Lin is employed at some or every level.

The second main strategy is diffusion. This method assumes that an initial partition (balance) is given, and load balance is achieved by repeatedly moving objects (nodes) from partitions (processors) that have too heavy load to neighboring partitions (processors) with too small load.

For further details about the algorithms in a specific library, please refer to the documentation that is distributed with that library.

Data structures

We use the ParMETIS parallel graph structure. This is implemented using 5 arrays:

1. *vtxdist*: gives the distribution of the objects (vertices) to processors
2. *xadj*: indices (pointers) to the *adjncy* array
3. *adjncy*: neighbor lists
4. *adjwgt*: edge weights
5. *vwgt*: vertex (object) weights

The *vtxdist* array is duplicated on all processors, while the other arrays are local.
For more details, see the ParMETIS User's Guide.

Parameters

Zoltan supports the most common parameters in ParMETIS and Jostle. These parameters are parsed in the package-specific wrapper routine (**Zoltan_ParMetis** or **Zoltan_Jostle**) and later passed on to the desired library via **Zoltan_ParMetis_Jostle**.

In addition, Zoltan has one graph parameter of its own: CHECK_GRAPH. This parameter is set in **Zoltan_ParMetis_Jostle** and specifies the amount of verification that is performed on the constructed graph. For example, it is required that the graph is symmetric and that the weights are non-negative.

Main routine

The main routine is **Zoltan_ParMetis_Jostle** but it should always be accessed through either **Zoltan_ParMetis** or **Zoltan_Jostle**.

[Table of Contents | Next: Hypergraph Partitioning | Previous: Recursive Inertial Bisection (RIB)]

Appendix: Hypergraph Partitioning

Hypergraph partitioning is a useful partitioning and load balancing method when connectivity data is available. It can be viewed as a more sophisticated alternative to the traditional graph partitioning.

A hypergraph consists of vertices and hyperedges. A hyperedge connects one or more vertices. A graph is a special case of a hypergraph where each edge has size two (two vertices). The hypergraph model is well suited to parallel computing, where vertices correspond to data objects and hyperedges represent the communication requirements. The basic partitioning problem is to partition the vertices into *k* approximately equal sets such that the number of cut hyperedges is minimized. Most partitioners (including Zoltan-PHG) allows a more general model where both vertices and hyperedges can be assigned weights. It has been shown that the hypergraph model gives a more accurate representation of communication cost (volume) than the graph model. In particular, for sparse matrix-vector multiplication, the hypergraph model **exactly** represents communication volume. Sparse matrices can be partitioned either along rows or columns; in the row-net model the columns are vertices and each row corresponds to an hyperedge, while in the column-net model the roles of vertices and hyperedges are reversed.

Zoltan contains a native parallel hypergraph partitioner, called PHG (Parallel HyperGraph partitioner). In addition, Zoltan provides access to PaToH, a serial hypergraph partitioner. Note that PaToH is not part of Zoltan and should be obtained separately from the PaToH web site. Zoltan-PHG is a fully parallel multilevel hypergraph partitioner. For further technical description, see [Devine et al, 2006].

Algorithm:

The algorithm used is multilevel hypergraph partitioning. For coarsening, several versions of inner product (heavy connectivity) matching are available. The refinement is based on Fiduccia-Mattheysis (FM) but in parallel it is only an approximation.

Parallel implementation:

A novel feature of our parallel implementation is that we use a 2D distribution of the hypergraph. That is, each processor owns partial data about some vertices and some hyperedges. The processors are logically organized in a 2D grid as well. Most communication is limited to either a processor row or column. This design should allow for good scalability on large number of processors.

Data structures:

The hypergraph is the most important data structure. This is stored as a compressed sparse matrix. Note that in parallel, each processor owns a local part of the global hypergraph (a submatrix of the whole matrix). The hypergraph data type is *struct HGraph*, and contains information like number of vertices, hyperedges, pins, compressed storage of all pins, optional vertex and edge weights, pointers to relevant communicators, and more. One cryptic notation needs an explanation: The arrays *hindex, hvertex* are used to look up vertex info given a hyperedge, and *vindex, vedge* are used to look up hyperedge info given a vertex. Essentially, we store the hypergraph as a sparse matrix in both CSR and CSC formats. This doubles the memory cost but gives better performance. The data on each processor is stored using local indexing, starting at zero. In order to get the global vertex or edge number, use the macros *VTX_LNO_TO_GNO* and *EDGE_LNO_TO_GNO*. These macros will look up the correct offsets (using the dist_x and dist_y arrays). Note that *phg->nVtx* is always the local number of vertices, which may be zero on some processors.

Parameters:

In the User's Guide, only the most essential parameters have been documented. There are several other parameters, intended for developers and perhaps expert "power" users. We give a more complete list of all parameters below. Note that these parameters *may change in future versions!*
For a precise list of parameters in a particular version of Zoltan, look at the source code (phg.c).

Method String:	**HYPERGRAPH**
Parameters:	

HYPERGRAPH_PACKAGE	PHG (parallel) or PaToH (serial)
CHECK_HYPERGRAPH	Check if input data is valid. (Slows performance;intended for debugging.)
PHG_OUTPUT_LEVEL	Level of verbosity; 0 is silent.
PHG_FINAL_OUTPUT	Print stats about final partitioning? (0/1)
PHG_NPROC_VERTEX	Desired number of processes in the vertex direction (for 2D internal layout)
PHG_NPROC_HEDGE	Desired number of processes in the hyperedge direction (for 2D internal layout)
PHG_COARSENING_METHOD	The method to use in matching/coarsening; currently these are available. *agg* - agglomerative inner product matching (a.k.a. heavy connectivity matching) *ipm* - inner product matching (a.k.a. heavy connectivity matching) *c-ipm* - column ipm; faster method based on ipm within processor columns *a-ipm* - alternate between fast method (l-ipm) and ipm *l-ipm* - local ipm on each processor. Fastest option but often gives poor quality. *h-ipm* - hybrid ipm that uses partial c-ipm followed by ipm on each level
PHG_COARSENING_LIMIT	Number of vertices at which to stop coarsening.
PHG_VERTEX_VISIT_ORDER	Ordering of vertices in greedy matching scheme: 0 - random 1 - natural order (as given by the query functions) 2 - increasing vertex weights 3 - increasing vertex degree 4 - increasing vertex degree, weighted by pins
PHG_EDGE_SCALING	Scale edge weights by some function of size of the hyperedges: 0 - no scaling 1 - scale by 1/(size-1) [absorption scaling] 2 - scale by 2/((size*size-1)) [clique scaling]
PHG_VERTEX_SCALING	Variations in "inner product" similarity metric (for matching): 0 - Euclidean inner product: <x,y> 1 - cosine similarity: <x,y>/(|x|*|y|) 2 - <x,y>/(|x|^2 * |y|^2) 3 - scale by sqrt of vertex weights 4 - scale by vertex weights
PHG_COARSEPARTITION_METHOD	Method to partition the coarsest (smallest) hypergraph; typically done in serial: *random* - random *linear* - linear (natural) order *greedy* - greedy method based on minimizing cuts *auto* - automatically select from the above methods (in parallel, the processes will do different methods)
PHG_REFINEMENT_METHOD	Refinement algorithm: *fm* - two-way approximate FM *none* - no refinement
PHG_REFINEMENT_LOOP_LIMIT	Loop limit in FM refinement. Higher number means more refinement.
PHG_REFINEMENT_MAX_NEG_MOVE	Maximum number of negative moves allowed in FM.
PHG_BAL_TOL_ADJUSTMENT	Controls how the balance tolerance is adjusted at each level of bisection.
PHG_RANDOMIZE_INPUT	Randomize layout of vertices and hyperedges in internal parallel 2D layout? (0/1)
PHG_EDGE_WEIGHT_OPERATION	Operation to be applied to edge weights supplied by different processes for the same hyperedge: *add* - the hyperedge weight will be the sum of the supplied weights *max* - the hyperedge weight will be the maximum of the supplied weights *error* - if the hyperedge weights are not equal, Zoltan will flag an error, otherwise the hyperedge weight will be the value returned by the processes
EDGE_SIZE_THRESHOLD	Ignore hyperedges greater than this fraction times number of vertices.
PATOH_ALLOC_POOL0	Memory allocation for PaToH; see the PaToH manual for details.
PATOH_ALLOC_POOL1	Memory allocation for PaToH; see the PaToH manual for details.

Default values:

HYPERGRAPH_PACKAGE = PHG
CHECK_HYPERGRAPH = 0
PHG_OUTPUT_LEVEL=0
PHG_FINAL_OUTPUT=0
PHG_REDUCTION_METHOD=ipm
PHG_REDUCTION_LIMIT=100
PHG_VERTEX_VISIT_ORDER=0
PHG_EDGE_SCALING=0
PHG_VERTEX_SCALING=0
PHG_COARSEPARTITION_METHOD=greedy
PHG_REFINEMENT_METHOD=fm
PHG_REFINEMENT_LOOP_LIMIT=10
PHG_REFINEMENT_MAX_NEG_MOVE=100
PHG_BAL_TOL_ADJUSTMENT=0.7
PHG_RANDOMIZE_INPUT=0
PHG_EDGE_WEIGHT_OPERATION=max
EDGE_SIZE_THRESHOLD=0.25
PATOH_ALLOC_POOL0=0
PATOH_ALLOC_POOL1=0

Required Query Functions:

ZOLTAN_NUM_OBJ_FN
ZOLTAN_OBJ_LIST_FN or **ZOLTAN_FIRST_OBJ_FN**/**ZOLTAN_NEXT_OBJ_FN** pair
ZOLTAN_HG_SIZE_CS_FN
ZOLTAN_HG_CS_FN

Optional Query Functions:

ZOLTAN_HG_SIZE_EDGE_WTS_FN
ZOLTAN_HG_EDGE_WTS_FN

It is possible to provide the graph query functions instead of the hypergraph queries, though this is not recommended. If only graph query functions are registered, Zoltan will automatically create a hypergraph from the graph, but some information (specifically, edge weights) will be lost.

[Table of Contents | Next: Refinement Tree Partitioning | Previous: ParMetis]

Zoltan Developer's Guide | *Next* | *Previous*

Appendix: Refinement Tree

Overview of structure (algorithm)

The refinement tree based partitioning algorithm was developed and implemented by William Mitchell of the National Institute of Standards and Technology. It is similar to the Octree method except that it uses a tree representation of the refinement history instead of a geometry based octree. The method generates a space filling curve which is cut into *K* appropriately-sized pieces to define contiguous partitions, where the size of a piece is the sum of the weights of the elements in that piece. *K*, the number of partitions, is not necessarily equal to *P*, the number of processors. It is an appropriate load balancing method for grids that are generated by adaptive refinement when the refinement history is available. This implementation consists of two phases: the construction of the refinement tree, and the definition of the partitions.

Refinement Tree Construction

The refinement tree consists of a root node and one node for each element in the refinement history. The children of the root node are the elements of the initial coarse grid. The children of all other nodes are the elements that were formed when the parent element was refined. Upon first invocation, the refinement tree is initialized. This creates the root node and initializes a hash table that maps global IDs into nodes of the refinement tree. It also queries the user for the elements of the initial grid and creates the children of the root node. Unless the user provides the order through which to traverse the elements of the initial grid, a path is determined through the initial elements along with the "in" vertex and "out" vertex of each element, i.e., the vertices through which the path passes to move from one element to the next. This path can be determined by a Hilbert space filling curve, Sierpinski space filling curve (triangles only), or an algorithm that attempts to make connected partitions (connectivity is guaranteed for triangles and tetrahedra). The refinement tree is required to have all initial coarse grid elements, not just those that reside on the processor. However, this requirement is not imposed on the user; a communication step fills in the elements from other processors. This much of the tree persists throughout execution of the program. The remainder of the tree is reconstructed on each invocation of the refinement tree partitioner. The remainder of the tree is built through a tree traversal. At each node, the user is queried for the children of the corresponding element. If there are no children, the user is queried for the weight of the element. If there are children, the order of the children is determined such that a tree traversal produces a space filling curve. The user indicates what type of refinement was used to produce the children (bisection of triangles, quadrasection of quadrilaterals, etc.). For each supported type of refinement, a template based ordering is imposed. The template also maintains an "in" and "out" vertex for each element which are used by the template to determine the beginning and end of the space filling curve through the children. If the refinement is not among the types supported by templates, an exhaustive search is performed to find an appropriate order, unless the user provides the order.

Partition algorithm

The algorithm that determines the partitions uses four traversals of the refinement tree. The first two traversals sum the weights in the tree. In the first traversal, each node gets the sum of the weights of all the descendant nodes that are assigned to this processor. The processors then exchange information to fill in the partial sums for the leaf elements that are not owned by this processor. (Note that an unowned leaf on one processor may be the root of a large subtree on another processor.) The second traversal completes the summation of the weights. The root now has the sum of all the weights, which, in conjunction with an array of relative partition sizes, determines the desired weight of each partition. Currently the array of partition sizes are all equal, but in the future the array will be input to reflect heterogeneity in the system. The third traversal determines the partitioning by adding subtrees to a partition until the size of the partition meets the desired weight, and counts the number of elements to be exported. Finally, the fourth traversal constructs the export list.

Data structures

The implementation of the refinement tree algorithm uses three data structures which are contained in *reftree/reftree.h*. *Zoltan_Reftree_data_struct* is the structure pointed to by *zz->LB.Data_Structure*. It contains a pointer to the refinement tree root and a pointer to the hash table. *Zoltan_Reftree_hash_node* is an entry in the hash table. It consists of a global ID, a pointer to a refinement tree node, and a "next" pointer from which linked lists at each table entry are constructed to handle collisions. *Zoltan_Reftree_Struct* is a node of the refinement tree. It contains the global ID, local ID, pointers to the children, weight and summed weights, vertices of the element, "in" and "out" vertex, a flag to indicate if this element is assigned to this processor, and the new partition number.

Parameters

There are two parameters. REFTREE_HASH_SIZE determines the size of the hash table. REFTREE_INITPATH determines which algorithm to use to find a path through the initial elements. Both are set by **Zoltan_Reftree_Set_Param** in the file *reftree/reftree_build.c*.

Main routine

The main routine is **Zoltan_Reftree_Part** in file *reftree/reftree_part.c*.

[Table of Contents | Next: Hilbert Space-Filling Curve (HSFC) | Previous: Hypergraph Partitioning]

Appendix: Hilbert Space Filling Curve (HSFC)

Outline of Algorithm

This partitioning algorithm is loosely based on the 2D & 3D Hilbert tables used in Octree and on the BSFC partitioning implementation by Andrew C. Bauer, Department of Engineering, State University of New York at Buffalo, as his summer project at SNL in 2001. Please refer to the corresponding section in the Zoltan User's guide, **Hilbert Space Filling Curve (HSFC),** for information about how to use this module and its parameters. Note: the partitioning, point assign and box assign functions in this code module can be trivially extended to any space filling curve for which we have a state table definition of the curve.

First, the weights and inverse Hilbert coordinates for each object are determined. If the objects do not have weights, unit weights are assigned. If the objects have multiple weights, only the first weight is currently used. The smallest axis-aligned box is found that contains all of the objects using their two or three dimensional spatial coordinates. This bounding box is slightly expanded to ensure that all objects are strictly interior to the boundary surface. The bounding box is necessary in order to calculate the inverse Hilbert Space Filling curve coordinate. The bounding box is used to scale the problem coordinates into the [0,1]^d unit volume (d represents the number of dimensions in the problem space.) The inverse Hilbert coordinate is calculated and stored as a double precision floating point value for each object. This code works on problems with one, two or three dimensions (the 1-D Inverse Hilbert coordinate is simply the problem coordinate itself, after the bounding box scaling.)

The algorithm seeks to cut the unit interval into P segments containing equal weights of objects associated to the segments by their inverse Hilbert coordinates. The code allows a user vector to specify the desired fraction of the total weight to be assigned to each interval. Note, a zero weight fraction prevents any object being assigned to the corresponding interval. The unit interval is divided into N bins, N=k(P-1)+1, where k is a small positive constant.) Each bin has an left and right endpoint specifying the half-open interval [l,r) associated with the bin. The bins form a non-overlapping cover of [0,1] with the right endpoint of the last bin forced to include 1. The bins are of equal size on the first loop. (Hence each interval or part of the partition is a collection of bins.)

For each loop, an MPI_Allreduce call is made to globally sum the weights in each bin. This call also determines the maximum and minimum (inverse Hilbert) coordinate found in each bin. A greedy algorithm sums the weights of the bins from left to right until the next bin would cause an overflow for the current partition. This results in new partition of P intervals. The location of each cut (just before an "overflowing" bin) and the size of its "overflowing" bin are saved. The "overflowing" bin's maximum and minimum are compared to determine if the bin can be practically subdivided. (If the bin's maximum and minimum coordinates are too close relative to double precision resolution, the bin can not be practically subdivided.) If at least one bin can be further refined, then looping will continue. In order to prevent a systematic bias, the greedy algorithm is assumed to exactly satisfy the weight required by each partition.

Before starting the next loop, the P intervals are again divided into N bins. The P-1 "overflow" bins are each subdivided into k-1 equal bins. The intervals before and after these new bins determine the remaining bins. This process maintains a fixed number of bins. No bin is "privileged." Specifically, any bin is subject to later refinement, as necessary, on future loops.

The loop terminates when there is no need to further divide any "overflow" bin. A slightly different greedy algorithm is used to determine the final partition of P intervals from the N bins. In this case, when the next bin would cause an overflow, the tolerance is computed for both underfilling (excluding this last bin) and overfilling (including the last bin). The tolerance closest to the target tolerance is used to select the dividing point. The tolerance obtained at each dividing point is compared to the user's specified tolerance. An error is returned if the user's tolerance is not satisfied at any cut. After each cut is made, a correction is calculated as the ratio of the actual weight to the target weight used up to this point. This correction is made to the target weight for the next partition. This correction fixes the subsequent partitions when a "massive" weight object is on the border of a cut and its assignment creates an excessive imbalance.

Generally, the number of loops is small (proportional to log(number of objects)). A maximum of MAX_LOOPS is used to prevent an infinite looping condition. A user-defined function is used in the MPI_Allreduce call in order to simultaneously determine the sum, maximum, and minimum of each bin. The message length in the MPI_Allreduce is proportional to the P, the number of partitions.

Note, when a bin is encountered that satisfies more than two partitions, that bin is refined into a multiple of k-1 intervals which maintains a total of N bins.

Hilbert Transformations

The HSFC now uses table driven logic to convert from spatial coordinates (2 or 3 dimensions) (the Inverse Hilbert functions) and from the unit interval into spatial coordinates (Hilbert functions). In each case there are two associated tables: the data table and the state table. In all cases, the table logic can be extended to any required precision. Currently, the precision is determined for compatibility with the the double precision used in the partitioning algorithm.

The inverse transformation is computed by taking the highest order bit from each spatial coordinate and packing them together as 2 or 3 bits (as appropriate to the dimensionality) in the order xyz (or xy) where x is the highest bit in the word. The initial state is 0. The data table lookup finds the value at the column indexed by the xyz word and the row 0 (corresponding to the initial state value.) This data are the 3 (or 2) starting bits of the Hilbert coordinate. The next state value is found by looking up the corresponding element of the state table (xyz column and row 0.)

The table procedure continues to loop (using loop counter i, for example) until the required precision is reached. At loop i, the ith bits from each spatial dimension are packed together as the xyz column index. The data table lookup finds the element at column xyz and the row determined by the last state table value. This is appended to the Hilbert coordinate. The state table is used to find the next state value at the element corresponding to the xyz column and row equal to the last state value.

The inverse transformation is analogous. Here the 3 (or 2 in the 2-d case) bits of the Hilbert coordinate are extracted into a word. This word is the column index into the data table and the state value is the row. This word found in the data table is interpreted as the packed xyz bits for the spatial coordinates. These bits are extracted for each dimension and appended to that dimension's coordinate. The corresponding state table is used to find the next row (state) used in the next loop.

Point Assign

The user can use **Zoltan_LB_Point_Assign** to add a new point to the appropriate partition. The bounding box coordinates, the final partition, and other related information are maintained after partitioning if the KEEP_CUTS parameter is set by the user. The KEEP_CUTS parameter must be set by the user for Point Assign! The extended bounded box is used to compute the new point's inverse Hilbert coordinate. Then the routine performs a binary search on the final partition to determine the partition (interval) which includes the point. The routine returns the partition number assigned to that interval.

The Point Assign function now works for any point in space, even if the point is outside the original bounding box. If the point is outside the bounding box, it is first scaled using the same equations that scale the interior points into the unit volume. The point is projected onto the unit volume. For each spatial dimension, if the scaled coordinate is less than zero, it is replace by zero. If it is greater than one, it is replaced by one. Otherwise the scaled coordinate is directly used.

Box Assign

The user can use **Zoltan_LB_Box_Assign** to determine the partitions whose corresponding subdomains intersect the user's query box. Although very different in implementation, the papers by Lawder and King ("Querying Multi- dimensional Data Index Using the Hilbert Space-Filling Curve", 2000, etc.) were the original inspiration for this algorithm. The Zoltan_HSFC_Box_Assign routine primarily scales the user query region and determines its intersection with the Hilbert's bounding box. Points exterior to the bounding box get projected along the coordinate axis onto the bounding box. A fuzzy region is built around query points and lines to create the boxes required for the search. It also handles the trivial one-dimensional case. Otherwise it repeatedly calls the 2d and 3d query functions using the next highest partition's left end point to start the search. These query routines return the next point on the Hilbert curve to enter the query region. A binary search finds the partition associated with this point. The query functions are called one more time than the number of partitions that have points interior to the query region.

The query functions decompose the unit square (or cube) level by level like the Octree method. Each level divides the remaining region into quadrants (or octets in 3d). At each level, the quadrant with the smallest inverse Hilbert coordinate (that is, occurring first along the Hilbert curve) whose inverse Hilbert coordinate is equal or larger than the starting inverse Hilbert coordinate and which intersects with query region is selected. Thus, each level calculates the next 2 bits (3 bits in 3d) of the inverse Hilbert coordinate of the next point to enter the query region. No more than once per call to the query function, the function may backtrack to a nearest previous level that has another quadrant that intersects the query region and has a higher Hilbert coordinate.

In order to determine the intersection with the query region, the next 2 bits (3 in 3 dimensions) of the Hilbert transformation are also computed (by table lookup) at each level for the quadrant being tested. These bits are compared to the the bits resulting from the intersection of the query region with the region determined by the spatial coordinates computed to the precision of the previous levels.

If the user query box has any side (edge) that is "too small" (effectively degenerate in some dimension), it is replaced by a minimum value and the corresponding vertex coordinates are symmetrically expanded. This is refered to as a "fuzzy" region.

This function requires the KEEP_CUTS parameter to be set by the user. The Box Assign function now works for any box in space, even if it has regions outside the original bounding box. The box vertices are scaled and projected exactly like the points in the Point Assign function described above. However, to allow the search to use a proper volumn, projected points, lines, and planes are converted to a usable volume by the fuzzy region process described above.

This algorithm will work for any space filling curve. All that is necessary is to provide the tables (derieved from the curve's state transition diagram) in place of the Hilbert Space Filling Curve tables.

Data Structure Definitions

The data structures are defined in *hsfc/hsfc.h*. The objects being load balanced are represented by the *Dots* Structure which holds the objects spacial coordinates, the corresponding inverse Hilbert coordinate, the processor owning the object, and the object's weight(s). The *Partition* structure holds the left and right endpoints of the interval represented by this element of the partition and the index to the processor owning this element of the partition. The structure *HSFC_Data* holds the "persistant" data needed by the point assign and box assign routines. This includes the bounding box, the number of loops necessary for load balancing, the number of dimensions for the problem, a pointer to the function that returns the inverse Hilbert Space-Filling Curve coordinate, and the final Partition structure contents.

Parameters

The parameters used by HSFC and their default values are described in the HSFC section of the **Zoltan User's Guide**. These can be set by use of the **Zoltan_HSFC_Set_Param** subroutine in the file *hsfc/hsfc.c*.

When the parameter REDUCE_DIMENSIONS is specified, the HSFC algorithm will perform a lower dimensional partitioning if the geometry is found to be degenerate. More information on detecting degenerate geometries may be found in another section.

Main Routine

The main routine for HSFC is **Zoltan_HSFC** in the file *hsfc/hsfc.c*.

[Table of Contents | Next: Handling Degenerate Geometries | Previous: Refinement Tree]

Appendix: Handling Degenerate Geometries

The geometry processed by one of the geometric methods RCB, RIB, or HSFC may be degenerate. By this we mean it may have 3-dimensional coordinates but be essentially flat, or it may have 3 or 2-dimensional coordinates but be essentially a line in space. If we treat the geometry as a lower dimensional object for the purpose of partitioning, the result may be a more natural partitioning (one the user would have expected) and a faster run time.

The caller may set the **REDUCE_DIMENSIONS** parameter to TRUE in any of the three geometric methods if they want Zoltan to check for a degenerate condition and do a lower dimensional partitioning if such a condition if found. They may set the **DEGENERATE_RATIO** to specify how flat or thin a geometry must be to be considered degenerate.

Outline of Algorithm

All three geometric methods call **Zoltan_Get_Coordinates** to obtain the problem coordinates. If **REDUCE_DIMENSIONS** is TRUE, we check in this function to see if the geometry is degenerate. If it is, we transform the coordinates to the lower dimensional space, flag that the problem is now lower dimensional, and return the transformed coordinates. The RCB, RIB, or HSFC calculation is performed on the new coordinates in the lower dimensional space.

If **KEEP_CUTS** is TRUE, the transformation is saved so that in **Zoltan_LB_Box_Assign** or **Zoltan_LB_Point_Assign** the coordinates can be transformed before the assignment is calculated. If **RCB_REUSE** is TRUE in the RCB method, the transformation is also saved. On re-partitioning, we can do some simple tests to see if the degeneracy condition has changed before completely re-calculating the coordinate transformation.

To determine if the geometry is degenerate, we calculate the same inertial matrix that is calculated for RIB, except that we ignore vertex weights. The 3 orthogonal eigenvectors of the inertial matrix describe the three primary directions of the geometry. The bounding box oriented in these directions is tested for degeneracy. In particular (for a 3 dimensional geometry) if the length of the longest side divided by the length of the shortest side exceeds the DEGENERATE_RATIO, we consider the geometry to be flat. If in addition, the length longest side divided by the length of the middle side exceeds the DEGENERATE_RATIO, we consider the geometry to be essentially a line.

If a 3 dimensional geometry is determined to be flat, we transform coordinates to a coordinate system where the XY plane corresponds to the oriented bounding box, and project all coordinates to that plane. These X,Y coordinates are returned to the partitioning algorithm, which performs a two dimensional partitioning. Similarly if the geometry is very thin, we transform coordinates to a coordinate system with the X axis going through the bounding box in it's principal direction, and project all points to that axis. Then a 1 dimensional partitioning is performed.

There is a small problem in calculating **Zoltan_LB_Box_Assign** when the partitioning was performed on transformed geometry. The caller provides the box vertices in problem coordinates, but the partitioning was calculated in transformed coordinates. When the vertices are transformed, they are in general no longer the vertices of an axis-aligned box in the new coordinate system. The **Box_Assign** calculation requires an axis-aligned box, and so we use the bounding box of the transformed vertices. The resulting list of processes or partitions intersecting the box may therefore contain some processes or partitions which actually do not intersect the box in problem coordinates, however it will not omit any.

Data Structure Definitions

The transformation is stored in a **Zoltan_Transform_Struct** structure which is defined in *zz/zz_const.h*. It is saved as part of the algorithm specific information stored in the **LB.Data_Structure** field of the **Zoltan_Struct**. The flag that indicates whether the geometry was found to be degenerate is the **Target_Dim** field of this structure.

To use the degenerate geometry detection capability from a new geometric method, you would add a **Zoltan_Transform_Struct** structure to the algorithm specific data structure, add code to **Zoltan_Get_Coordinates** to look for it, and check the **Target_Dim** field on return to see if the problem dimension was reduced. You would also need to include the coordinate transformation in your Box_Assign and Point_Assign functionality.

[Table of Contents | Previous: Hibert Space Filling Curve (HSFC)]