

SAND REPORT

SAND2004-0365

Unlimited Release

Printed February 2004

LDRD Report: Parallel Repartitioning for Optimal Solver Performance

Erik Boman, Karen Devine, Robert Heaphy, Bruce Hendrickson, Mike Heroux, and
Robert Preis

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.doe.gov/bridge>

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: <http://www.ntis.gov/ordering.htm>



SAND2004-0365
Unlimited Release
Printed February 2004

LDRD Report: Parallel Repartitioning for Optimal Solver Performance

Erik Boman, Karen Devine, Robert Heaphy and Bruce Hendrickson

Discrete Algorithms and Mathematics Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1111

Mike Heroux

Computational Mathematics and Algorithms Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1110

Robert Preis

Computer Science, Electrical Engineering and Mathematics
University of Paderborn
Warburger StraÙe 100
D-33098, Paderborn
Germany

Abstract

We have developed infrastructure, utilities and partitioning methods to improve data partitioning in linear solvers and preconditioners. Our efforts included incorporation of data repartitioning capabilities from the Zoltan toolkit into the Trilinos solver framework, (allowing dynamic repartitioning of Trilinos matrices); implementation of efficient distributed data directories and unstructured communication utilities in Zoltan and Trilinos; development of a new multi-constraint geometric partitioning algorithm (which can generate one decomposition that is good with respect to multiple criteria); and research into hypergraph partitioning algorithms (which provide up to 56% reduction of communication volume compared to graph partitioning for a number of emerging applications). This report includes descriptions of the infrastructure and algorithms developed, along with results demonstrating the effectiveness of our approaches.

1	Introduction.....	7
2	Parallel Data Redistribution in The Petra Object Model.....	9
2.1	Parallel Data Redistribution.....	9
2.1.1	PDR and Sparse Matrix Calculations.....	10
2.2	An Object-oriented approach to PDR.....	10
2.2.1	Communication Classes.....	10
2.2.2	Element Spaces.....	11
2.2.3	Distributed Objects.....	13
2.2.4	Import and Export Operations.....	15
2.2.5	Import/Export Uses.....	15
2.3	Implementation Issues for PDR.....	17
2.3.1	Computing Importers and Exporters.....	17
2.3.2	Implementing the <i>DistObject</i> Base Class.....	18
2.4	Parallel Data Redistribution Results.....	19
2.4.1	Robust Calculations and Dynamic Load Balancing.....	19
2.4.2	Epetra-Zoltan Interface.....	20
2.4.3	Partitioning for Highly Convective Flows.....	20
2.5	Conclusions.....	20
3	Distributed Data Directory.....	21
3.1	Distributed Data Directory Usage.....	21
3.2	Distributed Data Directory Implementation.....	22
3.3	Distributed Data Directory Functions.....	23
4	Unstructured Communication.....	24
5	Multicriteria partitioning and load balancing.....	27
5.1	Introduction.....	27
5.2	Linear partitioning and bisection.....	27
5.3	Multiconstraint or multiobjective?.....	28
5.4	Multiobjective methods.....	29
5.5	Multiobjective bisection.....	29
5.6	Implementation in Zoltan.....	31
5.7	Empirical results.....	32
5.8	Future work.....	34
6	Hypergraph Partitioning.....	36
6.1	Multilevel Hypergraph Partitioning.....	38
6.1.1	Terminology.....	39
6.1.2	Coarsening Strategies.....	40
6.1.3	Augmentation Strategies.....	44
6.1.4	Scaling Strategies.....	45
6.1.5	Global (Coarse) Partitioning Strategies.....	45
6.1.6	Refinement Strategies.....	47
6.2	Hypergraph Partitioning Results.....	48
6.2.1	Design of Experiment.....	48
6.2.2	Hypergraph Partitioning in Sandia Applications.....	52
6.3	Future Work.....	55
7	Conclusions and Future Work.....	56

8 References..... 57

1 Introduction

The distribution of data across distributed-memory computers significantly affects the performance of linear solvers. Often, data and work distribution for solvers is dictated solely by the application as solvers simply use the application's distribution of the matrices and vectors. Application developers typically distribute their data with the goal of minimizing load imbalance and communication costs within the application. Both of these goals are important for solvers. However, there are other goals that are equally or even more important, especially for difficult problems. These goals include optimal grouping of matrix entries for robust preconditioning, and minimizing load imbalance in the presence of overlapping domains.

In the LDRD "Parallel Repartitioning for Optimal Solver Performance" (FY01-03), we improved Sandia's capabilities to effectively partition linear systems for parallel computation. Our efforts were based in two libraries: the Trilinos solver framework [24] and the Zoltan parallel data management toolkit [7, 18]. These two libraries are used by more than a dozen Sandia applications; thus, the technology developed in the LDRD can have immediate impact across Sandia's computational science communities.

Trilinos [24] is a common framework for all current and future solver projects. It provides state-of-the-art, robust, scalable solvers, an object-oriented application programmer interface (API) for users of algebraic software, and a common set of software components for solver development. Using Trilinos, application developers can easily construct and manipulate vectors, matrices and other algebraic objects in a parallel environment, solve eigenproblems, and solve systems of equations. The Petra linear algebra services package [25] provides the underlying foundation for all Trilinos solvers. It includes the fundamental data structures, construction routines and services that are required for serial and parallel linear algebra libraries, and, thus, was the target package for development under the LDRD.

The Zoltan toolkit [7, 18] is a general-purpose library providing dynamic load balancing and related services to parallel applications. It includes graph-based, geometric, and tree-based load-balancing algorithms. Zoltan serves as a repository for other parallel technology, including data migration tools, an easy-to-use unstructured communication package, and a matrix-ordering interface.

The primary goal of our work was to incorporate Zoltan into Trilinos (through Petra) to allow load balancing to be done at the solver level. Given this capability, optimal decompositions for linear solvers and preconditioners could be pursued. Toward this goal, we developed classes and methods within Petra to support parallel data redistribution. We enhanced Zoltan's unstructured communication library and Trilinos' interface to it to allow variable-sized messages within Petra; this capability is necessary for efficient communication of matrix rows with unequal numbers of non-zeros. We also developed within Zoltan a distributed data directory to allow efficient location of needed off-processor data, and developed appropriate Petra interfaces to it.

In addition to developing infrastructure in Trilinos for parallel data redistribution, we developed new partitioning strategies that can be effective for solvers within applications. We invented a new multi-criteria geometric partitioning strategy that allows the creation of a single partition that is good with respect to a number of weights per data object. Examples scenarios where this capability is useful include balancing with respect to computation time and memory usage and balancing with respect to several phases of an application (e.g., preconditioning and linear solution). While the multi-criteria geometric partitioners were not always as effective as multi-criteria graph partitioners, the results are encouraging and important at Sandia, where many applications prefer to use geometric partitioners.

We also developed Sandia's first hypergraph partitioner. By using a more accurate model of communication, hypergraph models allow more effective partitioning than standard graph models for highly connected and semi-dense systems. Hypergraph models also have greater applicability than graph models, allowing effective partitioning of both non-symmetric and rectangular systems. Our experiments comparing hypergraph partitioning to graph partitioning on matrices from Sandia application Tramonto [21] produced up to 56% reductions in communication volume for matrix-vector multiplication, a key kernel of linear solvers. Design-of-experiment research with our hypergraph partitioners provided valuable insight into commonly used heuristics and proved no statistically significant benefit to a number of published strategies. These efforts laid the foundation for continuing research into parallel hypergraph partitioning.

In this report, we document the research, development and experiments performed under the LDRD. In section 2, we describe the object-oriented interfaces for parallel data redistribution implemented in Petra. Sections 3 and 4 describe new capabilities developed in Zoltan and Petra for distributed data directory services and unstructured communication. In section 5, we present a new multi-criteria geometric partitioning algorithm and results comparing its performance to multi-criteria graph partitioners. Section 6 documents our development of hypergraph partitioning capability in Zoltan and presents results from design-of-experiment analysis and partitioning of unstructured matrices from Tramonto. When appropriate, individual sections include directions for further work.

2 Parallel Data Redistribution in The Petra Object Model

Parallel distributed memory matrix and vector computations are critical in many engineering and scientific applications. Some of the most important capabilities are

1. Construction of matrices, graphs and vectors on a distributed memory machine in a scalable way using a single global address space;
2. Support of basic computations such as vector updates, dot products, norms and matrix multiplication; and
3. Redistribution of already-distributed objects in a scalable, flexible way.

A number of linear algebra libraries [2, 3, 53,8] exist to address capabilities 1 and 2. However, capability 3 is not as commonly available. The Petra Object Model (POM) is an object-oriented design that supports all three capabilities. Presently there are three different implementations of POM:

- Epetra: The current production implementation of POM. Epetra is restricted to real-valued, double-precision data, using a stable core of the C++ language standard.
- Tpetra: The next generation C++ version. Templates are used for the scalar and ordinal fields, allowing any floating-point data type and any sized integer. Tpetra uses more advanced features of C++.
- Jpetra: A pure Java implementation that is byte-code portable across all modern Java Virtual Machines.

These three implementations provide the foundation for solver development in the Trilinos Project [24, 26]. In this work, we focus on parallel data redistribution, the third capability listed above.

2.1 Parallel Data Redistribution

Effective use of distributed memory parallel computers for single program, multiple data (SPMD) programming typically requires distribution of data that can be thought of as a single object. For example, in large-scale engineering applications, sparse matrices are often distributed such that no single processor can directly address all matrix entries. There is a large degree of flexibility in this data distribution. For the purposes of good performance, we want to distribute data to balance workloads and minimize communication costs. For unstructured problems such as finite element methods on irregular grids and gridless problems such as circuit modeling, finding the optimal distribution is a nontrivial problem; what may be optimal for one phase of the application may be suboptimal for another phase. This type of setting leads to the need for *parallel data redistribution (PDR)*, which we define as redistributing already-distributed data in a parallel, scalable way.

2.1.1 PDR and Sparse Matrix Calculations

The need for PDR shows up in many practical situations when working with sparse matrices. Probably the most frequent situation is when performing sparse matrix-vector multiplication. Figure 1 shows a simple example on two processing elements (PEs), computing $w = Ax$ where the first (last) two rows of A and the first (last) two entries of x and w are stored on PE 0 (PE 1). One approach to computing w is to have the PE that owns row i of A compute the i^{th} element of w . Using this approach, PE 0 needs x_4 from PE 1, and PE 1 needs x_1 and x_2 from PE 0. Thus, each time a matrix-vector multiplication is performed, a PDR operation is also needed.

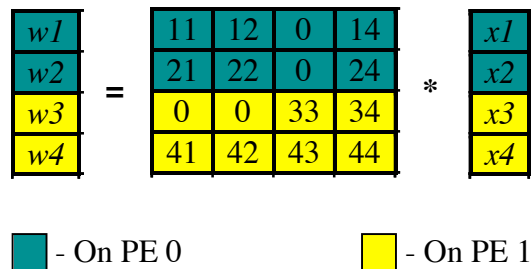


Figure 1 Sparse matrix multiplication on two PEs.

2.2 An Object-oriented approach to PDR

The Zoltan library provides an excellent, easy-to-integrate model for implementing PDR in an application. In this discussion, we present how that model is extended to an object-oriented approach. In order to do this, we first introduce the basic classes that we use to define distributions and create distributed objects.

2.2.1 Communication Classes

The POM uses a collection of three abstract interfaces to support parallel functionality. By using abstract interfaces, we do not create an explicit dependence on any particular machine or communication library. The three interfaces are listed below; the UML diagram in Figure 2 describes their interrelationships.

1. *Comm*: Supports basic communication operations such as barriers, broadcasts, and collective operations. Any implementation of this interface is also responsible for creating compatible instances of the *Distributor* and *Directory* interfaces listed next.
2. *Distributor*: Supports unstructured all-to-all communication that is commonly present in calculations where connectivity is described by adjacency graphs. The distributor has a setup phase that allows it to construct a “plan” for unstructured communication given a set of processors that each calling processor wants to communication with. Concrete *Distributor* instances are created by concrete *Comm* instances.

3. *Directory*: Supports the process of locating the owning processor of one or more global identifiers (GIDs) in an *ElementSpace* object (see below). This class facilitates the use of arbitrary indexing on our parallel machine. Concrete *Directory* instances are created by concrete *Comm* instances.

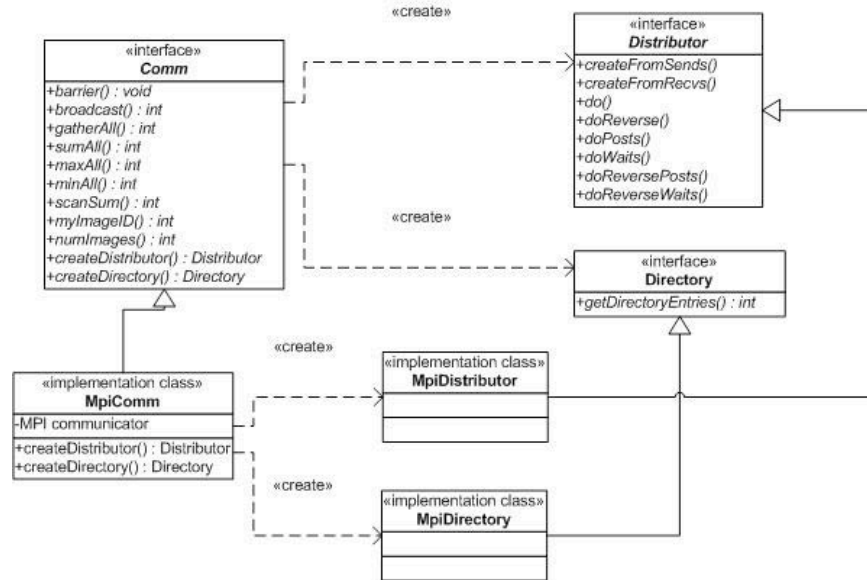


Figure 2: UML Diagram of POM Classes.

2.2.2 Element Spaces

Data redistribution requires the identification of data packets that should be moved as part of the redistribution. For a simple vector that is distributed across a parallel machine, the natural packet is a single vector value. For a collection of vectors with the same distribution, what we define to be a multivector, the natural packet is all values across a row of column vectors. For matrices, if we store nonzero entries row-by-row, then the index and nonzero values data are used to define a packet. We can similarly define a column-oriented matrix, or more generally a compressed index matrix, where the row or column orientation is part of the definition of the class attributes. A compressed index graph is similar to a matrix, except that it involves only pattern information. Table 1 lists the common linear algebra objects we use, and describes the packet definition for each object.

Object	Packet Definition
Vector	Single vector value
Multivector	Row of vector values
Compressed Index Storage Graph (CISGraph)	List of column/row indices for one graph row/column
Compressed Index Storage Matrix (CISMatrix)	List of values and column/row indices for one matrix row/column

Table 1: Packet Definition for Various Object Types

To facilitate redistribution and provide a generic analysis capability, we use elements as a representation of packets. Specifically, regardless of packet definition, we associate an element GID with each packet of a distributed object. We do this by defining an *ElementSpace* object (called a *Map* object in Epetra). *ElementSpace* objects are used to

1. Define the layout of distributed object across a parallel machine, and
2. Compute a plan to redistribute an object distributed via one *ElementSpace* to another *ElementSpace* distribution.

Example: Suppose we want to construct a vector with 99 entries so that it is approximately evenly distributed across four processors, and vector entries are stored in increasing order. Table 2 lists a natural distribution of element GIDs that would describe the layout of this type of vector.

Processor ID	Element GIDs
0	{0, 1, ..., 24}
1	{25, 26, ..., 49}
2	{50, 51, ... 74}
3	{75, 76, ..., 98}

Table 2: Standard Distribution of 99 elements

In our object-oriented model for PDR, we construct a distributed object by first defining an *ElementSpace* object. Given an *ElementSpace* object, we can define any number of linear algebra objects with a compatible layout. For example, using an *ElementSpace* object with the element ID distribution in Table 2, we can define any number of vectors having the first 25 vector entries on PE 0, the next 25 on PE 1, etc. We can also define a row matrix having rows 0 through 24 on PE 0, rows 25 through 49 on PE 1, etc.

To summarize, *ElementSpace* objects allow us to specify the desired distribution of any object for which we can define a packet. Given an *ElementSpace* object, we can then construct any number of distributed objects with the prescribe layout. As we will see below, *ElementSpace* objects also allow us to support redistribution of existing objects.

Repeated GIDs in ElementSpace Objects

An important property of an *ElementSpace* object is whether or not the GIDs are listed more than once. Certain kinds of operations make sense only if each GID is listed once in the *ElementSpace* object. In particular, as we discuss below, import and export operations need one of the two *ElementSpace* objects to have non-repeated GIDs.

2.2.3 Distributed Objects

A distributed object (*DistObject*) is any object constructed using an *ElementSpace* object. Although data for a given *DistObject* is kept locally on each processor, a *DistObject* is logically a single object and retains extra information, usually in the *ElementSpace* object itself, about the state of the object across all processors. In the Petra Object Model, there are a number of classes that implement the *DistObject* interface. Doing so requires the implementation of four virtual methods in the *DistObject* base class. These methods are discussed below.

Figure 3 shows the relationship between the major classes in the POM. In particular, it shows that the *MultiVector*, *Vector* (because *Vector* is a subclass of *MultiVector*), *CISGraph* and *CISMatrix* are all *DistObject* classes. In the case of the *Vector* class, the implementation of *DistObject* associates a single vector value with each GID of the *ElementSpace* object that describes the vector layout across the parallel machine. Thus, by implementing the *DistObject*, each of the concrete *DistObject* implementations builds an association between the packet definition (as listed in Table 1) and the GIDs in the *ElementSpace* object.

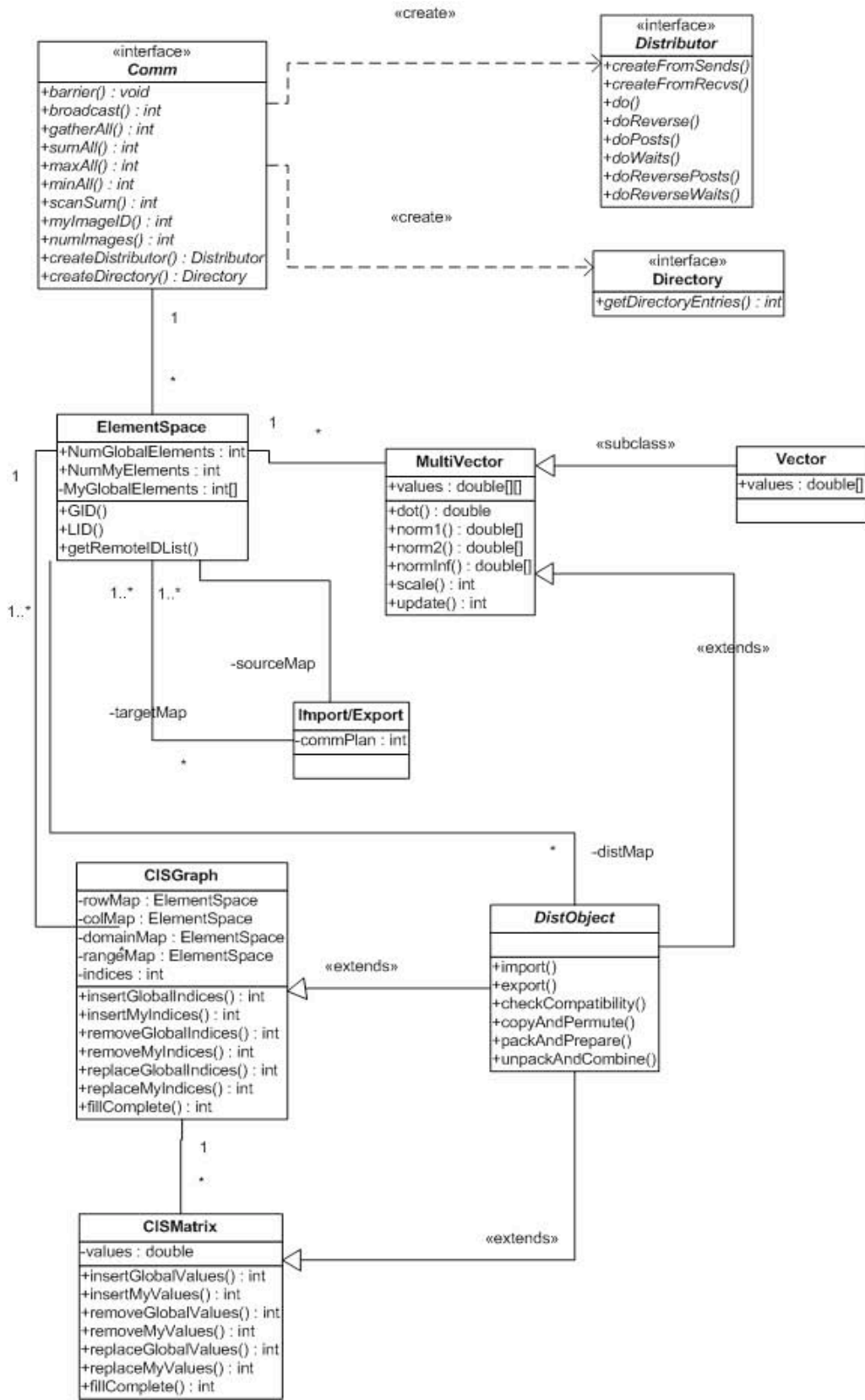


Figure 3 Basic Petra class diagram.

2.2.4 Import and Export Operations

The primary purpose of the *DistObject* base class is to facilitate PDR operations in the POM. We define PDR operations as one of two types:

1. **Import:** In this case, the calling processor knows which elements of the source *DistObject* are needed. For example, when performing row-oriented matrix vector multiplication, as illustrated in Figure 1, each processor scans the column entries of the matrix rows that are locally owned. In Figure 1, PE 0 has entries in the first, second and fourth column. Therefore, PE 0 knows that it needs the corresponding entries from the x vector. Since each PE knows which elements of x are needed, an *Import* object is constructed; each time matrix vector multiplication is performed, the *Import* object is invoked to get the elements of x needed by each processor. Note that in this case, the source x vector must have non-repeated GIDs in its *ElementSpace* in order for the *Import* object to uniquely identify the needed x values.
2. **Export:** In this case, the calling processor has contributions to elements that it wants to send to another processor. For example, when performing matrix transpose vector multiplication with a row-oriented matrix like the one in Figure 1, we interpret the rows of the matrix as columns of A^T . Thus, no communication is required to get elements of the x vector. However, when the local computation is completed, each processor may have contributions to the w vector that must be sent to the appropriate processors. For the matrix in Figure 1, PE 0 has a contribution to the fourth entry of w , which must be sent to PE 1. PE 1 has contributions to both the first and second entries of w , which must be sent to PE 0.

2.2.5 Import/Export Uses

Figure 4 shows an Epetra code fragment (part of a working example) that reads in a matrix (*readA*), an initial guess (*readx*), a right-hand-side (*readb*) and an exact solution (*readxexact*) from a data file. The function *Trilinos_Util_ReadHb2Epetra* creates these objects and also constructs a map (the Epetra equivalent to an *ElementSpace*) such that PE 0 owns all data. Once the data is available on PE 0 (read from file), we create a new map that has a uniform distribution of elements across all processors. We then create an *Export* object that uniformly redistributes any object based on the *readMap* to all PEs. Finally we create A , x , b , and $xexact$, and export data across all PEs.

```

Epetra_Map * readMap;
Epetra_CrsMatrix * readA;
Epetra_Vector * readx;
Epetra_Vector * readb;
Epetra_Vector * readxexact;

// Call routine to read in HB problem. All data on PE 0.
Trilinos_Util_ReadHb2Epetra(argv[1], Comm, readMap, readA, readx,
readb, readxexact);

// Create uniform distributed map
Epetra_Map map(readMap->NumGlobalElements(), 0, Comm);

// Create Exporter to distribute read-in matrix and vectors
Epetra_Export exporter(*readMap, map);
Epetra_CrsMatrix A(Copy, map, 0);
Epetra_Vector x(map);
Epetra_Vector b(map);
Epetra_Vector xexact(map);

// Distribute data from PE 0 to uniform layout across all PEs.
x.Export(*readx, exporter, Add);
b.Export(*readb, exporter, Add);
xexact.Export(*readxexact, exporter, Add);
A.Export(*readA, exporter, Add);
A.FillComplete();

```

Figure 4 Code fragment for reading file on PE0 and uniformly distributing data across all PEs.

Import and *Export* operations can be used with any *DistObject* (graphs, matrix, etc.). They can be used to provide elegant implementations of many basic algorithms:

- Parallel matrix and vector assembly for finite element and finite volume applications – Shared nodes receive contributions from multiple processors; the reverse operation replicates results back: When assembling a global stiffness matrix from local element stiffness matrices in parallel, a popular method for decomposing the domain is the creation of subdomains of elements. In this situation, multiple processors make contributions to nodes that are shared by elements on subdomain boundaries. Using export operations, it is possible to perform the global assembly process by first assembling locally and then arbitrating the ownership of shared node data via export of matrix rows. This process is used in the *Epetra_FECrsMatrix* and *Epetra_FEVbrMatrix* classes [25].
- Higher order interpolations – The needed values along subdomain boundaries can be imported: Higher order interpolations in a distributed memory environment can require information from nodes that are on the interior of neighboring subdomains. Import operations can be used to bring those data from neighboring processors, followed then by completely local calculation of the interpolant.

- Ghost node distributions: A more general framework for the above two examples is ghost node support. In this framework, multiple copies of nodes in a distributed mesh are maintained. Import and export operations provide an easy way to update and maintain the synchronization of ghost node data across the parallel machine.
- Rebalancing of changing work loads: As a simulation proceeds, or in different phases of a simulation, there may be benefit in redistributing already-distributed objects. Import and export operations make this an efficient and easily expressed operation.
- Explicit computation of the transpose of a sparse matrix: Use of an export operation can make forming a distributed transpose of an existing matrix a two-step process of forming the local transpose, followed by merging of row data across processors.
- Gradual transformation of a shared memory or serial application to distributed memory: Starting from a single processor application, export operations can be used to distribute data from a root processor in order to introduce distributed computing gradually. As one section of code is confirmed to run well in parallel, the export can be executed early in the code sequence. Also, this approach is effective for reading data on a single processor, avoiding parallel I/O, and then exporting the data for parallel execution.
- Rendezvous algorithms such as the *Directory* class in the POM: Rendezvous algorithms support unstructured communication algorithms by adding an indirect layer of registration and querying much like the alphabetizing of names in a phone book. Export and import operations facilitate the setup and use of a directory.

2.3 Implementation Issues for PDR

Following the overview of PDR in the previous section, we now proceed to discuss implementation issues. In particular, we discuss the internal structure of *Import* and *Export* objects and the details of the *DistObject* base class.

2.3.1 Computing Importers and Exporters

As mentioned in the previous section, *Import* and *Export* objects are constructed using two *ElementSpace* objects, which we call the *source* and *target*. The source *ElementSpace* object represents the layout of objects prior to the PDR operation. The target *ElementSpace* object represents the desired layout of objects after the PDR operation. Table 3 lists the restrictions on the source and target map properties for each operation.

Operation	<i>ElementSpace</i> that must have non-repeated GIDS	<i>ElementSpace</i> that has no GID restrictions
Import	Source	Target
Export	Target	Source

Table 3: Rules for Import and Export *ElementSpaces*

To construct an *Import* or *Export* object, the user passes in a source and a target *ElementSpace* object to the *Import* or *Export* constructor. The constructor then compares the GIDs of the source and the target. For the *Import* operation, the GIDs of the target are placed in one of three categories. For an *Export*, the GIDs of the source are categorized. In both cases the categories are

1. Identity mapping: We start comparing the GIDs of the source and target by first checking if the first and any immediately subsequent GIDs are identical in both the source and target. We then record how many we found. This test allows us to efficiently handle the important special case where most of the GIDs in the source and target are identical. Once we detect a difference in the list of GIDs, we stop this scan, even if other GIDs in the list may be identical matches.
2. Permuted mapping: After scanning for the initial list of identical GIDs, we then scan the remaining GIDs in each map to determine if there are GIDs that are in both the source and target that are local to the calling process. The PDR operation for these GIDs can be a local memory copy, requiring no communication.
3. Remote mapping: Any remaining GIDs, after the first two categories are determined, must necessarily be associated with another PE. Thus we form an instance of the *Distributor* class to handle the remote communication. This *Distributor* object will be invoked whenever an *Import* or *Export* operation is executed.

After an *Import* or *Export* object is constructed, it can be used as an input argument for any *Import* or *Export* methods implemented by the concrete *DistObject* classes such as *CISMatrix*, *CISGraph* or *MultiVector*.

2.3.2 Implementing the *DistObject* Base Class

The *DistObject* base class has four methods that must be implemented by any derived class. The methods provide the details of the packet definition for the derived class. Once these methods are defined the *DistObject* class can perform any necessary redistribution. The four methods are

1. **checkCompatibility**: This method allows the implementing class to check if a source object and a target object are compatible with each other and with the import or export object that is doing the PDR operation. A detail that we have left out until now is that, in addition to the *DistObject* class, the POM has a

source-distributed object (*SrcDistObject*) class that is a base class for *DistObject*. In practice, the requirements for an object to be a *SrcDistObject* are fewer than the requirements to be a target distributed object. For example, it is possible for an abstract *RowMatrix* class to be a source object for the construction of a *CISMatrix* that is a redistribution of the *RowMatrix* object. *RowMatrix* is an abstract base class that can be implemented in many concrete ways. It is conceptually a distributed object and can be queried for row matrix data packets just like a *CISMatrix* object. However, it does not make sense to have a *RowMatrix* object as a target, since we must construct something concrete for a target.

2. **copyAndPermute:** Prior to packing data that will be sent to other processors, we handle any packets that are copied locally. This step allows us to use export and import operations as efficient techniques to locally permute data. For best efficiency, this method must be implemented such that source-object packets are copied locally if they are to go in either the same location in the target object or are to be locally permuted.
3. **packAndPrepare:** This method packs source-object packets in preparation for the global communication step. Data is packed in a generic form that a general communication library can redistribute. The basic type is an eight-bit byte. For example, if the source object were a double precision vector, each double value would be recast as eight generic bytes.
4. **unpackAndCombine:** This method is called after data packets have been redistributed according to the “plan” that was encoded in the *Import* or *Export* object. At this point, the external data is in generic form and must be encoded to the form that is understood by the target object. Also, if multiple packets associated with the same GID are received, combining rules must be applied. The most commonly used combining rule is “Add,” where results are summed together, but “Max,” “Min,” “Ignore” and “Average” are also allowed.

2.4 Parallel Data Redistribution Results

Given the tools described above, we have made substantial progress on implementing PDR in applications. In particular, the circuit modeling code Xyce [27] has adopted the techniques we developed for redistributing data to improve the performance of preconditioned iterative solvers. We also note that some preliminary studies using the application Sundance [39] show the importance of PDR for highly convective flows. Continued work in this area is important.

2.4.1 Robust Calculations and Dynamic Load Balancing

One of the most important uses for PDR is the redistribution of data for robust preconditioning. Preconditioned iterative methods are an important component in many applications. Of all phases of computation in a parallel distributed-memory application, the preconditioner is the only phase whose robustness is impacted by how data is

distributed on the parallel machine. Other phases of computation may have imbalanced loads, but the numerical calculations are essentially unaffected by the distribution of data. In contrast, most robust parallel preconditioners rely on the locality of data in order to introduce parallelism into an otherwise sequential algorithm. The data distribution can make the difference between convergence and divergence.

Even when robustness is not an issue, the layout for optimal load balancing of the application is often starkly different from an optimal layout for the solver. In this case, it is also important to consider PDR as a means for improving the load balance for the solver.

2.4.2 Epetra-Zoltan Interface

In light of the above facts, one of the most important uses for PDR is for determining a distribution for the preconditioner or for the preconditioned solver, even if that distribution is different from the rest of the application. To facilitate this, we have developed an interface between Epetra and Zoltan. This interface provides connectivity information to Zoltan via an *Epetra_CrsGraph* object. Given this graph, Zoltan produces an approximately optimal redistribution; the interface returns this redistribution information in an *Epetra_Export* object. Given this *Export* object, Epetra objects can be redistributed to the Zoltan-produced distribution. This mechanism has been successfully used in the Xyce circuit simulation package.

2.4.3 Partitioning for Highly Convective Flows

We mention briefly some studies related to our work. In the thesis by Michael W. Boldt [6], we see the impact of partitioning on highly convective flows. In these example problems, developed within the Sundance application [39], we clearly see that partitioning the problem with respect to streamlines can have a large impact on the convergence of the linear solver. Determining the proper partitioning for the problems in this thesis was fairly straightforward because the grids were structured. In future work, we plan to incorporate the ability to automatically partition with respect to streamlines so that similar results can be obtained for unstructured meshes and for problems that are intrinsically discrete.

2.5 Conclusions

We have presented an object-oriented model for parallel data redistribution that is both flexible and powerful. Using the Petra Object Model, we can deliver qualitative improvement in robustness and provide a set of tools that make parallel distributed-memory implementations of many algorithms far more tractable. As part of this effort, we have developed and incorporated a variety of algorithms to support robust, scalable computations for unstructured problems on distributed-memory parallel machines. We have also delivered these capabilities to a broad set of users in the Epetra package (as part of the Trilinos Project) and in the Zoltan package.

3 Distributed Data Directory

Dynamic applications often need to locate off-processor information. For example, after repartitioning, a processor may need to rebuild ghost cells and lists of objects to be communicated; it may know which objects it needs, but may not know where they are located. Similar information is needed to build Import and/or Export maps (see Section 2.2.4). To help locate off-processor data, we developed in Zoltan a distributed data directory algorithm based on the rendezvous algorithm developed by Pinar and Hendrickson [45]. Applications can use the data directory to efficiently track processor ownership and general information about their computational objects. The distributed data directory avoids communication, storage, and data processing bottlenecks associated with storing a data directory on a single processor or duplicating a directory on each processor. Instead, all available processors are used to create one logical data directory, with all directory information about a specific computational object located on exactly one processor.

3.1 Distributed Data Directory Usage

General use of the directory is outlined in Figure 5. Processors register their owned objects' global identifiers (GIDs) along with their processor number in a directory (by calling *Zoltan_DD_Update*). This directory is distributed evenly across processors in a predictable fashion (through either a linear decomposition of the objects' GIDs or a hashing of GIDs to processors). Then, other processors can obtain the processor number of a given object by sending a request for the information to the processor holding the directory entry (by calling *Zoltan_DD_Find*).

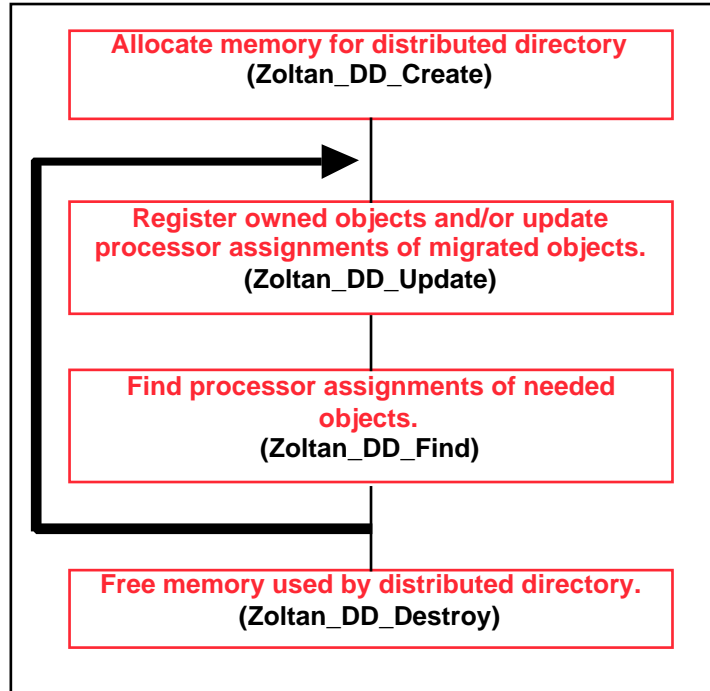


Figure 5 Outline of distributed data directory usage. Once created, directories can be updated and re-used throughout an application to reflect, say, changing processor assignments due to dynamic load balancing.

Because some of Sandia's parallel computers do not support threads, the distributed data directory does not use threads. Thus, applications using the distributed data directory must perform directory operations on all processors at the same point in the application program, rather than asynchronously. All processors create, update, search and destroy the data directory simultaneously. Processors having no current data requests must still participate in directory operations, calling the directory functions with empty GID lists (NULL pointers).

3.2 Distributed Data Directory Implementation

The distributed data directory is implemented as a Zoltan utility. It is included in both the Zoltan library *libzoltan.a* and in its own linkable library *libzoltan_dd.a*. It depends upon the Zoltan memory (*libzoltan_mem.a*) and communication (*libzoltan_comm.a*) utilities, but has no dependency on the Zoltan library itself. Thus, applications can use only the directory capabilities without linking with the entire Zoltan library.

The directory is distributed in roughly equal parts to all processors. A globally known function is used to map an object's GID to the specific processor maintaining the directory information for that object; this function can be a simple linear ordering of GIDs or a hash function. Messages to insert, update, find, or remove an object's directory data are sent to the processor storing the object's directory entry. Thus, the rendezvous algorithm requires only $O(n)$ total memory usage for n objects, and all look-up operations require only $O(1)$ communication.

The data directory on each processor is a fixed length table whose entries are pointers to the heads of singly linked lists. Each node in the linked list contains all of the directory information for one object: the GID, the current owner (processor number), the current partition, the local identification (LID), and a user-defined data field. A local hash function (which must be different from the global hash function) indexes into the table to find the head of the appropriate linked list. The linked list is traversed to find the object's data. If an object's data is not found, a node is automatically created at the end of the appropriate linked list for that object's data.

3.3 Distributed Data Directory Functions

Zoltan implementation

Zoltan_DD_Create initializes the data directory by allocating space for the fixed size hash table of linked list pointers. The linked lists are created and resized dynamically as needed.

Zoltan_DD_Destroy frees all memory associated with the distributed data directory.

Zoltan_DD_Update inserts new data into the directory and updates existing data. It can be used, for example, to update processor and partition assignments after load balancing, or add new data after adaptive mesh refinement.

Zoltan_DD_Remove is used to remove directory entries for the GIDs in its calling list. This function is useful, for example, after adaptive mesh coarsening.

Zoltan_DD_Find returns all of the information known about each GID in its calling list.

Zoltan_DD_Set_Hash_Fn allows users to register their own global hash function (to map objects to processors), if desired. A default hash function is used in the distributed directory, so use of *Zoltan_DD_Set_Hash_Fn* is optional. Several additional hash functions are provided with the distributed data directory; they can be used directly or serve as templates for user-provided hash functions that take advantage of the user's particular naming scheme.

Zoltan_DD_Print prints the entire contents of the directory.

Zoltan_DD_Stats prints a summary of directory information, including the hash table size, the number of linked lists, and the length of the longest linked list.

Epetra implementation

The Zoltan distributed data directory has been incorporated into Epetra as an implementation of the Epetra *Directory* class (see Section 2.2.1). The class *Epetra_ZoltanDirectory* has methods *Remove*, *Update*, *Find*, *Stats*, *Print*, and *SetHash*; these methods call the equivalent Zoltan routines directly. The class constructors and destructors contain calls to the Zoltan routines to *Create* and *Destroy*.

4 Unstructured Communication

Unlike static applications where communication patterns remain fixed throughout the computation, dynamic applications can have complicated, changing communication patterns. For example, after adaptive mesh refinement, new communication patterns must reflect the dependencies between newly created elements. Multi-physics simulations, such as crash simulations, may require complicated communication patterns to transfer data between decompositions for different simulation phases. Similarly, applications such as Xyce may use separate decompositions for matrix assembly and linear system solution, requiring transfer of data between the decompositions.

Zoltan provides an unstructured communication package to simplify communication. The package generates a communication “plan” based on the number of objects to be sent and their destination processors. This plan includes information about both the sends and receives for a given processor. The plan may be used and reused throughout the application, or it may be destroyed and rebuilt when communication patterns change. It may also be used in reverse to return data to requesting processors. The package includes simple communication primitives that insulate the user from details of sends and receives.

In Figure 6, we show how the communication package can be used to transfer data between two different meshes in a loosely coupled physics simulation. In this crash simulation, a static graph-based decomposition generated by Chaco [23] (left) is used for the finite element analysis; a dynamic Recursive Coordinate Bisection (RCB) decomposition (right) is used for contact detection. A communication plan is built to describe data movement between the two decompositions (through a call to *Zoltan_Comm_Create*). Using the plan, data is transferred between the graph-based and RCB decompositions through calls to *Zoltan_Comm_Do* and *Zoltan_Comm_Do_Reverse*.

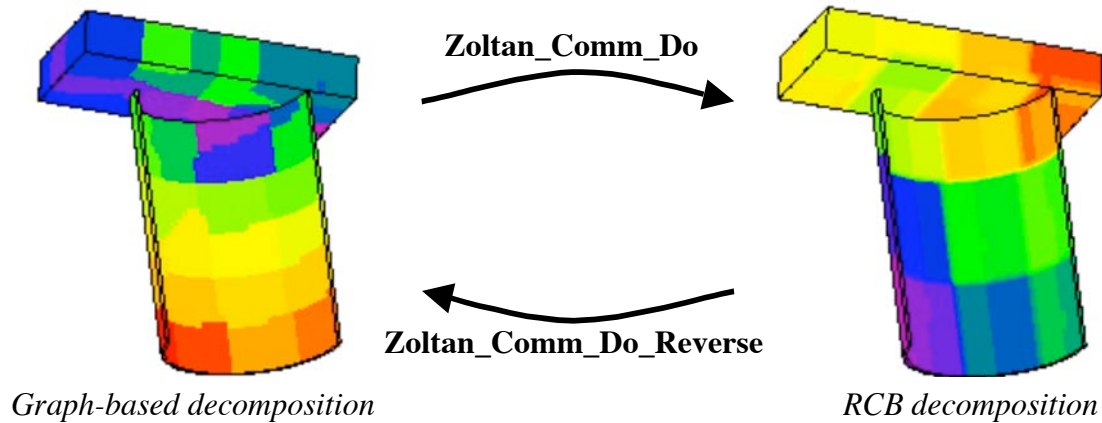


Figure 6 Demonstration of Zoltan’s unstructured communication package for loosely coupled physics. In this example, communication primitives simplify mapping of data between two different decompositions. (Images courtesy of Attaway, et al., Sandia National Laboratories.)

Zoltan development

The Epetra’s ability to send and receive variable-sized data packets is important for many new applications. While matrices arising from finite element simulations have roughly the same number of non-zeros per row (and, thus, roughly the same size packets for each object), applications like the Xyce parallel circuit simulator can have widely differing numbers of non-zeros per row. For example, a system bus in a circuit creates a dense row in the resulting matrix. Requiring all data packets to use the maximum packet size results in unacceptable additional memory and communication costs.

Because the Zoltan communication package supports variable-sized data packets, its inclusion in Epetra as an implementation of a *Distributor* class (see Section 2.2.1) was critical to Epetra’s performance. Several enhancements were made in the Zoltan communication package to support all methods of the Epetra *Distributor* class. Most significantly, “post” and “wait” versions of *Zoltan_Comm_Do* and *Zoltan_Comm_Do_Reverse* were added to allow applications to overlap computation and communication. Users can replace *Zoltan_Comm_Do* with the new function *Zoltan_Comm_Do_Post* (with the same calling arguments as *Zoltan_Comm_Do*) to initiate communication, perform computations, and later complete the communication by calling *Zoltan_Comm_Do_Wait* (with the same arguments as *Zoltan_Comm_Do*). Analogously, the new functions *Zoltan_Comm_Do_Reverse_Post* and *Zoltan_Comm_Do_Reverse_Wait* allow computation to be overlapped with the communication previously done by *Zoltan_Comm_Do_Reverse*.

The internal utility function *Zoltan_Comm_Sort_Ints* sorts messages by processor number to make the unstructured communication repeatable (deterministic) and sequence the order of sends and receives to improve throughput. To improve performance, we

replaced the quick sort algorithm in this function with the faster distribution count sort [37].

Epetra Implementation

The Zoltan unstructured communications library was incorporated into Epetra as an implementation of the *Epetra_MpiDistributor* class. The class constructor creates a default class object with an instantiation counter to prevent freeing the Zoltan communication plan *ZOLTAN_COMM_OBJ* until the last occurrence is destroyed. The class destructor uses the instantiation counter to determine when *Zoltan_Comm_Destroy* should be called to free the *ZOLTAN_COMM_OBJ*. The Epetra *Distributor* class method *CreateFromSends* is a shallow wrapper calling *Zoltan_Comm_Create* to make a communication plan based on data the processor will export. The class method *CreateFromRecvs* creates a communication plan using data the processor will import; it calls *Zoltan_Comm_Create* to create a plan and *Zoltan_Comm_Do* to communicate the buffer sizes needed to create the inverse plan. The *Distributor* class methods *Do*, *DoPost*, *DoWaits*, *DoReverse*, *DoReversePost*, and *DoReverseWaits* call the analogous Zoltan functions described above. The overloaded methods for *Do* and *DoReverse* that accept variable size messages also invoke *Zoltan_Comm_Resize* to update packet sizes in the communication plan. The class method *Print* calls *Zoltan_Comm_Info* to print detailed plan information.

5 Multicriteria partitioning and load balancing

5.1 Introduction

Load balancing is important to get good performance in parallel computing. Most work has focused on the case where there is a single type of load to be balanced. For example, one may wish to distribute data such that the computational work for each processor is about the same for all processors. Recently, there has been interest in *multicriteria* load balancing, where there are several loads that need to be balanced. The challenge is to compute a single partitioning (balance) that is fairly balanced with respect to all the different loads. One practical example is to balance data with respect to both computation and memory usage. Another example is to partition a sparse matrix both for sparse matrix-vector multiplication and preconditioning. Yet another application is in multi-physics simulation, where each phase is different but some data are shared among phases.

Karypis et al. [34, 49] have considered the problem of multiconstraint graph partitioning. Their algorithm is implemented in version 3 of the ParMETIS library, which has been integrated into the Zoltan load-balancing toolkit. Many Sandia applications prefer geometric partitioning methods, like recursive coordinate bisection (RCB) or space-filling curve partitioning. Our goal in this part of the project is to generalize these algorithms to multiple criteria. We are not aware of any previous attempts to do this. One reason little work has been done in this area is that for general problems, there may not exist any good solutions to the multicriteria partitioning problem for geometric methods. Thus, the best we can hope for are heuristics that work well on many problems.

5.2 Linear partitioning and bisection

The generic partitioning problem is: Given a set of n data objects (each with a scalar weight) and a positive integer k , partition the objects into k partitions (subsets) such that the sum of the weights in each partition is approximately the same. In the geometric version of this problem, each object also has a set of coordinates in some geometric vector space, usually R^3 .

Most geometric partitioning methods reduce the partitioning problem to a linear (or one-dimensional) problem. For example, a space-filling curve partitioner defines a linear ordering of the objects and cuts the ordering into k equally sized pieces. Similarly, the RCB algorithm bisects the geometry perpendicular to only one coordinate axis at a time; the corresponding coordinate of the objects defines a linear order. Thus, even if the original partitioning problem has data objects with coordinates in a multidimensional space (typically R^3), we restrict our attention to the one-dimensional partitioning problem. This problem is also known as chains-on-chains, and has been well studied [28, 40, 43]. Again, we are unaware of any results for the multiple load case.

Although the partitioning problem allows for k partitions, we will focus on the bisection problem, i.e., $k=2$. The solution for general k can be obtained by recursively bisecting the resulting partitions (see Figure 7).

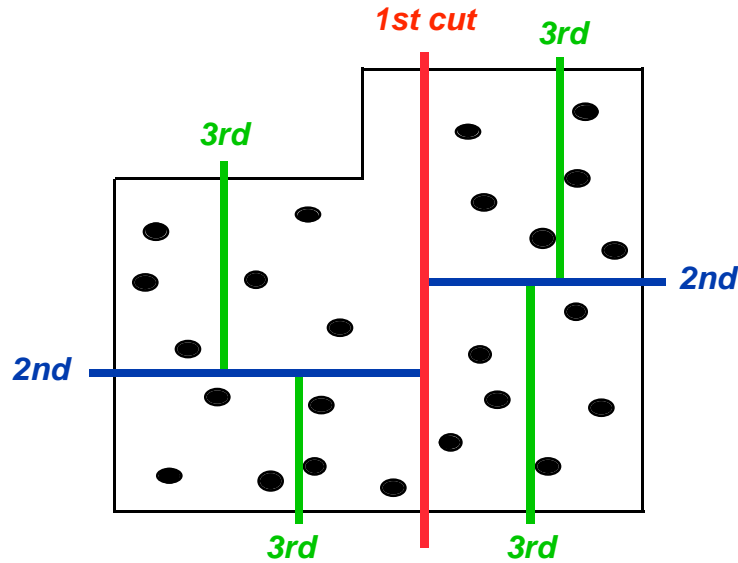


Figure 7 Diagram of cuts made during Recursive Coordinate Bisection partitioning. Cuts orthogonal to a coordinate axis divide a domain’s work into evenly sized sub-domains. The algorithm is applied recursively to resulting sub-domains.

5.3 Multiconstraint or multiobjective?

Traditional optimization problems are written in the standard form: minimize $f(x)$ subject to some constraints. Hence, handling multiple constraints is easy, but handling multiple objectives is much harder (i.e., it does not fit into this form). The multicriteria load-balancing (partitioning) problem can be formulated either as a multiconstraint or a multiobjective optimization problem. Often, the balance of each load is considered a constraint, and has to be within a certain tolerance. Such a formulation fits the standard optimization model, where, in this case, there is no objective, only constraints.

Unfortunately, there is no guarantee that a solution exists to this problem. In practice, we would like a “best possible” partitioning even if the desired balance criteria cannot be satisfied. Thus, an alternative is to make the constraints objectives; that is, we want to achieve as good balance as possible with respect to all the different loads. Multiobjective optimization is a very hard problem, because, in general, the objectives conflict and there is no unique “optimal solution.”

There is a vast collection of literature in this area, and we discuss some of the most popular methods in the next section.

5.4 Multiobjective methods

Consider the multiobjective optimization problem

$$\text{Minimize } f(x) \text{ s.t. } c(x) \geq 0,$$

where both c and f are vector-valued functions. We briefly describe three different approaches.

a) Weighted sum. This approach is perhaps the simplest. The idea is to form a linear combination of all the objectives and sum them up into a single objective. More precisely, if there are d objectives, choose positive weights w_1, \dots, w_d and define $F(x) = w_1 f_1(x) + \dots + w_d f_d(x)$. $F(x)$ can then be minimized using standard methods. The difficulty with this method is choosing appropriate weights.

b) Pareto optimal set. A set of points is said to be Pareto optimal if, in moving from one point to another point in the set, any improvement in one of the objective functions from its current value would cause at least one of the other objective functions to deteriorate from its current value. The Pareto optimal set yields an infinite set of solutions. One difficulty with this method is how to choose a solution from the Pareto optimal set. Also, computing the Pareto optimal set can be expensive. If the problem is convex, one can use the weighted sum method to compute points in the Pareto set.

c) Global criterion methods. These methods require an estimate of an ideal solution f^* . One then tries to minimize the distance from the ideal solution in some metric. The most common variation is

$$\min g(x) = \sum_i ((f_i(x) - f_i^*) / f_i^*)^p, \text{ where typically, } p=1 \text{ or } p=2.$$

5.5 Multiobjective bisection

In dynamic load balancing, speed is often more important than quality of the solution. We therefore focus on fast algorithms. The unicriterion (standard) RCB algorithm is fast because each bisecting cut can be computed very quickly. Computing the cuts is fast because it requires solving only a unimodal optimization problem. We want the same speed to apply in the multicriteria case. Thus, we can remove many methods from consideration because we can't afford to solve a global optimization problem, not even in one dimension.

We consider mathematical models of the multicriteria bisection problem. Suppose we have a set of n data points. Let a_1, a_2, \dots, a_n be the corresponding loads (weights), where each a_i is a vector. Informally, our objective is to find an index s , $1 \leq s \leq n$, such that

$$\sum_{i \leq s} a_i \leq \sum_{i > s} a_i.$$

When each a_i is scalar, this problem is easy to solve. One can simply minimize the larger sum, that is,

$$\min_s \max\left(\sum_{i \leq s} a_i, \sum_{i > s} a_i\right)$$

However, in the multicriteria case, each a_i is a vector and the problem is not well-defined. In general, there is no index s that achieves approximate equality in every dimension.

Applying the weighted sum method to the formula above yields

$$\min_s w^T \max\left(\sum_{i \leq s} a_i, \sum_{i > s} a_i\right),$$

where the maximum of two vectors is defined element-wise and w is some cost vector, possibly all ones. (The vector w here plays the same role as in the weighted sum method.) This problem has a reasonable interpretation in load balancing if the j^{th} component of a_i represents the work associated with the i^{th} object and the j^{th} phase. We want to minimize the total time over all phases, assuming that one phase cannot start before the previous one has finished. Unfortunately, this problem is hard to solve because the function is non-convex, so global optimization is required.

Instead, we propose the following heuristic:

$$\min_s \max\left(g\left(\sum_{i \leq s} a_i\right), g\left(\sum_{i > s} a_i\right)\right)$$

where g is a monotonically increasing (or non-decreasing) function in each component of the input vector. Motivated by the global criterion method, we suggest using either $g(x) = \sum_j x_j^p$ with $p=1$ or $p=2$, or $g(x) = \|x\|$ for some norm. This formulation has

one crucial computational advantage: The objective function is unimodal with respect to s . In other words, starting with $s=1$ and increasing s , the objective decreases, until at some point the objective starts increasing. That defines the optimal bisection value s . Note that the objective may be locally flat (constant), so there is not always a unique minimizer.

Example: Suppose we are given a sequence of $n=6$ items, each with two weights:

$$a^{(1)} = (2,2,0,1,2,2) \quad (\text{first weight for the six items})$$

$$a^{(2)} = (2,1,1,1,1,0) \quad (\text{second weight for the six items})$$

There are five ways to partition this sequence with no empty partition, and we evaluate the cuts below:

<i>s</i>	<i>left</i>	<i>right</i>	<i>1-norm</i>	<i>2-norm</i>	<i>max-norm</i>	<i>time</i>
1	(2,2)	(7,4)	11	65	7	11
2	(4,3)	(5,3)	8	34	5	8
3	(4,4)	(5,2)	8	32	5	9
4	(5,5)	(4,1)	10	50	5	9
5	(7,6)	(2,0)	13	85	7	13

We computed the weight sum vector in the left and right halves, respectively, and the norms of the larger half. As predicted, the norm sequences are all unimodal. The different norms have different minimizers. In the 1-norm, there is a tie between $s=2$ and $s=3$, while the 2-norm has a unique minimizer at $s=3$. In the max-norm, there is a three-way tie. We also computed a fourth metric called “time” which corresponds to the time a parallel application would take if the weights were the times for two different phases.

This example shows that the choice of norm may affect the output of the bisection algorithm. In general, one cannot say one norm is superior. In the example above, all three norms yield reasonable results.

5.6 Implementation in Zoltan

Zoltan Recursive Coordinate Bisection (RCB) code

We have implemented a version of the algorithm proposed in the section above. Zoltan contains a fully parallel implementation of RCB. No assumption is made about how the data is distributed. A core routine in the RCB module is the routine *find_median*. Given a distributed set of real numbers, it computes the median value. This median is where RCB places the cut in the single-criteria case. We have replaced *find_median* with *find_bisector*, which computes a cut value (which we hope is good) for the multicriteria problem.

```

Serial algorithm:
function bisect(coord, wgt)
lo = min(coord)
hi = max(coord)
while ({lo<coord<hi} not empty)
  cut = (lo+hi)/2
  if (norm(sum(wgt(coord<cut))) < norm(sum(wgt(coord>cut))))
    lo = cut
  else
    hi = cut
end

```

Figure 8 Serial algorithm for *find_bisector*.

The actual implementation is more complicated. A dot (data item) is called active if it has not yet been assigned to a partition, and inactive if it has been. The loop runs until there are no more active dots.

At present, no attempt is made to find the best cut-directions. The code simply bisects in the largest dimension of the current geometry. Finding better choices is future work.

Zoltan Recursive Inertial Bisection (RIB) code

The Zoltan RIB routine has been modified in the same way as the RCB routine. An unresolved question is how to compute the axis of inertia when there are multiple weights (loads). Currently we only use the first weight for each object to compute the axis of inertia, but more clever algorithms should be explored in the future.

Scaling issues

While we did not explicitly scale the multidimensional weights (loads) in our algorithm, scaling is clearly important since the algorithm implicitly compares numbers corresponding to the different weight dimensions (types of load). We have chosen to make two types of scaling available: No scaling, or imbalance-tolerance scaling. No scaling is useful if the magnitude of the different types of weight reflects the importance of that load type. But in general, little is known about the multi-weights and the natural scaling is to make all weight dimensions (load types) equally important by scaling the sum of each weight dimension to one. Since the algorithm above does not take into account the desired imbalance tolerances, we make a slight modification. We scale the weights (loads) such that the load types with the largest imbalance tolerance have the smallest sum, and vice versa. This scaling is the default behavior. Currently the scaling is performed in *find_bisector*, but an alternative is to scale only once, not in every level of recursion in the recursive bisection algorithm.

5.7 Empirical results

We present results from two test examples, both finite element meshes. All computer simulations were run on a 32-processor Compaq/DEC Alpha machine at Sandia (stratus).

The first example, *ti_4k*, is a 4000-element mesh of a chemical reactor from MPSalsa [50, 51] with two weights per element ($d=2$). The first weight is one for each element; the second weight corresponds to how many sides (surfaces) of an element have no neighbors (i.e., are on the external surface). Such a weighting scheme is realistic for contact problems. (We did not have access to real data for contact problems.)

We partitioned this mesh into $k=9$ parts using our multicriteria RCB code and compared against ParMETIS. Results are shown in Table 4. $BALANCE[i]$ is computed as the maximum processor load for weight i divided by the average processor load for weight i , $i=0, \dots, d-1$. We observe that there is little difference between the multicriteria RCB algorithms with different norms. The balances are not quite as good as ParMETIS, which was expected since the RCB cuts are restricted to orthogonal planes. Still, the multicriteria RCB algorithm produces reasonable load balance for this problem, and in less time than ParMETIS. For comparison, we include the results for $d=1$; i.e., only the first set of weights are used. The edge cuts are the number of edges that are cut between partitions in the graph model, which approximately corresponds to the communication in a parallel code. The RCB algorithm does not use any graph information, while ParMETIS does.

$k=9$	RCB ($d=1$)	RCB ($d=2, \text{norm}=1$)	RCB ($d=2, \text{norm}=2$)	RCB ($d=2, \text{norm}=\text{max}$)	ParMETIS ($d=2$)
$BALANCE[0]$	1.00	1.06	1.06	1.06	1.01
$BALANCE[1]$		1.15	1.08	1.08	1.01
Edge Cuts	1576	1474	1488	1488	1462
Time	0.09	0.10	0.15	0.11	0.23

Table 4 Results comparing balance, edge cuts, and computation time using multicriteria RCB and ParMETIS for the *ti_4k* finite element mesh.

The second test problem is *brack2_3*. *Brack2* is a 3D mesh for a brackish water area with 62,631 nodes. The version we used has 3 weights per node ($d=3$). These weights were artificially generated by Schloegel and Karypis while testing the multiconstraint feature in ParMETIS [49]. Results for $k=4, 8, \text{ and } 16$ partitions are shown in Table 5. From these results, we see that the load balance for RCB deteriorates rapidly with increasing number of partitions k . Again, ParMETIS does better. We also note that there is little difference between the RCB variations with different norms; in fact some produce exactly the same results.

More experiments are needed to draw firm conclusions, but it looks as if the multicriteria RCB algorithm is useful only for small numbers of partitions k ; for larger k , the imbalance grows too large. The RCB method is very competitive in terms of cut quality and execution time.

$k=4$	RCB ($d=1$)	RCB ($d=3, \text{norm}=1$)	RCB ($d=3, \text{norm}=2$)	RCB ($d=3, \text{norm}=\text{max}$)	ParMETIS ($d=3$)
<i>BALANCE[0]</i>	1.00	1.21	1.21	1.21	1.1
<i>BALANCE[1]</i>		1.21	1.21	1.21	1.1
<i>BALANCE[2]</i>		1.14	1.14	1.14	1.04
<i>EDGE CUTS</i>	16540	18548	18548	18548	12198
<i>TIME</i>	0.23	0.26	0.25	0.26	1.1
$k=8$	RCB ($d=1$)	RCB ($d=3, \text{norm}=1$)	RCB ($d=3, \text{norm}=2$)	RCB ($d=3, \text{norm}=\text{max}$)	ParMETIS ($d=3$)
<i>BALANCE[0]</i>	1	1.64	1.64	1.53	1.08
<i>BALANCE[1]</i>		1.29	1.29	1.34	1.09
<i>BALANCE[2]</i>		1.43	1.43	1.34	1.04
<i>EDGE CUTS</i>	22650	26882	26882	23710	22796
<i>TIME</i>	0.31	0.31	0.25	0.24	0.91
$k=16$	RCB ($d=1$)	RCB ($d=3, \text{norm}=1$)	RCB ($d=3, \text{norm}=2$)	RCB ($d=3, \text{norm}=\text{max}$)	ParMETIS ($d=3$)
<i>BALANCE[0]</i>	1	1.78	1.78	1.43	1.08
<i>BALANCE[1]</i>		1.52	1.52	1.74	1.08
<i>BALANCE[2]</i>		1.5	1.5	1.53	1.09
<i>EDGE CUTS</i>	33574	36414	36414	32756	37910
<i>TIME</i>	0.22	0.32	0.25	0.25	1.04

Table 5 Results comparing balance, edge cuts, and computation time using multicriteria RCB and ParMETIS for the brack2_3 graph.

5.8 Future work

Our bisection algorithm has so far been implemented only in RCB and RIB, but it applies equally well to space-filling curve partitioning. However, the code would need to be substantially modified because the data structures and the parallel distribution are different.

At present, a crude rule is used to decide the cut directions in RCB: the cut direction is selected to be orthogonal to the longest direction in the geometry. Cut directions are much more important for multicriteria partitioning than standard partitioning. A simple improvement is to try all three dimensions and pick the one giving the best results. We plan to implement this improvement in the next fiscal year.

RCB is restricted to cutting along the coordinate axes, while RIB is not. A natural question is how much better one can do if one is allowed to choose an arbitrary cutting

plane at every step. There is an interesting theoretical result, known as the **Ham Sandwich Theorem** [52]:

Given n solid bodies in R^n , there exists a $(n-1)$ -dimensional hyperplane that simultaneously bisects (exactly) all n bodies.

A popular interpretation for $n=3$ follows: If you take a sandwich with ham and cheese on bread, it is possible to slice it such that each half contains exactly the same amount of bread, ham, and cheese.

The Ham Sandwich Theorem implies that a set of points in R^n , each with a n -dimensional binary weight vector, can be cut by a $(n-1)$ -dimensional hyperplane such that the vector sum in the two half-spaces differs by at most one in each vector component. A linear time algorithm exists for $n=2$, and some efficient algorithms exist for other low dimensions [38].

6 Hypergraph Partitioning

Graph partitioning is generally accepted as one of the most effective partitioning strategies for mesh-based PDE simulations. In graph partitioning, vertices represent the data to be partitioned (e.g., finite element nodes, matrix rows). Edges represent relationships between vertices (e.g., shared element faces, off-diagonal matrix entries). Thus, the number of edges that are “cut” by partition boundaries approximates the volume of communication needed during computation (e.g., flux calculations, matrix-vector multiplication). Both vertices and edges can be weighted to reflect associated computation and communication costs, respectively. The goal of graph partitioning, then, is to assign equal total vertex weight to processors while minimizing the weight of cut edges.

It is important to note that the edge-cut metric is only an approximation of an application’s communication volume. For example, in Figure 9 (left), a grid is divided into two partitions (separated by the red line). In the graph model, grid point *A* has four edges associated with it; each edge (shown in blue) connects *A* with a neighboring grid point. Two of the edges are cut by the partition boundary; however, the actual communication volume associated with sending *A* to the neighboring processor is only one grid point.

Nonetheless, there are countless examples of successful uses of graph partitioning in mesh-based PDE applications like finite element methods and the sparse iterative solvers. These successes have led to the development of many high quality serial and parallel graph partitioning tools (e.g., Chaco [23], METIS [35], Jostle [54], Party [47], Scotch [42], ParMETIS [32]). The Zoltan library includes interfaces to ParMETIS and Jostle to provide parallel, dynamic graph partitioning.

While graph partitioners have served well in mesh-based PDE simulations, many new simulation areas such as electrical systems, computational biology, linear programming and nanotechnology show the limitations of these algorithms. Critical differences between these areas and mesh-based PDE simulations include high connectivity, heterogeneity in topology and matrices that are non-symmetric or rectangular. The examples in Table 8 and Table 9 include the non-zero structure of matrices representative of these new applications; it is easy to see the vastly different structure of these matrices compared to a traditional finite element matrix (Table 7). Current technology based on graph partitioning does not address these problems well. Hypergraph partitioning models [14], on the other hand, show great promise in helping these applications achieve high performance scalability.

As in graph models, hypergraph vertices represent the work of a simulation. However, hypergraph edges (hyperedges) are sets of two *or more* related vertices (see Figure 9). The number of hyperedges cut by partition boundaries is an *exact* representation of communication volume, not merely an approximation as in graph models [14]. In the example in Figure 9 (right), a single hyperedge (shown in blue) including vertex *A* and its neighbors is associated with *A*; this single cut hyperedge accurately reflects the volume of communication associated with *A*.

Catalyurek and Aykanat [14] also demonstrated the greater expressiveness and applicability of hypergraph models over graph models. Graph models imply symmetry in all relationships, making them appropriate only for problems that can be represented by square, symmetric matrices. Hypergraph models do not imply symmetry in relationships, allowing both non-symmetric and rectangular matrices to be represented. For example, in Figure 10, the vertices of a hypergraph (right) represent rows of a rectangular matrix (left). Each matrix column is represented by one hyperedge connecting all non-zero values in the column. For a two-processor, row-based matrix decomposition (indicated by red and blue), colored hyperedges represent local operations in matrix-vector multiplication, while black hyperedges require interprocessor communication.

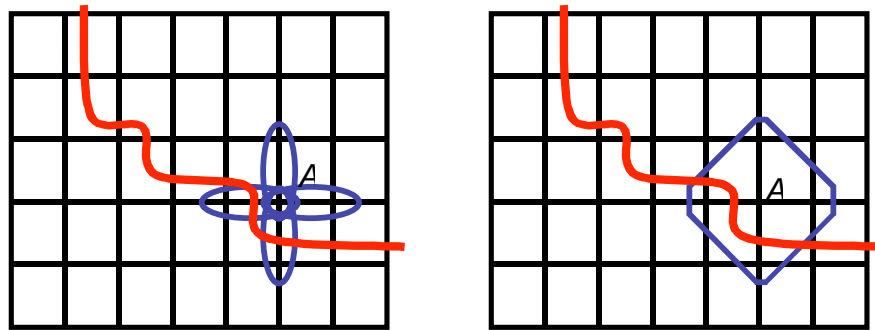


Figure 9 Example of communication metrics in graph partitioning (left) and hypergraph partitioning (right). Edges are shown in blue; the partition boundary is shown in red.

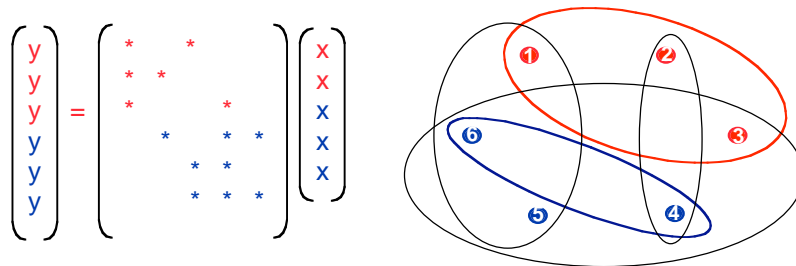


Figure 10 Representation of a rectangular matrix (left) by a hypergraph (right). Matrix rows are represented by hypergraph vertices. Each matrix column is represented by one hyperedge connecting all non-zero values in the column.

Hypergraph partitioning's effectiveness has been demonstrated in a number of areas. It has been used for decades for VLSI layout to put devices into clusters with minimal inter-cluster connections [10]. It has been used effectively for sparse matrix decompositions [14, 5]. It is an important technology in database storage and data mining [15, 41]. Several serial hypergraph partitioners are available (e.g., hMETIS [29], PaToH [13], Mondriaan [5]). However, no parallel hypergraph partitioners exist. Parallel partitioning is needed for two reasons. First, very large data sets (such as those used in most Sandia applications) can overwhelm the capabilities of serial partitioners; parallel hypergraph partitioners can be used to statically decompose such data sets. Second, adaptive simulations require repartitioning to

redistribute work as processor workloads change; this partitioning must be done in parallel to maintain application scalability.

6.1 Multilevel Hypergraph Partitioning

As a precursor to parallel hypergraph partitioning, we have developed a serial hypergraph partitioner in Zoltan. The hypergraph partitioner includes generalizations of many algorithms used in graph partitioners. It uses a multi-level algorithm [22], in which a hypergraph is coarsened into successively smaller hypergraphs by some form of vertex matching. A global optimizing algorithm partitions the smallest hypergraph. The coarse decomposition is then projected back to the larger hypergraphs, with Fiduccia-Mattheyes [20] local optimization used to reduce hyperedge cuts while maintaining balance at each projection.

The coarsening phase uses reduction methods that are variations on graph matching algorithms adapted to hypergraphs. There are three types of reduction methods: matching, packing, and grouping. All reduction methods select a set of vertices and combine them into a single, “larger” vertex. In hypergraph matching, a pair of connected vertices is replaced with an equivalent vertex; the new vertex’s weight, connectivity, and associated hyperedge weights are computed to reasonably represent the original pair of vertices. Packing reduction methods replace all vertices connected by one hyperedge with an equivalent vertex. Grouping reduction methods replace all ungrouped vertices connected by a single hyperedge with an equivalent vertex. Packing differs from grouping because a hyperedge is packed only if no vertices in the hyperedge have been selected for packing in an overlapping hyperedge; grouping allows unselected vertices in the hyperedge to be grouped.

Optimal matching, packing and grouping algorithms are typically very time consuming; they either have run-times that are $O(\text{high-degree polynomial})$ or are NP-complete. Thus, fast heuristics are used to compute good results. We implemented several fast approximation algorithms for these tasks, but the results of the heuristics may lack local optimality structure. Therefore, an optional augmentation algorithm may be applied at the end of the matching, packing, or grouping algorithm to improve the result. Augmentation algorithms improve hypergraph matching, packing or grouping reductions by finding alternative reductions with higher hyperedge weights. Augmentation may remove existing reductions and replace them with alternative ones.

Reduction typically proceeds until the number of coarse vertices equals the number of desired partitions. On some difficult problems, however, it can be more time efficient to terminate the reduction process sooner because only a small fraction of the vertices are being successfully matched/packed/grouped. (Nominally, a matching should pair nearly half of the vertices at each level.) In these cases, coarsening is stopped when it produces less than 10% reduction in the number of vertices compared with the parent hypergraph. In addition, a user parameter `HG_REDUCTION_LIMIT` may be used to set the number of vertices at which the reduction process is stopped. Use of this parameter allows greater flexibility in partitioning the coarsest hypergraph. The coarsest hypergraph is then partitioned. If the coarsest hypergraph has the same number of vertices as the number of requested partitions, each

vertex is trivially assigned to a partition. Otherwise, a global optimizer (using greedy partitioning) establishes the coarse-hypergraph partition.

After the coarse-hypergraph partition is computed, the coarse partition is projected onto the successively finer hypergraphs. A coarse vertex's partition assignment is given to each of the fine vertices that were reduced into the coarse vertex. At each projection, a variation of the Fiduccia-Mattheyes [20] optimizer reduces the hyperedge cut weight while maintaining (or establishing) partition load balance. The local optimizer generates only two partitions ($k=2$). For $k>2$, the entire hypergraph partitioner is applied recursively. We also implemented a greedy direct k -way local optimizer, but it is currently much weaker than the recursive approach; that is, it results in a considerably larger number of hyperedge cuts than the recursive approach. Currently, we are implementing a direct k -way local optimizer based on Kernighan-Lin [36] and Fiduccia-Mattheyes [20]; this approach has the advantage of directly operating on k partitions, with the disadvantage of being a much more complicated strategy than improving a bisection.

6.1.1 Terminology

neighbor vertex: a vertex sharing a hyperedge with a given vertex.

normal vertex (hyperedge) order: The order vertices (hyperedges) are stored in the hypergraph data structure; initially, this order reflects the order in which vertex (hyperedge) data was received from the application. If no ordering is specified, normal order is assumed.

pins: Connections between vertices and hyperedges; given a bipartite graph with vertices on one side and hyperedges on the other, the edges connecting a hyperedge with its vertices represent pins. This nomenclature comes from the electronic circuit community where it corresponds to the physical pins of the components.

$|e|$: Hyperedge size; i.e., the number of vertices in hyperedge e .

$$|e|_{\max} = \max_{\text{all edges } e} (|e|)$$

$|\text{Hyperedges}|$: Total number of hyperedges in a hypergraph.

$|\text{Vertices}|$: Total number of vertices in a hypergraph.

$$|\text{Pins}| = \prod_{\text{all edges } e} |e|$$

$w(e)$: hyperedge weight; a weight assigned to a hyperedge. Unit weights are used if no weights are given by the application. During coarsening, identical hyperedges are combined by summing their weights.

$w(v)$: vertex weight; a weight assigned to a vertex. Unit weights are used if no weights are given by the application. During coarsening, matched vertices' weights are summed to compute the coarse vertex's weight.

k : number of partitions.

6.1.2 Coarsening Strategies

The following reduction algorithms produce maximal matchings in the sense that for matching algorithms, there are no unmatched vertices that share a common hyperedge; for packing algorithms, there is no hyperedge with all vertices unmatched; and for grouping algorithms, there is no hyperedge with more than one unmatched vertex. There may be many distinct maximal matchings, packings, and groupings for the same hypergraph. In graphs, a maximal matching is also at least a 1/2-approximation of a maximum cardinality matching (i.e., the solution cardinality is at least one-half the cardinality of the maximum cardinality matching). Another useful approximation is the ratio of the minimum sum of matched (graph) edge weights to the maximum weight matching. We have begun to extend these concepts from graphs to hypergraphs [48]. When a maximum weight matching approximation is known for an algorithm, it is indicated below.

Matching Reductions

MXM (maximal matching): MXM is a hypergraph version of the graph maximal matching algorithm. It visits all vertices in normal vertex order. If a vertex is unmatched, it becomes the current vertex. The vertices of the current vertex's hyperedges are visited until an unmatched neighbor vertex is found. The neighbor vertex is matched to the current vertex, and the algorithm advances to find the next current vertex. MXM runs in $O(|Pins|)$ time. This method is the simplest, most intuitive matching algorithm. However, it is highly dependent on the normal order of the vertices. It works surprisingly well for circuit problems where the vertices and hyperedges provided by circuit codes (e.g., Xyce [27]) are inherently well-ordered. For other (non-circuit) hypergraphs, this method is weak.

REM (random hyperedge matching): REM is a hypergraph version of the random edge graph-matching algorithm. The algorithm randomly visits all hyperedges. For each hyperedge, it visits each hyperedge vertex to find pairs of unmatched vertices. When two unmatched vertices are found, they are matched together, and the search for more unmatched pairs in the hyperedge continues. After all vertices in the current hyperedge are examined, the algorithm advances to the next random hyperedge. REM runs in $O(|Pins|)$ time. This algorithm uses randomization to reduce the sensitivity of the algorithm to the normal ordering of hyperedges.

RRM (random, random matching): RRM randomly visits all vertices. If the current vertex is unmatched, RRM visits all hyperedges containing the vertex to find unmatched neighbor vertices. From all unmatched neighbors of the current vertex, RRM randomly selects one unmatched neighbor and matches it with the current vertex. The algorithm then advances to the next random vertex. RRM runs in $O(|Pins| * |e|_{\max})$ time. This algorithm uses randomization to reduce the sensitivity to orderings of both vertices and hyperedges.

RM2 (random, random matching, version 2): RM2 is similar to **RRM** except that only the unmatched neighbor vertices in the lightest weight hyperedge (hyperedges in case of ties) are considered for matching. The idea is to keep highly localized circuit elements (here

represented by light hyperedges) together during the coarsening phase and thus to force splitting circuits between blocks of circuit elements (represented by heavy hyperedges). Although this strategy should produce a poor graph partitioning algorithm, it yielded unusually good partitioning results for circuit problems when we used it with the augmentation algorithm **aug2**. The other matching, packing, and grouping algorithms whose name ends in a number were further experiments along this line. RM2 runs in $O(|Pins| * |e|_{\max})$ time.

GRM (greedy random matching): GRM is a hypergraph version of the greedy graph-matching algorithm. GRM sorts the hyperedges in decreasing order by weight and, in case of ties, by decreasing hyperedge size. The hyperedges are then visited in sorted order. All of the vertices in the current hyperedge are visited. As two unmatched vertices are found, they are matched together. The need to sort makes this algorithm slower than the previously described reduction methods. It runs in $O(|Pins| + |Hyperedges| * \log(|Hyperedges|))$ time and is a 1/2-approximation to the maximum weight matching.

GM2 (greedy random matching, version 2): GM2 is similar to GRM except that the hyperedge weights are sorted in increasing order (and increasing hyperedge size, in case of ties). This modification is intended to partition circuits between blocks of circuit elements rather than through the blocks. GM2 runs in $O(|Pins| + |Hyperedges| * \log(|Hyperedges|))$ time.

GM3 (greedy random matching, version 3): GM3 is similar to GRM except that the hyperedge weights are sorted by the ratio of hyperedge weights to hyperedge size. Hyperedges are visited by decreasing weight/size ratio. GM3 runs in $O(|Pins| + |Hyperedges| * \log(|Hyperedges|))$ time.

GM4 (greedy random matching, version 4): GM4 is similar to GM3 except that the hyperedge weight-to-size ratios are sorted in increasing order. GM4 runs in $O(|Pins| + |Hyperedges| * \log(|Hyperedges|))$ time.

LHM (local heavy matching): LHM is a locally heavy edge-matching algorithm based on Preis' linear-time 1/2-approximation graph matching algorithm (LAM) [46]. LHM begins by converting the hypergraph to a graph by creating a graph clique for each hyperedge; the weight of hyperedge e is distributed to the resulting graph edges so that each graph edge has weight equal to $2w(e) / (|e|^2 - |e|)$. This conversion makes LHM slower than LAM and most other hypergraph reduction methods.

LHM starts with an empty set of matched (graph) edges. Starting from an unmatched vertex in normal vertex order, the edges containing this vertex are searched in normal order to find an unmatched edge (i.e., an edge with two unmatched vertices), which becomes the current edge. LHM begins a backtracking search for adjacent edges with a higher weight. That is, it looks at the current edge and each neighboring edge (i.e., an edge sharing a vertex with the current edge) and selects the unmatched edge with the highest edge weight; if a neighbor had the highest edge weight, the neighbor edge becomes the new current edge, and a new search starting from this edge is made. At the end of the backtracking search, the weight of the current edge is at least as high as all available adjacent edges. This edge is added to the set

of matched edges and its vertices are marked as matched. The next unmatched vertex in normal order is selected to begin the next search.

By choosing the locally heaviest edges, LHM produces a 1/2-approximation to the maximum weight matching. To reduce the repeated examination of edges, LHM uses a recursive trial matching/search routine to keep track of the set of tested edges. Edges may be checked several times using this recursive backtracking, but the average number of comparisons is linear in the number of graph edges. Overall runtime is $O(|Pins| * |e|_{\max})$ due to the conversion from hypergraph to graph edges.

PGM (path growing matching): PGM is a hypergraph version of the path-growing graph-matching algorithm in [19]. It visits all vertices in normal order, building a set of disjoint paths (sequences of vertices) from unmatched vertices. For each starting vertex, PGM creates graph edges and their weights by converting hyperedges to their associated graph edges logically (not actually building the complete graph) using the routine *Sim*. *Sim* creates graph edges from a vertex by expanding hyperedges containing the vertex as a clique (with appropriate weighting). A path grows from the starting vertex by appending the heaviest (graph) edge with an unmatched neighbor; path growing terminates when no unmatched neighbor vertices exist. Within each path, two sets of edges are then considered: the first set contains every other edge along the path starting from the initial vertex, while the second set contains the remaining path edges. The set with the larger sum of edge weights is selected, and vertices connected by those edges are matched together. This algorithm is a 1/2-approximation to the maximum weight matching. PGM runs in $O(|Pins| * |e|_{\max})$ time.

RHM (random heavy hyperedge matching): RHM is a hypergraph version of random heavy-edge graph-matching algorithm. All vertices are visited in random order. If a vertex is unmatched, it becomes the current vertex. Code similar to *Sim* (described above) is used to compute the equivalent graph edges and weights for the hyperedges containing the vertex. The current vertex is then matched with a vertex that is randomly selected from all unmatched neighbor vertices in the (graph) edges with the largest weight. The randomness reduces the sensitivity to the original vertex ordering. Some variation of RHM is the default matching in most hypergraph packages (such as hMETIS [29]). RHM runs in $O(|Pins| * |e|_{\max})$ time.

Packing Reductions

MXP (maximal packing): This algorithm visits all hyperedges in hyperedge order. If all vertices in the current hyperedge are unmatched, MXP matches all the vertices together to produce one equivalent vertex. MXP runs in $O(|Pins|)$ time.

REP (random hyperedge packing): All hyperedges are visited in random order. If all vertices in the current hyperedge are unmatched, REP matches the vertices together to produce one equivalent vertex. REP runs in $O(|Pins|)$ time.

RRP (random, random packing): RRP randomly visits every vertex. If the current vertex is unmatched, it then visits all hyperedges containing this vertex. A random hyperedge is selected from the set of hyperedges containing the current vertex and only unmatched

vertices. If such a hyperedge exists, all vertices in that hyperedge are matched together. RRP runs in $O(|Pins|)$ time.

RHP (random heavy hyperedge packing): RHP randomly visits all vertices. The set of the heaviest hyperedges containing the current vertex and only unmatched vertices is created. A hyperedge is randomly selected from this set, and all its vertices are matched together. RHP runs in $O(|Pins|)$ time.

GRP (greedy packing): GRP is a greedy algorithm that sorts all hyperedges by decreasing hyperedge weight (and, in case of ties, decreasing hyperedge size). Hyperedges are visited in sorted order. If all vertices in the current hyperedge are unmatched, they are all matched together. GRP runs in $O(|Pins| + |Hyperedges| \cdot \log(|Hyperedges|))$ time and guarantees a $1/|e|_{\max}$ -approximation to the maximum weight set packing.

LHP (local heavy packing): LHP is similar to LHM, but it works directly with hyperedges rather than converting the hypergraph to a graph. It finds a locally heaviest edge by adding the heaviest edge along a path of unmatched edges to the set of matched edges, marking its vertices as matched. Each search starts from the next unmatched vertex in normal vertex order. A recursive search routine maintains a list of examined edges and vertices for each search to minimize the number of times edges and vertices are examined. Edges may be checked several times during the recursive backtracking, but average costs are $O(|Pins| \cdot |e|_{\max})$ time. LHP guarantees a $1/|e|_{\max}$ -approximation to the maximum weight set packing.

PGP (path-growing packing): PGP is a generalization of PGM. While PGM constructs a set of paths along graph edges, PGP constructs paths of intersecting hyperedges. PGP visits all hyperedges in normal order. If the current hyperedge is not yet in a path, it is added to the path. PGP then considers all hyperedges intersecting (i.e., sharing a vertex with) the current edge and adds the heaviest intersecting hyperedge to the path. This intersecting hyperedge becomes the current hyperedge, and the path growing is repeated until the current hyperedge has no available intersecting hyperedges. PGP then continues with the next hyperedge in normal order. The resulting set of paths can be split into two disjoint sets of hyperedges; the one with the higher weight is used for the hypergraph reduction. PGP runs in time $O(|Pins|)$ and gives an approximation of $1/(2(|e|_{\max}-1))$ for the maximum weight set packing.

Our implementation of PGP generates two disjoint sets of hyperedges by using a flag (called *side*) to indicate to which disjoint set a hyperedge should be added. Each set has an associated accumulated weight and packing array. When a hyperedge is chosen for a path, it is added to the set indicate by the *side* flag, and the associated packing array is marked to pack together all vertices in the hyperedge. The hyperedge and all its vertices are marked to indicate to which set they are assigned and the hyperedge's weight is added to the set's total weight. The *side* flag is then changed so that the next hyperedge on the path is added to the opposite set. At the end of the algorithm, the set with the higher accumulated weight is selected.

Grouping Reductions

MXG (maximal grouping): In MXG, each hyperedge is visited in normal order. All unmatched vertices contained in the current hyperedge are matched together. MXG runs in $O(|Pins|)$ time.

REG (random hyperedge grouping): In REG, all hyperedges are visited in random order. All unmatched vertices contained in the current hyperedge are matched together. REG runs in $O(|Pins|)$ time.

RRG (random, random grouping): RRG visits all vertices in random order. It then randomly selects a hyperedge from the hyperedges containing the current vertex. All unmatched vertices in this hyperedge are matched together. RRG runs in $O(|Pins|)$ time.

RHG (random heavy hyperedge grouping): In RHG, all vertices are visited in random order. If the current vertex is unmatched, a hyperedge is randomly selected from the set of the heaviest hyperedges containing the current vertex. All unmatched vertices in the selected hyperedge are matched together. RHG runs in $O(|Pins|)$ time.

GRG (greedy grouping): In GRG, all hyperedges are sorted by decreasing hyperedge weight (and hyperedge size in case of ties). Hyperedges are visited in sorted order. All unmatched vertices in the current hyperedge are matched together. GRG runs in $O(|Pins| + |Hyperedges| * \log(|Hyperedges|))$.

6.1.3 Augmentation Strategies

We implemented three augmentation algorithms (**aug1**, **aug2**, and **aug3**) to improve the cardinality of the reductions computed by the reduction algorithms. Augmentation strategies for matchings were modified for packing and grouping to allow more than two vertices to be matched together. Starting from a selected vertex, **aug1**, **aug2** and **aug3** try to grow (augment) a path of length one, two, or three hyperedges, respectively, along the heaviest neighboring hyperedges. If an augmented path has a heavier total hyperedge weight than any current path of the same length starting from the original vertex, then the original reductions involving the vertices are undone and replaced by the augmentation-induced reductions. Augmentation method **aug2** is the default method.

Augmenting paths improve the approximation to the maximum cardinality matching. If M_l is the minimum cardinality matching and M_{MCM} is the maximum cardinality matching, then

$$|M_l| \geq \frac{l+1}{l+1} |M_{MCM}|$$

where l is the length of the augmenting path [46]. Augmentation algorithm **aug1** was essentially equivalent to the maximal matching algorithm **mxm**. It was discarded because it would improve only non-maximal reductions, and all the implemented reduction methods are maximal.

Augmentation method **aug3** has not been converted to directly use a hypergraph. It is not currently used due to the expense of converting the hypergraph to a graph, but we should be able to rewrite **aug3** in terms of *Sim*.

6.1.4 Scaling Strategies

Graph and hypergraph matchings occasionally suffer a pathological condition where a vertex or hyperedge becomes so dominant in the coarsening process that eventually a “star-like” hypergraph develops; that is, all of the remaining vertices connect to the dominant vertex, but not to each other. During further matching, packing, or grouping, only one vertex can be matched to the dominant. Thus, rather than matching roughly half of the vertices in each level of the multilevel algorithm, the reduction algorithm finds only one (or very few) matches. Practically, the coarsening phase must be terminated early, resulting in fewer levels for the local optimizer to produce a good partition. A heuristic to scale the hyperedge weights sometimes leads to preventing or slowing this star-like formation. These hyperedge scalings are applied once on each coarsening level prior to calling the reduction algorithm. After the reduction algorithm creates its matchings, the hyperedge weights are returned to their original values.

Note: The hypergraph partitioner needs to scale vertex and hyperedge weights at other several points to create the equivalent coarsened vertices and their hyperedges. These scalings are done inline in the applicable code and are not the subject of this section.

There are five hypergraph scalings and four graph scalings. (The graph scalings were created when the reduction methods were still graph-based, rather than hypergraph-based; they are not currently used.) The five hypergraph scalings are numbered rather than named. They divide the weight of a hyperedge by the (1) hyperedge size, (2) the product of all of the hyperedge’s vertex weights, (3) the sum of the vertex weights of the hyperedge, (4) the maximum vertex weight in the hyperedge, or (5) the minimum vertex weight in the hyperedge. The scaling method (1) is the current default method.

6.1.5 Global (Coarse) Partitioning Strategies

After the hypergraph has been reduced to a fairly small size in the multilevel scheme, we need to partition this coarse hypergraph into k partitions. One option is to require the coarse hypergraph to have exactly k vertices, and then simply assign one vertex to each partition. However, experience from graph partitioning has shown that it is often advantageous to stop coarsening earlier, i.e., with more than k vertices left. We have, therefore, implemented several so-called global methods to partition the coarse hypergraph, and these are briefly explained below.

RAN (Sequence partitioning, random order): RAN ignores the hypergraph structure. The vertices are randomly ordered, and the partitions are formed by groups of consecutive

vertices. Approximate load balance is achieved by computing the cumulative sum of the weights, and starting partition j when the cumulative sum is greater than $W^*(j/k)$, where W is the sum of all weights.

LIN (Sequence partitioning, linear order): LIN is similar to RAN except the vertices are ordered by their vertex numbers (labels).

BFS (Breadth-first-search): BFS first finds a pseudo-peripheral start vertex (that is, a vertex for which there exists some vertex whose distance from the start vertex is almost the diameter of the graph) by breadth-first search from a random vertex. Then, it computes a breadth-first search ordering from this start vertex. This order is used to do sequence partitioning as in LIN and RAN.

RBFS (Restarted BFS): RBFS is similar to BFS except the BFS algorithm is restarted whenever we start a new partition. Note that for bisection ($k=2$), this method is identical to BFS.

BFSH (BFS with heavy edges first): BFSH visits hyperedges in the BFS algorithm in decreasing order by weight.

RBFSH (Restarted BFSH): RBFSH is a restarted version of BFSH.

The remaining methods are all of greedy type, and thus named GR_x for some integer x . The idea here is to greedily add vertices to a partition until the partition has reached the right size. We always start at a pseudo-peripheral vertex, which is found by doing BFS from a random node. Each vertex that has not yet been assigned to a partition has a certain “preference value” with respect to the current partition. Different preference functions yield different variations. Intuitively, the more connected a vertex is to the current partition, the higher the preference value should be. An alternative is to estimate the hyperedge cut size. For efficiency, a priority queue is maintained using a heap so that the vertex with the highest preference value can be found quickly.

GR0 (Greedy method 0): The preference function is the decrease in total cut size for the partitioning formed so far. This corresponds to the gain value in the FM algorithm.

GR1 (Greedy method 1): This variation is closely related to the “absorption” metric used in the circuit community. Let S be the set of vertices in the current partition and let v be an unassigned vertex. The preference function $p(S, v)$ is given by

$$p(S, v) = \sum_e w(e) \frac{|e \cap S|}{|e|},$$
 where e denotes any hyperedge containing v , $|e \cap S|$ is the number of vertices in both e and S , and $w(e)$ is the weight of hyperedge e .

GR2 (Greedy method 2): GR2 is a scaled version of GR1. The preference function is

$$p(S,v) = \prod_e \frac{w(e)}{esum(v)} \frac{|e \cap S|}{|e|}, \text{ where } e \text{ denotes any hyperedge containing } v, \text{ and}$$

$$esum(v) = \prod_j w(j) \text{ for all hyperedges } j \text{ containing } v.$$

GR3 (Greedy method 3): The intuition behind this method is that it is good to add a vertex to the current partition if the hyperedges covering that vertex already are mostly contained in the current partition. Specifically, the larger the fraction of a hyperedge is contained in the current partition, the better. An exponentially damped weighting formula is used:

$$p(S,v) = \prod_e w(e) * 2^{\lfloor |e \cap S| / |e| \rfloor}, \text{ where } e \text{ denotes any hyperedge containing } v.$$

GR4 (Greedy method 4): This method is a scaled version of GR3.

Limited experiments indicate that the greedy methods generally perform better than the more basic methods. The GR0 method appeared to be the most consistent overall and is currently the default method. GR1-4 perform better than GR0 on some hypergraphs. We have not performed any extensive empirical study. The choice of global method may be of little importance, since global partitioning is always followed by a refinement phase.

6.1.6 Refinement Strategies

FM2: FM2 is a version of the Fiduccia-Mattheyses (FM) [20] graph optimizer that improves the edge-cut and balance of a bipartition ($k=2$). FM2 is applied to a fine hypergraph after every projection of a coarse hypergraph partition to a finer hypergraph. The local optimizer selects vertices to be moved between the two partitions, measuring each move's gain (the difference between total cut-edge weights before and after the move). Unlike the classical FM algorithm, which uses a fixed number of gain buckets, this version stores the moves in heaps ordered by gain. The heaps allow arbitrary floating-point gains. The gains are computed and placed in the heap associated with the target partition. The heaps allow easy access to the highest gain move.

The local optimizer makes a series of vertex moves, attempting to find an improved partition. At each step, both heaps are checked to find the move with the highest gain that maintains the required balance. The vertex associated with this move is assigned to the target partition, and the gains of its neighboring vertices are modified to reflect the vertex's move. A stack stores the sequence of vertex moves. The move that produces the lowest edge-cut weight is marked. When either both heaps are empty or the maximum allowed number of moves (steps) has been taken since the last minimum, no more moves are attempted. Moves that occurred after the marked minimum-edge-cut move are undone. The minimum-edge-cut configuration is accepted, and another pass of FM begins from that configuration. This process continues until either no further improvement is found or until the maximum number of passes is reached.

GRKWAY: GRKWAY is a parallelizable greedy optimizer based on the graph algorithm by Karypis and Kumar [33]. Each processor stores all hyperedges (and their associated vertices) that intersect its domain. Like FM2, the local optimizer makes a series of vertex moves, attempting to find an improved partition. The vertex moves are divided into two phases. Each phase has two communications.

In the first phase, each processor computes the gain for all possible vertex moves to all other processors and saves the highest-gain move for each vertex. For each positive-gain move to a higher-numbered processor, the processor sends the move to the target processor. Each processor receives messages from lower-numbered processors and stores those moves in a heap. Moves are then taken off the heap (highest gain first) and their vertices are assigned to the receiving processor until the partition reaches its maximum total vertex weight. All moves to this point are successful and the receiving processor becomes the new owner of those vertices. Remaining moves in the heap are unsuccessful. Messages are returned to the sending processors indicating whether each move was successful. Each processor then updates its ownership, total vertex weight, and gain information based on the return messages.

The second phase is nearly identical to the first, except that the move messages are sent only to lower-numbered processors.

Since this algorithm makes only moves with positive gain, it has no hill climbing capability to escape local extrema. (Fortunately, the V-cycle itself provides new opportunities at each level to find better solutions.) The actual minimum of weighted hyperedge cuts is achieved when all vertices are in the same partition, leaving no cuts at all. When a partition is initially very unbalanced, most gains favor moving toward this extreme. Since this decomposition violates the imbalance tolerance, the actual moves are not made, but there are no alternative moves to restore balance. It is necessary to have a preprocessing phase to force a legal balance. Then the algorithm has many available legal moves. Currently GRKWAY's decompositions are not as good as FM2's, but GRKWAY can be made to run in parallel. It is not clear that FM2 can be made to run effectively in parallel.

6.2 Hypergraph Partitioning Results

6.2.1 Design of Experiment

The large number of reduction methods, augmentation methods, scaling methods, global optimizers, and internal parameters (e.g., the maximum number of steps since the last minimum in the local FM optimization method) makes exhaustive testing of all possible combinations unrealistic. The best combination of methods also depends upon the actual problem data. In our hypergraph research, we are using design-of-experiment techniques to determine these parameters and select from competing algorithms to produce a robust hypergraph partitioner. Our goal is to enable the hypergraph partitioner to produce acceptably good results over the widest possible range of problems and problems sizes.

Our design-of-experiment research uses a standard IBM test suite of 18 circuit hypergraphs, ISPD98 [1]. Our goal is to select combinations of methods that minimize the weight of cut

hyperedges induced by a two-way partition while maintaining the specified vertex weight imbalance.

The original experiments showed no statistically significant difference between the original 17 reduction methods over the 18 test problems. These experiments were used to set the default scaling methods, augmentation methods, and major internal parameters.

Table 6 shows the benefit of our early design-of-experiment tests. For each test hypergraph, the best Zoltan results before our design-of-experiment tests are compared to the best results after algorithmic adjustments made as a result of the design-of-experiment tests. The changes guided by the design-of-experiment tests produced edge-cut reductions in 16 out of 18 tests. The average total edge cuts (over 23 reduction methods in Zoltan) after the design-of-experiment modifications are also reported. (After obtaining these results, we changed the design-of-experiment methodology to improve the average, rather than the best, results. This change was made to improve the robustness of the algorithms and to avoid “tuning” the algorithms to a specific class of problems.)

Table 6 also includes results verifying the competitiveness of the Zoltan hypergraph algorithms with the widely used hMETIS hypergraph partitioner [29]. The best Zoltan test results after the design-of-experiment modifications are compared with published results from hMETIS [30]. Like the Zoltan results, the published hMETIS results represent the lowest number of edge cuts obtained from many different algorithms and parameters in multiple runs of hMETIS. The Zoltan hypergraph methods proved to be competitive with hMETIS; total edge cuts for Zoltan range from 13.7% more cuts than hMETIS to 7.4% fewer.

Input Hypergraph	Number of Cut Hyperedges				% Reduction in Edge Cuts	
	hMETIS (published)	Zoltan best: Before DoE	Zoltan best: After DoE	Zoltan avg: After DoE	(Zoltan best after DoE vs. before DoE)	(Zoltan best after DoE vs. hMETIS)
IBM01	243	240	225	308	6.25	7.41
IBM02	272	306	262	342	14.38	3.68
IBM03	781	1002	797	1100	20.46	-2.05
IBM04	440	451	447	554	0.89	-1.59
IBM05	1718	1751	1741	2036	0.57	-1.34
IBM06	376	788	370	704	53.05	1.60
IBM07	762	763	745	856	2.36	2.23
IBM08	1157	1200	1173	1513	2.25	-1.38
IBM09	523	531	529	980	0.38	-1.15
IBM10	778	901	831	1415	7.77	-6.81
IBM11	701	735	797	1047	-8.44	-13.69
IBM12	2006	2037	2020	2507	0.83	-0.70
IBM13	884	917	888	1444	3.16	-0.45
IBM14	1636	1687	1615	2399	4.27	1.28
IBM15	1809	2427	1854	3483	23.61	-2.49
IBM16	1723	2135	1733	2937	18.83	-0.58
IBM17	2397	2392	2382	3548	0.42	0.63
IBM18	1539	1615	1620	2367	-0.31	-5.26

Table 6 Design of Experiment results for Zoltan hypergraph partitioners

The experiments then focused on the FM local optimizer **fm** (the predecessor to **fm2**), since any significant improvement to the local optimizer reduces the importance of other choices and makes the hypergraph partitioner more robust for arbitrary inputs. In these experiments, we minimized the sum of the hyperedge cuts for all 23 reduction methods applied to the same test problem with all other factors identical. We used a 95% confidence interval to establish statistical significance. That is, 15 or more of the 18 test cases must show a decrease in the test statistic for success; four or fewer must show an increase for an equally significant negative result.

A literature search provided many “tricks” for improving hypergraph partitioning:

- Temporarily increasing the imbalance tolerance [11];
- Limiting the maximum number of FM passes on a level and exiting early from FM passes [31];
- Controlling how long a vertex is locked before allowing it to move (several heuristics) [4];
- Multiple unlocking: allowing vertices to move more than once during a FM pass, allowing illegal solutions during a pass, and applying several tie-breaking heuristics [12];
- Using tie-breaking strategies for selecting vertices to move [9]; and
- Relaxing vertex locking of vertex moves (2 algorithms) [17].

We tested many of the heuristics above; none demonstrated a statistically significant improvement. The first heuristic (temporarily increasing the imbalance tolerance) showed a statistically significant negative result.

During this research, we developed three new heuristics and verified one known heuristic that passed this statistical test (in order of decreasing statistical significance):

- Allow each FM pass to process all vertices. Naturally, moving all vertices significantly increases the processing time and is therefore not practical. This test verified a known result. Our experiments showed the best approach is to limit the number of moves to 25% of the number of vertices after no improvement is found during a pass.
- Tighten the imbalance tolerance below the seventh level of the V-cycle to 50% of the specified imbalance tolerance. The specified imbalance tolerance is thus abruptly restored at level 7 and maintained until the V-cycle is complete. This approach is the opposite of the published heuristic that had the significantly negative result.
- Perform one additional round of FM after the normal FM exit.
- Require at least 3 rounds of FM at each level.

At a later time, we developed two more heuristics that were significantly significant. Since the FM baseline code had been improved by the above heuristics, the new heuristics were not ranked against them:

- We discovered a small number of vertices were in the opposite partition as all of their neighbors. We moved these vertices to the partition containing their neighbors. This heuristic was expanded to force the movement of vertices with a very large percentage (specified by an experimental parameter) of their neighbors in the opposite partition. We found that this heuristic needed to be applied only after the V-cycle was complete since FM itself moves many of these “orphans” in upper levels of the V-cycle.
- We alternated our FM local optimizer with a greedy optimizer at each level. This strategy provided better results than using either optimizer alone. The greedy FM is the baseline FM with one change: it always allows the first move (highest gain move) even if the move violates the imbalance tolerance.

With these heuristics, both the test statistic and its standard deviation improved, yielding the test results shown in Table 6. In particular, the improvement of the standard deviation indicated that the FM algorithm was becoming more robust.

Since the hypergraph algorithms will need extensive changes to run effectively in parallel, these heuristics may no longer be valid. The current version of FM in Zoltan is FM2. FM2 has none of the above heuristics since it will be the springboard to the parallel version. These and other heuristics will be tested on the parallel version of FM to improve its performance. The real purpose of the experimental design tests was to validate the design-of-experiment

methodology and automate the tests and analysis for use in developing a parallel hypergraph partitioner.

6.2.2 Hypergraph Partitioning in Sandia Applications

To demonstrate the effectiveness of hypergraph partitioning for emerging applications, we applied both graph- and hypergraph partitioners to a variety of matrices and compared the resulting communication volume required for matrix-vector multiplication using the resulting decompositions. For both methods, each row i of the matrix A was represented by node n_i . For hypergraph methods, each column j was represented by a hyperedge he_j with $he_j = \{n_i : a_{ij} \neq 0\}$ (e.g., see Figure 10). For graph methods, edge e_{ij} connecting nodes n_i and n_j existed for each non-zero a_{ij} or a_{ji} in the matrix A ; that is, the graph represented the symmetrized matrix $A+A^T$. This symmetrization was necessary because undirected graphs cannot represent non-symmetric matrices.

The first matrix, HexFEM, comes from a standard 3D hexahedral finite element simulation; it is symmetric and very sparse. This type of matrix is represented reasonably well by graphs, as graphs accurately represent symmetric systems and provide reasonable approximations to communication required for very sparse matrices. As a result, hypergraph partitioning offers little additional benefit compared to graph partitioning for this example. For a five-partition decomposition, hypergraph partitioning reduced total communication volume 2-22% compared to graph partitioning. Detailed results of HexFEM experiments are in Table 7.

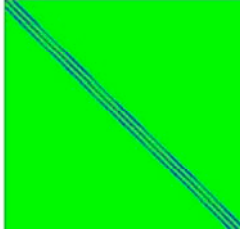
HexFEM Matrix: <ul style="list-style-type: none"> • Hexahedral 3D structured-mesh finite element method. • 32,768 rows • 830,584 non-zeros • Five partitions 						
Partitioning Method	Imbalance (Max / Avg Work)	Number of Neighboring Partitions per Partition		Communication Volume over all Partitions		Reduction of Total Communication Volume
		Max	Avg	Max	Total	
Graph partitioning (ParMETIS PartKWay)	1.03	4	3.6	1659	6790	
Best Zoltan hypergraph method (RRM)	1.013	4	3.6	1164	5270	22%
Worst Zoltan hypergraph method (RHP)	1.019	4	2.8	2209	6644	2%

Table 7 Results comparing graph and hypergraph partitioning methods for the HexFEM matrix.

A second matrix, FluidDFT, comes from a fluid simulation in the Density Functional Theory code Tramonto [21]. It has 1643 rows and 1,167,426 non-zeros. Because of its greater density, hypergraph partitioning shows some reduction in communication volume over graph partitioning. For a twelve-partition decomposition, hypergraph partitioning reduced total communication volume 15-33% compared to graph partitioning. Detailed results of FluidDFT experiments are in Table 8.

FluidDFT Matrix: <ul style="list-style-type: none"> • Fluid simulation matrix from Tramonto • 1643 rows • 1,167,426 non-zeros • Twelve partitions 						
Partitioning Method	Imbalance (Max / Avg Work)	Number of Neighboring Partitions per Partition		Communication Volume over all Partitions		Reduction of Total Communication Volume
		Max	Avg	Max	Total	
Graph partitioning (ParMETIS PartKWay)	1.037	11	11	1276	13,506	
Best Zoltan hypergraph method (GRG)	1.037	9	6.8	1184	9,055	33%
Worst Zoltan hypergraph method (REM)	1.037	9	7.5	1260	11,376	15%

Table 8 Results comparing graph and hypergraph partitioning methods for the FluidDFT matrix. Matrix structure is shown in the upper right, with a detailed view of a sub-matrix to the right of the entire matrix

Even greater benefit of hypergraph partitioning is seen in the third matrix, PolymerDFT, which comes from a polymer self-assembly simulation in Tramonto. PolymerDFT has 46,176 rows with 3,690,048 non-zeros in an intricate sparsity pattern arising from the very wide stencil used in the DFT computations. For an eight-partition decomposition, hypergraph partitioning reduced total communication volume 37-56% relative to graph partitioning. With hypergraph partitioning, the number of neighboring partitions was also reduced. Detailed results of the PolymerDFT experiments are in Table 9.

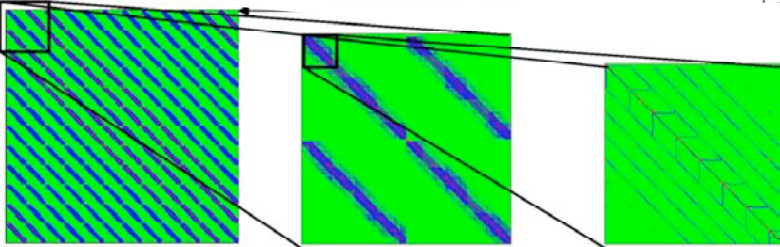
PolymerDFT Matrix: <ul style="list-style-type: none"> • Polymer self-assembly matrix from Tramonto • 46,176 rows • 3,690,048 non-zeros • Eight partitions 						
Partitioning Method	Imbalance (Max / Avg Work)	Number of Neighboring Partitions per Partition		Communication Volume over all Partitions		Reduction of Total Communication Volume
		Max	Avg	Max	Total	
Graph partitioning (ParMETIS PartKWay)	1.03	7	6	7382	44,994	
Best Zoltan hypergraph method (MXG)	1.018	5	4	3493	19,427	56%
Worst Zoltan hypergraph method (GRP)	1.03	6	5.25	5193	28,067	37%

Table 9. Results comparing graph and hypergraph partitioning methods for the PolymerDFT matrix. Matrix structure is shown in the upper right, with detailed views of sub-matrices to the right of the entire matrix.

Given the effectiveness of hypergraph methods for these emerging simulations, we will pursue development of a parallel hypergraph partitioner within Zoltan. This partitioner will include the first parallel implementation of hypergraph partitioning algorithms. It will provide critical capabilities for large-scale, emerging applications at Sandia.

6.3 Future Work

Hypergraph partitioning promises greater accuracy and more effective partitioning than the graph partitioners commonly used today. By accurately modeling communication volume, it provides more efficient decompositions for emerging applications such as circuit modeling and computational biology simulations. Its ability to represent rectangular and non-symmetric systems allows it to be used for a much broader range of applications, including interior point methods and least-squares methods. Using our serial hypergraph algorithms as a starting point, we will develop a parallel hypergraph partitioner. This partitioner will be the first parallel implementation of these methods, providing parallel and dynamic partitioning to Sandia's large-scale simulations. Using our hypergraph framework, we will also complete

implementation of the iterative diffusive schemes proposed by Pinar [43, 44]. These capabilities will be delivered to applications through interfaces in both Trilinos and Zoltan.

7 Conclusions and Future Work

Effective distribution of data to processors is a critical capability for efficient linear solvers and preconditioners. This report documents development of partitioning infrastructure and strategies for optimal solver performance. Parallel data redistribution capabilities were added to the Trilinos solver framework through new classes and methods in the Petra linear algebra package. Efficient parallel directory and unstructured communication utilities were added to the Zoltan parallel data management toolkit, along with appropriate interfaces to them from Petra. We developed new strategies for multi-criteria geometric partitioning and hypergraph partitioning and performed experiments demonstrating their effectiveness for new classes of applications.

Improvements to the multi-criteria geometric partitioners will allow these methods to be more effective for many applications of interest to Sandia. For example, multi-criteria geometric partitioning may prove to be effective for some crash simulations, where, currently, separate decompositions are used for surface and volume computations. It may also improve the performance of applications requiring both balanced computation and memory usage.

The hypergraph partitioning work begun in this effort is the basis for the development of the first parallel hypergraph partitioner. Parallelization will enable hypergraph partitioning to be used for problems too large to be partitioned serially and for problems requiring dynamic partitioning to adjust processor workloads as computations proceed. Hypergraph partitioning promises to play an important role in emerging Sandia applications, where matrices are often non-symmetric, highly connected, and/or rectangular. Delivery of this capability directly through Zoltan and indirectly through Trilinos will allow our continuing work to impact the largest number of Sandia applications.

8 References

1. C. Alpert. "The ISPD98 Circuit Benchmark Suite." *International Symposium on Physical Design*, pp 18-25, ACM, April 1998.
2. S. Balay, W. Gropp, L. McInnes, and B. Smith. *PETSc 2.0 users manual*. Tech. Rep. ANL-95/11 - Revision 2.0.22, Argonne National Laboratory.
3. S. Balay, W. Gropp, L. McInnes, and B. Smith. PETSc home page. <http://www.mcs.anl.gov/petsc>.
4. R. Battiti and A. Bertossi. "Greedy, Prohibition, and Reactive Heuristics for Graph Partitioning." *IEEE Transactions on Computers*, Vol 48, No 4, April 1999.
5. R. Bisseling. *MONDRIAAN*. <http://www.math.uu.nl/people/bisseling/Mondriaan/mondriaan.html>
6. M.W. Boldt, M.W. *An Excursion in High-level Mathematical Modeling through Convection-Diffusion Model Problems*, Honors Thesis. Advisor: Michael A. Heroux, Saint John's University, Collegetown, MN, 2003.
7. E. Boman, K. Devine, R. Heaphy, B. Hendrickson, W. Mitchell and C. Vaughan. "Zoltan: A Dynamic Load-Balancing Library for Parallel Applications – User's Guide." Sandia National Labs. Tech. Rep. SAND99-1377, Albuquerque, NM, 1999. http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.
8. L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Jemmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users' Guide*. SIAM Pub, 1997.
9. A. Caldwell, A. Kahng, A. Kennings, and J. Markov. "Hypergraph Partitioning for VLSI CAD: Methodology for Heuristic Development, Experimenting and Reporting." *Proc. ACM/IEEE Design Automation Conf.*, pp. 349-354, June 1999.
10. A. Caldwell, A. Kahng, and J. Markov. "Design and Implementation of Move-Based Heuristics for VLSI Partitioning." *ACM Journal on Experimental Algorithms*, **5**, 2000.
11. A. Caldwell, A. Kahng, and J. Markov. "Improved Algorithms for Hypergraph Bipartitioning." *ASPDAC '00*, ACM/IEEE, pp. 661-666. January, 2000.
12. A. Caldwell, A. Kahng, and J. Markov. "Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning." *Proc. Workshop on Algorithm Engineering and Experimentation (ALENEX)*, January, 1999.
13. U. Catalyurek and C. Aykanat. *PaToH*. <http://www.cs.umd.edu/~umit/software.htm>.
14. U. Catalyurek and C. Aykanat. "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication." *IEEE Transactions on Parallel and Distributed Systems*, 10(7): 673–693, 1999.
15. C. Chang, T. Kurc, A. Sussman, U. Catalyurek, and J. Saltz. "A Hypergraph-Based Workload Partitioning Strategy for Parallel Data Aggregation." *Procs. of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, March 2001.
16. E. Chow, T. Eliassi-Rad, and V. Henson (with B. Hendrickson, A. Pinar, and A. Pothen). *Parallel graph algorithms for complex networks*. FY04 LDRD Proposal, LLNL, 2003.
17. A. Dasdan and C. Aykanat. "Two Novel Multiway Circuit Partitioning Algorithms Using Relaxed Locking." *IEEE Trans. CAD*, v16, n2, February 1997.
18. K. Devine, E. Boman, R. Heaphy, B. Hendrickson, W. Mitchell and C. Vaughan. *Zoltan home page*. <http://www.cs.sandia.gov/Zoltan>.

19. D. Drake and S. Hougardy. "A Simple Approximation Algorithm for the Weighted Matching Problem." *Information Processing Letters*, 85 (2003) 211-213.
20. C.M. Fiduccia and R.M. Mattheyses. *A Linear Time Heuristics for improving Network Partitions*. Proc. ACM-IEEE Design Automation Conf., 1982.
21. L.J.D. Frink, A.G. Salinger, M.P. Sears, J.D. Weinhold, and A.L. Frischknecht. "Numerical challenges in the application of density functional theory to biology and nanotechnology." *J. Phys. Cond. Matter*, vol. 14, 12167-12187 (2002).
22. B. Hendrickson and R. Leland. *A multilevel algorithm for partitioning graphs*. Proc. SC'95, ACM, 1995.
23. B. Hendrickson and R. Leland. "The Chaco User's Guide, Version 2.0." Sandia National Laboratories Tech. Rep. SAND95-2344, Albuquerque, NM, 1995.
24. M. Heroux et al. *Trilinos*. <http://software.sandia.gov/trilinos/>.
25. M. Heroux et al. *Epetra*. <http://software.sandia.gov/trilinos/packages/epetra/index.html>.
26. M.A. Heroux, et. al. "An Overview of the Trilinos Project." *ACM Trans. Math. Software*, submitted, 2003.
27. S. Hutchinson, et al. Xyce home page. <http://www.cs.sandia.gov/xyce>.
28. M.A. Iqbal. "Approximate algorithms for partitioning and assignment problems." *Int. J. Par. Prog.*, **20**, 1991.
29. G. Karypis. *hMETIS*. <http://www-users.cs.umn.edu/~karypis/metis/hmetis/index.html>.
30. G. Karypis. hMETIS published results. <http://www.visicad.cs.ucla.edu/~cheese/errata.html>.
31. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. "Multilevel Hypergraph Partitioning: Applications in VLSI Domain." *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 526-529.
32. G. Karypis, K. Schloegel, and V. Kumar. *ParMETIS home page*. <http://www-users.cs.umn.edu/~karypis/metis/parmetis/index.html>.
33. G. Karypis and V. Kumar. *A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm*. 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.
34. G. Karypis and V. Kumar. "Multilevel algorithms for multiconstraint graph partitioning", Tech. report 98-019, Computer Science Dept., Univ. of Minnesota.
35. G. Karypis and V. Kumar. METIS home page: <http://www-users.cs.umn.edu/~karypis/metis/>.
36. B. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs." *Bell System Technical Journal*, 29 (1970), pp291-307.
37. D. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching*. Addison Wesley, Reading, MA, 1998.
38. C.Y. Lo, J. Matousek, and W. Steiger. "Algorithms for Ham-Sandwich Cuts." *Disc. Comput. Geometry* (Jun 1994), v. 11, no. 4, pp. 433-452.
39. K.R. Long. Sundance Home Page: <http://csmr.ca.sandia.gov/~krlong/sundance.html>.
40. B. Olstad and F. Manne. "Efficient partitioning of sequences." *IEEE Trans. Comp.* (1995), v. 44, pp. 1322-1326.
41. M. Ozdal and C. Aykanat. "Hypergraph Models and Algorithms for Data-Pattern Based Clustering." to appear in *Data Mining and Knowledge Discovery*.
42. F. Pellegrini. "SCOTCH 3.4 User's guide." Research report RR-1264-01, LaBRI, November 2001. <http://www.labri.fr/Perso/~pelegrin/scotch/>

43. A. Pinar. Combinatorial Algorithms in Scientific Computing. Ph.D. Dissertation, UIUC, 2001.
44. A. Pinar and B. Hendrickson. "Partitioning for Complex Objectives." Proc. Irregular'01. San Francisco, CA, April 2001.
45. A. Pinar and B. Hendrickson. "Communication Support for Adaptive Computation." *Proc. SIAM Conf. on Parallel Processing for Scientific Computation*, 2001.
46. R. Preis. "Linear Time $1/2$ -Approximation Algorithm for Maximum Weighted Matching in General Graphs." *Symposium on Theoretical Aspects of Computer Science*, STACS 99, Meinel, Tison (eds), Springer, LNCS 1563, 1999, 259-269.
47. R. Preis and R. Diekmann. "The PARTY Partitioning-Library, User Guide - Version 1.1." Technical Report tr-rsfb-96-024, University of Paderborn, Sep. 1996.
<http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html>
48. R. Preis, B. Hendrickson, and A. Pothen. "Fast approximation algorithms for the weighted k-set packing problem." In progress.
49. K. Schloegel, G. Karypis and V. Kumar. "Parallel static and dynamic multiconstraint graph partitioning," *Concurrency and Computation - Practice & Experience*, 2002, v. 14, no. 3, pp. 219-240 (EuroPar 2000 Conference)
50. A. Salinger, K. Devine, G. Hennigan, H. Moffat, S. Hutchinson, and J. Shadid. "MPSalsa: A Finite Element Computer Program for Reacting Flow Problems; Part 2—User's Guide." Sandia National Laboratories Tech. Rep. SAND96-2331, Albuquerque, NM, 1996.
51. J.N. Shadid, et al. MPSalsa home page. <http://www.cs.sandia.gov/CRF/MPSalsa>.
52. A. H. Stone and J.W. Tukey. "Generalized Sandwich Theorems." *Duke Math. J.* **9**, 356-359, 1942.
53. R.S. Tuminaro, M.A. Heroux, S.A. Hutchinson, and J.N. Shadid. *Official Aztec User's Guide, Version 2.1*. Sandia National Laboratories, Albuquerque, NM, 1999.
54. C. Walshaw, M. Cross, and M. Everett. "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes." *J. Par. Dist. Comput.*, 47(2):102-108, 1997.
<http://www.gre.ac.uk/~c.walshaw/jostle/>

Distribution list:

External distribution:

Ken Stanley
322 W. College St.
Oberlin OH 44074

Matthias Heinkenschloss
Department of Computational and Applied Mathematics - MS 134
Rice University
6100 S. Main Street
Houston, TX 77005 - 1892

Dan Sorenson
Department of Computational and Applied Mathematics - MS 134
Rice University
6100 S. Main Street
Houston, TX 77005 - 1892

Yousef Saad
Department of Computer Science and Engineering
University of Minnesota,
4-192 EE/CSci Building, 200 Union Street S.E.
Minneapolis, MN 55455

Kris Kampshoff
Department of Computer Science and Engineering
University of Minnesota,
EE/CSci Building, 200 Union Street S.E.
Minneapolis, MN 55455

Eric de Sturler
2312 Digital Computer Laboratory, MC-258
University of Illinois at Urbana-Champaign
1304 West Springfield Avenue
Urbana, IL 61801-2987

Paul Sexton
Box 1560
St. John's University
Collegeville, MN 56321

Tim Davis, Assoc. Prof.
Room E338 CSE Building
P.O. Box 116120
University of Florida-6120
Gainesville, FL 32611-6120

Padma Raghavan
Department of Computer Science and Engineering
308 Pond Laboratory
The Pennsylvania State University
University Park, PA 16802-6106

Xiaoye Li
Lawrence Berkeley Lab
50F-1650
1 Cyclotron Rd
Berkeley, CA 94720

Richard Barrett
Los Alamos National Laboratory
Mail Stop B272
Los Alamos, NM 87545

Victor Eijkhout
Department of Computer Science,
203 Claxton Complex, 1122 Volunteer Boulevard,
University of Tennessee at Knoxville,
Knoxville TN 37996, USA

David Keyes
Appl Phys & Appl Math
Columbia University
200 S. W. Mudd Building
500 W. 120th Street
New York, NY, 10027

Lois Curfman McInnes
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Paul Hovland
Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Craig Douglas
325 McVey Hall - CCS
Lexington, KY 40506-0045

Robert Preis
University of Paderborn
Faculty of Computer Science, Electrical Engineering and Mathematics
Warburger Straße 100
D-33098 Paderborn
Germany

Joseph Flaherty
Dean, School of Science
Rensselaer Polytechnic Institute
Troy, NY 12180

Ali Pinar
NERSC
Lawrence Berkeley National Laboratory
One Cyclotron Road MS 50F
Berkeley, CA 94720

Umit Catalyurek
The Ohio State University
Department of Anatomy and Medical Education
3184 Graves Hall
333 W. 10th Ave.
Columbus, OH 43210

Internal Distribution:

1	MS 0310	R.W. Leland, 9220
1	MS 0316	R. J. Hoekstra, 9233
1	MS 0316	R. P. Pawlowski, 9233
1	MS 0316	S. A. Hutchinson, 9233
1	MS 0316	L.J.D. Frink, 9212
1	MS 0316	S.J. Plimpton, 9212
2	MS 0323	D.L. Chavez (LDRD office), 1011
1	MS 0826	A. B. Williams, 8961
1	MS 0826	J.R. Stewart, 9143
1	MS 0827	H.C. Edwards, 9143
1	MS 0834	R. P. Schunk, 9114
1	MS 0835	K. H. Pierson, 9142
1	MS 0835	M.W. Glass, 9141
1	MS 0847	C. R. Dohrmann, 9124
1	MS 0847	G. M. Reese, 9142
1	MS 1110	D. E. Womble, 9214
10	MS 1110	M. A. Heroux, 9214
1	MS 1110	J.D. Teresco, 9214
1	MS 1110	W. McLendon, 9223
1	MS 1111	A. G. Salinger, 9233
1	MS 1111	J. N. Shadid, 9233
10	MS 1111	K.D. Devine, 9215
5	MS 1111	R.T. Heaphy, 9215
5	MS 1111	E.G. Boman, 9215
1	MS 1111	B.A. Hendrickson, 9215
1	MS 1152	J. D. Kotulski, 1642
1	MS 1166	C. R. Drumm, 15345
1	MS 9217	J. J. Hu, 9214
1	MS 9217	K. R. Long, 8962
1	MS 9217	P. T. Boggs, 8962
1	MS 9217	R. S. Tuminaro, 9214
1	MS 9217	T. Kolda, 8962
1	MS 9217	V. E. Howle, 8962
1	MS 9018	Central Technical Files, 8945-1
2	MS 0899	Technical Library, 9616