The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring

Erik G. Boman [a], Ümit V. Çatalyürek [b], Cédric Chevalier [c] and Karen D. Devine [a,*]

[a] *Department of Scalable Algorithms, Sandia National Laboratories**, Albuquerque, NM, USA*
[b] *Department of Biomedical Informatics and Department of Electrical and Computer Engineering, The Ohio State University, Columbus, OH, USA*
[c] *Commissariat à l'Énergie Atomique et aux Énergies Alternatives, Direction des Applications Militaires, CEA, DAM, DIF, Arpajon, France*

Abstract. Partitioning and load balancing are important problems in scientific computing that can be modeled as combinatorial problems using graphs or hypergraphs. The Zoltan toolkit was developed primarily for partitioning and load balancing to support dynamic parallel applications, but has expanded to support other problems in combinatorial scientific computing, including matrix ordering and graph coloring. Zoltan is based on abstract user interfaces and uses callback functions. To simplify the use and integration of Zoltan with other matrix-based frameworks, such as the ones in Trilinos, we developed Isorropia as a Trilinos package, which supports most of Zoltan's features via a matrix-based interface. In addition to providing an easy-to-use matrix-based interface to Zoltan, Isorropia also serves as a platform for additional matrix algorithms. In this paper, we give an overview of the Zoltan and Isorropia toolkits, their design, capabilities and use. We also show how Zoltan and Isorropia enable large-scale, parallel scientific simulations, and describe current and future development in the next-generation package Zoltan2.

Keywords: Partitioning, load balancing, matrix ordering, fill-reducing ordering, graph coloring, parallel computing, combinatorial scientific computing

1. Introduction

The Zoltan project started in 1998 with the goal of providing a toolkit for partitioning and dynamic load balancing in dynamic applications [23]. Over the years, the scope of Zoltan has expanded to provide a collection of utilities within the area of combinatorial scientific computing. The three main capabilities of Zoltan are now partitioning, ordering, and coloring. Having all these capabilities in a single toolkit gives users a single interface to a wide range of capabilities. For example,

the ShyLU hybrid solver [51] uses all the main capabilities. Partitioning is used to decompose a sparse matrix into smaller submatrices that are loosely coupled. Matrix ordering is used to reorder the submatrices to minimize fill. Coloring is used within a "probing" scheme to form a sparse approximation to the Schur complement (see Section 2.4.2).

A key element of Zoltan's design was the separation of its data structures from the interfaces to support a wide range of applications. Zoltan relies on a small set of callback (query) functions that must be implemented by the user. When Zoltan was incorporated into the Trilinos solver framework [34], the need for a more "Trilinos-friendly" interface became clear. The solution was a new package called Isorropia. Isorropia is primarily a C++ layer on top of Zoltan that provides matrix-based interfaces to Zoltan capabilities for Trilinos' Epetra matrix/vector classes [35]. Isorropia provides the translation from a user's Epetra matrices and vectors to the data model (e.g., graph, hypergraph,

coordinates) needed by Zoltan for partitioning, coloring and ordering. While Zoltan's more general interface supports a broad set of applications, application developers using Epetra find Isorropia's interface to be much simpler in terms of the amount and complexity of application code needed.

The main purpose of this paper is to give an up-to-date description of the Zoltan and Isorropia packages. In Section 2, we give an overview of the capabilities and algorithms offered. In Section 3, we discuss the software design. Finally, in Section 4, we introduce Zoltan2, the planned successor to both Zoltan and Isorropia.

2. Capabilities and algorithms

2.1. Partitioning and load-balancing

Partitioning and dynamic load balancing are key components of scientific computations. Their goal is to divide applications' data and work among processes in a way that minimizes the overall application execution time. The goal is most often achieved when work is distributed evenly to processes (eliminating process idle time) while minimizing data dependence and movement between processes. A *partition* is an assignment of data and work to subsets called *parts* that are mapped to processes. More precisely, given a set of items V, a k-way partition $P = \{V_1, V_2, \ldots, V_k\}$ assigns members $v_i \in V$ with weight w_i to k parts such that $V_p \cap V_q = \emptyset$ for $p \neq q$ and $\bigcup_{p=1}^{k} V_p = V$. A partition is said to be balanced if $W_p \leqslant W_{\text{avg}}(1 + \epsilon)$ for $p = 1, 2, \ldots, k$, where part weight $W_p = \sum_{v_i \in V_p} w_i$, $W_{\text{avg}} = \frac{1}{k} \sum_{v_i \in V} w_i$, and $\epsilon > 0$ is a predetermined maximum tolerable imbalance.

Items $v_i \in V$ may depend on other v_j for, say, computing solution values at v_i. A dependence that extends between two or more parts requires communication or data movement to satisfy the dependence. The "quality" of a partition is typically considered to be inversely proportional to some measure of the costs to satisfy inter-part dependence. Various cost measures (e.g., number of neighboring parts, number of dependences split between parts) can be used, but the most common cost metric is the total volume of communication between parts.

A common partitioning use-case arises in finite element and volume methods used in structural analysis and fluid dynamics simulations. In these methods, mesh entities (elements, cells, mesh nodes) must

be distributed among processes. Each entity v_i can have an associated computational weight w_i representing, say, its number of degrees of freedom or the time to perform computations on the entity. Mesh adjacencies define data dependence; for example, a mesh node's solution values may be determined by integrations across its adjacent elements, or fluxes across an element face may depend on solution values from elements on both sides of the face. Thus, solutions of the partitioning problem above distribute specified mesh entities' computational weight evenly across processes while attempting to minimize the cost of communicating data for relevant mesh adjacencies that extend across two or more processes.

The partition used to distribute a mesh often induces a partition on a related linear system $Ax = b$ representing the physics on the mesh. For example, each mesh node v_i may correspond to a matrix row a_i in A; solution values at v_i are represented by vector entries x_i. Non-zeros a_{ij} in the matrix represent a data dependence between mesh nodes v_i and v_j. Thus, a partition of mesh nodes results in a row-based partition of the linear system. In this case, the communication metric is taken to be the total amount of communication needed to perform a matrix–vector multiplication Ax.

Other parallel applications needing partitioning include (but are certainly not limited to) particle methods (e.g., for hydrodynamics or biological simulations), contact detection (e.g., for crash simulations), device-based electrical circuit simulations, circuit design and layout, semantic analysis for term-document databases, and graph analysis on networks. Characteristics of these applications and appropriate partitioning strategies for them will be discussed in more detail below.

An application's characteristics determine the partitioning strategy best-suited to the application. No single algorithm is best for all applications. Trade-offs between partitioning strategies include geometric-based data locality versus connectivity-based data locality, static versus dynamic redistribution, and speed of partitioning versus quality of the resulting partition. For this reason, we have included a suite of partitioning algorithms in Zoltan, each of which is accessible from Isorropia as well. These algorithms range from fast, medium quality geometric partitioners to more expensive, higher quality graph and hypergraph-based methods.

Geometric partitioning. Geometric methods partition data based on their geometric locality. Items assigned to the same part are physically close to each

other in space. This geometric locality makes geometric methods ideal for particle-based applications and contact detection algorithms. In particle methods, the v_i are particles to be distributed to processes. Dependence arises, not between v_i and other fixed v_j, but rather between v_i and all other particles within a certain distance of v_i. Thus, in addition to maintaining load balance, partitioners must maintain data locality of the particles within processes for efficient force calculations. Similarly, in contact detection, searches for contact between entities are optimized when physically close entities are assigned to the same process; partitioners must account for the geometric locality of entities to maintain efficient local search as the geometry deforms. Geometric methods are also useful for applications requiring frequent partitioning (e.g., adaptive mesh-based simulations), because they are fast and require only coordinate information as input. However, because they do not explicitly model communication, they can produce partitions with only medium partition quality.

Geometric methods in Zoltan include Recursive Coordinate Bisection, Recursive Inertial Bisection and Hilbert Space-Filling Curve partitioning. Recursive Coordinate Bisection (RCB) [2] was originally designed for adaptive mesh refinement methods, because it assigns parent and child elements to the same process, eliminating the need for communication between mesh levels. In RCB, the work of a problem is divided in two equally weighted halves using a cutting plane orthogonal to a coordinate axis. Items are assigned to sub-domains based on their geometric position relative to the cutting plane. The resulting sub-domains are then recursively cut, until the number of sub-domains equals the number of parts requested (see Fig. 1, left). (Although we have described the algorithm for $k = 2^m$ parts for some m, the Zoltan implementation is applicable for all values of k.) RCB has the property that its resulting partitions are "incremental"; that is, small changes in the input data result in only small changes to the partition. This property can be exploited to reduce data movement costs when frequent repartitioning is needed. Recursive Inertial Bisection (RIB) [58,60] is a variant of RCB in which cutting planes are chosen to be orthogonal to the principal axes of the geometry, rather than to the coordinate axes. Hilbert space-filling curve (HSFC) partitioning [49,63] has been used for gravitational simulations, smoothed particle hydrodynamics, and adaptive finite element methods. In HSFC partitioning (Fig. 1, right), each item's position along a Hilbert space-filling curve is computed based on its coordinate values. The resulting positions provide a one-dimensional ordering of the three-dimensional items in the domain, with the property that items that are close to each other in the one-dimensional ordering are also close to each other in three-dimensional space. This one-dimensional ordering is then cut into k equally weighted parts. Like RCB, HSFC partitioning is fast and incremental, and creates parts containing geometrically close items.

With the geometric partitioners in Zoltan, each process can inexpensively store information about the entire partition. By storing the cutting-plane locations and directions used in RCB or RIB, or the cut positions in HSFC partitioning, each process can know the region of physical space assigned to every part; the total storage is O(k) for k parts. Zoltan includes additional functionality for querying this cut information
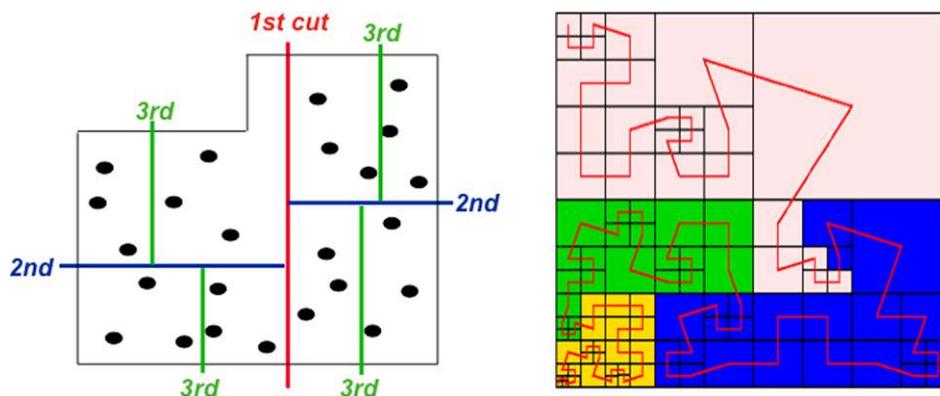

Fig. 1. Examples of Recursive Coordinate Bisection (RCB) (left) and Hilbert space-filling curve (HSFC) partitioning (right) in Zoltan. RCB recursively cuts the domain into equally weighted halves. HSFC partitioning linearly orders elements according to their position along a Hilbert space-filling curve (red), and cuts the curve equally among parts. (The colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0342.)
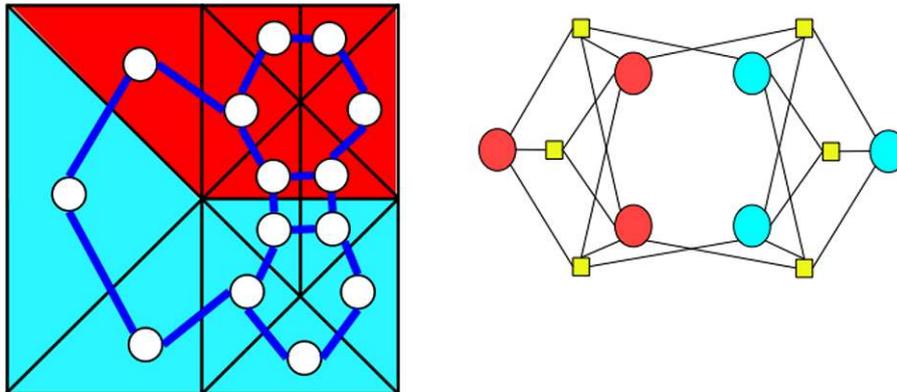
Fig. 2. The graph (left) and hypergraph (right) models for connectivity-based partitioning. In the graph example, graph vertices (white) represent mesh elements, while graph edges (blue) represent face adjacencies; element colors indicate part assignments. In the hypergraph, vertices (circles) are connected by hyperedges (squares); vertex colors indicate part assignments. (The colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0342.)

to determine which part owns a given point in space, or which parts overlap a given bounding box in space. These functions (`Zoltan_Point_PP_Assign` and `Zoltan_Box_PP_Assign`) are useful for global searches in contact detection where a process needs to know which other processes contain potential contacts with a given entity, based on the entity's position in space; local searches for contact can then be done within each of the processes with potential contacts.

Connectivity-based partitioning. Connectivity-based partitioning methods explicitly represent data dependence and use it to attempt to minimize communication and data movement costs. In connectivity-based methods, application data are represented by a weighted graph or hypergraph. Items to be partitioned are vertices. Pairwise or multiway data dependence is represented by graph or hypergraph edges, respectively (see Fig. 2). Any edge that has vertices in two or more parts requires communication to satisfy the data dependence it represents. Thus, partitioning vertices into equally weighted parts while minimizing the weight of edges that are split among one or more parts provides a balanced work distribution with approximately minimal total communication volume.

Graph partitioning [31] is perhaps the most well-known partitioning method. It has been applied extensively in finite element/volume applications and linear algebra frameworks. In linear systems, for example, the structure of matrix A can be interpreted as the adjacency matrix of a graph, with non-zero entry a_{ij} representing an edge from v_i to v_j. Since graph partitioners use an undirected model, the linear systems that can be represented are limited to square, structurally symmetric matrices. For structurally non-symmetric

matrices, symmetrization via $A + A^{\mathrm{T}}$ or AA^{T} allows graph partitioners to be applied, but reduces the accuracy of the communication model. Zoltan includes interfaces to two popular graph partitioning libraries: PT-Scotch [48] and ParMETIS [42]. It also has native graph-partitioning capabilities through its hypergraph partitioner.

Hypergraph partitioning [14] improves the model used in graph partitioning. In row-partitioning of a linear system, for example, matrix rows are hypergraph vertices, and matrix columns are hypergraph edges (with non-zero column entries specifying which vertices are in the edges) [8]. There is no restriction that the matrix be square or structurally symmetric. Thus, hypergraphs can represent a much broader class of problems (e.g., term-document matrices, circuit networks). While graph models must approximate multiway data dependence by several pairwise edges, hypergraph models accurately represent multiway dependence within a single hyperedge. Thus, hypergraph models more accurately represent communication volume for edges split by part boundaries, allowing hypergraph partitioners to generate higher quality partitions. The main drawback of hypergraph partitions is that they often require more time to be generated than graph partitions. Zoltan has a native parallel hypergraph partitioner [24], as well as an interface to the serial hypergraph partitioner PaToH [9].

Optimally balancing vertex weight while minimizing the weight of split edges is an NP-hard optimization problem [26], but fast multilevel algorithms and software produce good solutions in practice [7,9,24,32, 42]. In multilevel algorithms, an input (hyper)graph is repeatedly coarsened, with coarse vertices represent-

ing clusters of input vertices. The coarse (hyper)graph can be partitioned easily. The coarse partition is then projected back to the finer (hyper)graphs, with local improvements of the partition performed at each finer level. This approach produces very high quality partitions for many applications, but is more expensive computationally than geometric methods.

Static versus dynamic partitioning. The amount of change in an applications' computation over time determines whether static or dynamic partitioning should be used. In applications such as traditional finite element methods, the mesh and the work associated with each mesh entity do not change during the course of a computation. Thus, a good *static* partition can be computed once at the beginning of a simulation and will remain effective throughout a simulation. Because static partitioning is done only once, its cost can be amortized over the entire computation and, thus, more expensive partitioners (e.g., graph and hypergraph partitioning) can be used to obtain higher quality partitions. Moreover, because data is moved only once to the new partition, the partitioner does not have to minimize the cost of moving data from its old part assignment to its new one; that is, it does not have to account for an existing partition in computing the new one. All of Zoltan's geometric, graph-based and hypergraph-based methods can be used for static partitioning.

Applications whose computational loads or data locality vary during the course of a computation require *dynamic* partitioning or load balancing. In *h*-adaptive finite element methods, for example, the mesh is locally refined or coarsened to satisfy accuracy constraints, thus dynamically changing the amount of work per process and creating load imbalance. Geometric locality and load balance in particle methods and crash simulations are often lost as particles move and structures deform, respectively. Dynamic partitioning methods restore load balance and/or geometric locality by redistributing data and work among processes. But since they are working with data that is already distributed, dynamic partitioning methods must attempt to also minimize the cost to move data from the existing partition to the new one. Some dynamic methods (e.g., RCB, HSFC partitioning) achieve this goal implicitly; small changes in the data naturally result in only small changes to the partition. Other dynamic methods (e.g., Zoltan's hypergraph repartitioner, graph-based adaptive-refinement methods) explicitly account for an existing distribution in computing the new distribution. Diffusive algorithms, for example [56], in which work is shifted from heavily

loaded processors to more lightly loaded neighboring processors, provide one way to obtain incremental partitions in connectivity-based models, but their quality can degrade over several invocations of the partitioning algorithm; this approach is available in Zoltan through its interfaces to ParMETIS' adaptive-refinement graph algorithms [42]. The approach used in Zoltan's hypergraph and native graph algorithms [12] connects each vertex to its current process by an edge weighted by the cost of moving the vertex's data to a new part; Zoltan's partitioners then account for both data movement costs and data dependence costs in the new partition.

Additional partitioning functionality. Zoltan is the container for all partitioning algorithms in Trilinos. It includes the algorithms described above, as well as random and block partitioning. Random partitioning simply randomly assigns items to parts. Block partitioning assigns $|V|$ items to parts in a linear fashion, with the first $|V|/k$ items going to the first part, the next $|V|/k$ items going to the second part and so on.

Hierarchical partitioning strategies are also available in Zoltan. These methods can be used to exploit user-provided information about the underlying machine architecture in heterogeneous parallel computers. For example, in a parallel machine with 10 nodes, where each node is a 16-core processor, users can choose to partition first into 10 parts, and then further partition each of those parts into 16 parts. Different partitioning algorithms can be applied at each level to attempt to minimize costs related to data movement at the level. This hierarchical partitioning has shown benefit in some shared-memory clusters [61]; further experimentation for multicore systems is underway.

Isorropia provides a Trilinos Epetra matrix-partitioning interface to Zoltan. It maps matrix data structures (e.g., rows, non-zeros) to the partitioning models (e.g., vertices, edges) in Zoltan, provides access to all Zoltan partitioners, and provides data redistribution capabilities for Epetra matrices and vectors. The default Epetra partition is row-based; that is, entire rows of a matrix (along with corresponding vector entries) are assigned to parts. However, Isorropia provides more sophisticated partitions as well. For example, in two-dimensional matrix partitions, individual non-zeros of the matrix are assigned to parts, allowing greater flexibility to reduce total communication volume or number of neighboring processes in matrix–vector multiplication [11]. Two-dimensional matrix partitioning methods available in Isorropia include fine-grained hypergraph partitioning [11], Cartesian methods [11], and RCB partitioning of the (i, j) indices of the matrix non-zeros [22].

2.2. Ordering

The ordering of sparse data structures (graphs, matrices, meshes) is important to obtain good performance on modern architectures. In Zoltan/Isorropia, we have focused on two different goals: ordering irregular data for locality, and fill-reducing ordering for sparse matrix factorizations. The same interfaces and classes are used for both types of ordering.

2.2.1. Locality-enhancing orderings

A very common kernel in scientific computing is to loop over an irregular (unstructured) data structure such as a mesh or a sparse matrix. For example, in sparse matrix–vector multiplication $y = Ax$, each nonzero entry in A has to be traversed. Mathematically, the visit order does not matter because the result is invariant. However, the performance on a modern computer is highly dependent on the ordering since memory access is relatively slow compared to computation. The goal is thus to reorder the data such as to maximize locality and reduce the number of cache misses. This is also known as *loop permutation*. Typically, the ordering (permutation) is computed once but can be used many times.

In Zoltan/Isorropia, we provide two such orderings. One is based on space-filling curves, where items are ordered in local memory according to their position along a space-filling curve through the items (Fig. 1). The other is the Reverse Cuthill–McKee (RCM) algorithm [20], which attempts to minimize the bandwidth of a sparse matrix. RCM is also often used to reorder matrices for iterative methods [25].

The typical use case is to use locality-enhancing ordering on local data within a single compute node or core. Thus, only a serial implementation is provided in Zoltan/Isorropia.

2.2.2. Fill-reducing ordering of sparse matrices

In many scientific computing applications, it is necessary to solve linear systems $Ax = b$. Either direct or iterative methods can be used. When A is ill-conditioned or no good preconditioner is known, direct solvers are preferred since iterative solvers will converge slowly or not at all. The work and memory required to solve $Ax = b$ depends on the fill in the factorization. A standard technique is therefore to permute (reorder) the matrix A such that one solves the equivalent system $(PAQ)(Q^{\mathrm{T}}x) = Pb$, where P and Q are permutation matrices. This reordering is usually performed as a preprocessing step, and depends only on the non-zero structure, not on the numerical values. In the symmetric positive definite case, a symmetric permutation is used to preserve symmetry and definiteness, i.e., $Q = P^{\mathrm{T}}$. In the non-symmetric case, numerical pivoting is required for stability during the factorization phase. Generally, only a column permutation is therefore performed in the setup phase to reduce fill.

There are two broad categories of fill-reducing ordering methods: *minimum degree* [62] and *nested dissection* [29] methods. The minimum-degree class takes a local perspective and the algorithms are greedy. The algorithms are inherently sequential but fast to compute. The nested dissection methods, on the other hand, take a global perspective. Typically, recursive bisection is used, which is suited for parallel processing. Nested dissection methods usually give less fill for large problems, and are better suited for parallel factorization. In practice, hybrid methods that combine nested dissection with a local ordering are most effective.

Symmetric positive definite case. Most of the state-of-the-art symmetric ordering tools [10,16,33,40] are hybrid methods that combine multilevel (hyper)graph partitioning for computing nested dissections with a variant of minimum degree for local ordering. The main idea of nested dissection is as follows. Consider a partitioning of vertices (V) into three sets: V_1, V_2 and V_S, such that the removal of V_S, called a *separator*, decouples two parts V_1 and V_2. If we order the vertices of V_S after the vertices of V_1 and V_2, i.e., if the elimination process starts with vertices of either V_1 or V_2, no fill will occur between the vertices of V_1 and V_2. Furthermore, the elimination process in V_1 and V_2 are independent from each other and their elimination only incurs fill to themselves and V_S. In the nested dissection, the ordering of the vertices of V_1 and V_2 is simply computed by applying the same process recursively. In the hybrid methods, recursion is stopped when a part becomes smaller than predetermined size, and minimum degree-based local ordering is used to order the vertices in that part.

Non-symmetric case. In non-symmetric direct solvers, partial pivoting is required for numerical stability and to avoid breakdown. Partial pivoting is usually performed by looking down a column for a large entry and then swapping rows. This defines a row permutation but it is determined during the numerical factorization. Therefore, fill-reducing ordering has focused on column ordering. Consider $\hat{A} = PAQ$. The column permutation Q can be computed in a preprocessing step based on the structure of A. Note that P is not known at this point, so Q should reduce fill for any P.

There are two popular approaches to column ordering. First, one can simply symmetrize A and compute

a nested dissection ordering for either $A + A^\mathrm{T}$ or $A^\mathrm{T}A$. This is attractive because one can use the same algorithms and software as in the symmetric case. However, ordering based on $A + A^\mathrm{T}$ does not necessarily give good quality ordering for A. Ordering based on $A^\mathrm{T}A$ generally works better and has a theoretical foundation since the fill for LU factors of A is contained within the Cholesky factor of $A^\mathrm{T}A$. Unfortunately, $A^\mathrm{T}A$ is often much denser than A and takes too much memory to form explicitly. The second approach is therefore to order the columns of A to reduce fill in the Cholesky factor of $A^\mathrm{T}A$ but without forming $A^\mathrm{T}A$. Several versions of minimum degree have been adapted in this way; the most popular is CO-LAMD [21]. A limitation of COLAMD is that it is inherently sequential and not suited for parallel computations.

A third approach, the HUND method, was recently proposed in [30]. The idea is to use hypergraphs to perform a non-symmetric version of nested dissection, then switch to CCOLAMD on small subproblems. Hypergraph partitioning is used to obtain a singly bordered block diagonal (SBBD) form [1], see Fig. 3. It is shown in [30] that the fill in LU factorization is contained within the blocks in the SBBD form, even with partial pivoting. Experiments with serial LU factorization show that HUND compares favorably to other methods both in terms of fill and flops. Perhaps the greatest advantage of HUND, though, is that it can both be computed in parallel and produces orderings well suited for parallel factorization.

The fill-reducing orderings are intended for solving large systems, and thus must be computed in parallel. The focus in Zoltan is on nested-dissection methods, since these generally work best on large problems.
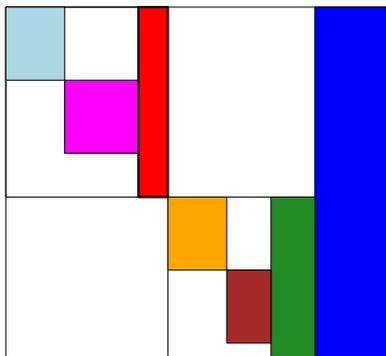

Fig. 3. Sparse matrix reordered by the HUND algorithm. All fill in LU factorization is limited to the colored blocks. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0342.)

Classic nested dissection for symmetric problems is provided through interfaces to the third-party libraries Scotch [16] and ParMETIS [41]. For non-symmetric problems, Zoltan contains a native parallel implementation of HUND. HUND can also be applied to symmetric systems, but works best in the non-symmetric case.

Isorropia provides the same capabilities as Zoltan, and in fact calls Zoltan. Zoltan and Isorropia do not provide a sparse direct solver; rather, they should be used to compute a permutation vector before calling a direct solver. Isorropia provides the most convenient interfaces for Trilinos solvers such as Amesos (Amesos2) [54], while Zoltan may be simpler to use for other solvers.

2.3. Coloring

Graph coloring is an archetypal model in many scientific applications. Its applications vary from identifying concurrent computations, such as in iterative solution of sparse linear systems [39], preconditioners [52], sparse tiling [59], and eigenvalue computation [45], to efficient computation of Jacobian and Hessian matrices [27], as well as channel assignment problems in radio networks [43,44]. A distance-k coloring of a graph is an assignment of colors to vertices such that any two vertices connected by a path consisting of at most k edges receive different colors, and the goal of the associated problem is to minimize the number of colors used.

The distance-k coloring problem is formally defined as follows. Let $G = (V, E)$ be a graph with $|V|$ vertices and $|E|$ edges. We call two distinct vertices in the graph distance-k neighbors if a shortest path connecting them consists of at most k edges. The set of distance-k neighbors of a vertex v is denoted by $adj_k(v)$; its cardinality, also called the *degree-k* of v, is denoted by $\delta_k(v)$. The degree of the vertex having the most neighbors is $\Delta_k = \max_v \delta_k(v)$. A distance-$k$ coloring $\phi : V \to \mathbb{N}$ is a function that maps each vertex of the graph to a color (represented by an integer), such that two distance-k neighbor vertices have different colors, i.e., $\forall u \in adj_k(v), \phi(u) \neq \phi(v)$. Without loss of generality, the number of colors used is $\max_{u \in V} \phi(u)$. The optimization problem at hand is to find a coloring with as few colors as possible. In distance-1 coloring, the minimal number of colors a graph can be colored with is called the chromatic number of the graph $\chi(G)$. The problem of finding $\chi(G)$ for an arbitrary graph G is known to be an NP-Complete
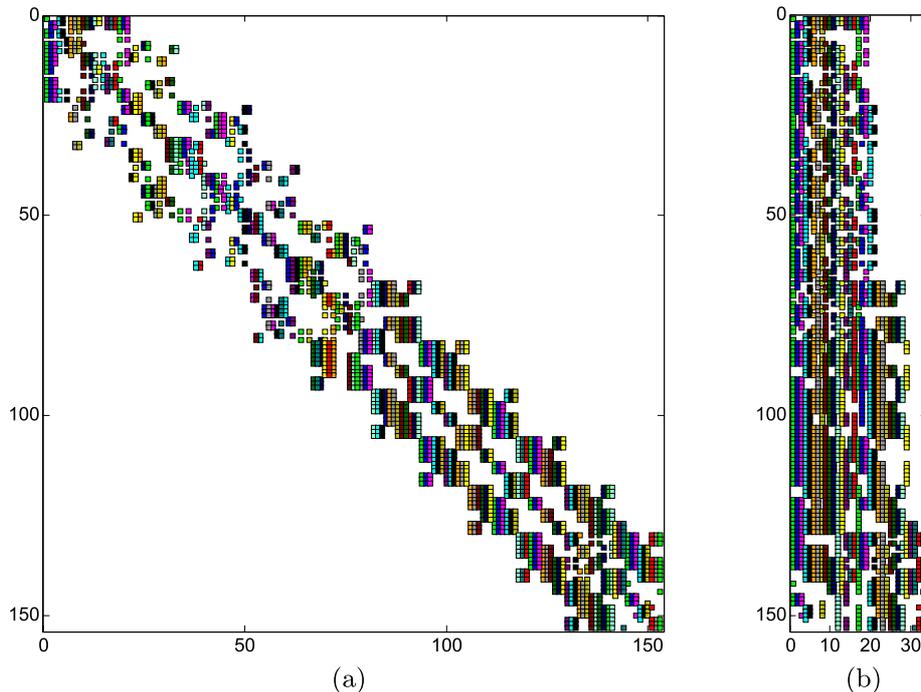
Fig. 4. A sample Jacobian matrix A and its compressed representation B. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0342.)

problem [46]; thus, finding optimal graph colorings for general graphs is NP-Hard. Fortunately, simple heuristics often work well in practice.

Partial distance-2 coloring is another variant of coloring that is used to identify a *structurally orthogonal* partition of a Jacobian matrix A [19]. Here structurally orthogonal means a partition of the columns of A in which no two columns in a group share a non-zero at the same row index. This approach yields a seed matrix S where the entries of A can be directly recovered from the compressed representation $B \equiv AS$ (see Fig. 4). In this problem, the structure of a Jacobian matrix A can be represented by the *bipartite* graph $G_b(A) = (V_1, V_2, E)$, where V_1 is the row vertex set, V_2 is the column vertex set, and $(r_i, c_j) \in E$ whenever the matrix entry a_{ij} is non-zero. A partitioning of the columns of the matrix A into groups consisting of structurally orthogonal columns is equivalent to a *partial distance-2 coloring* of the bipartite graph $G_b(A)$ on the vertex set V_2 [27]. The coloring, also known as *bipartite coloring*, is called "partial" distance-2 coloring because the row vertex set V_1 is left uncolored.

In large scientific parallel applications, the computational model (hence the graph) is already distributed onto the nodes of the parallel machine; therefore, coloring needs to be performed in parallel. An alternative

approach, if the graph is sufficiently small, is to aggregate it on a single node and color it there sequentially. A better approach would be taking advantage of the partitioning of the graph, and color the *interior* vertices (vertices for which all their distance-k neighbors are local) in parallel and then color the remaining (vertices that have at least one non-local distance-j neighbor) sequentially, by aggregating the graph induced by them on a single processor. However, as shown in [4], one can perform significantly better by using a distributed memory coloring algorithm.

The distributed memory coloring frameworks for distance-1 [4] and distance-2 [3] coloring have been implemented in Zoltan. The framework provides efficient implementation for *greedy graph coloring* algorithms [17]. In greedy coloring, the vertices of the graph are visited in some *order* and the smallest permissible color at each iteration is assigned to the vertex. Choosing the smallest permissible color is known as the *First Fit* strategy. This simple algorithm has two nice properties. First, for any vertex visit ordering, it produces a coloring with at most $1 + \Delta_k$ colors. Second, for some vertex-visit orderings it will produce an optimal coloring [18]. Many heuristics for ordering the vertices have been proposed in the literature [18,27]. Commonly known vertex orderings are *Largest First*,

Smallest Last, *Saturation Degree*, and *Incidence Degree* orderings. We refer the reader to [27] for a succinct summary of these ordering techniques, and to `ColPack` [28] for a sequential implementation of various coloring, ordering, matrix recovery and other supporting algorithms for Jacobian and Hessian computations.

Zoltan provides a parallel coloring framework. In this framework, a coloring is constructed in multiple rounds. In each round, all the uncolored vertices are tentatively colored using the greedy coloring algorithm. This process may yield conflicts that can be detected independently by each processor after coloring. When a conflict occurs, one of the vertices that are in conflict is marked for recoloring in the next round. The vertex to be colored is chosen based on a random total ordering generated beforehand to avoid load imbalance that may arise due to natural vertex numbering. The algorithm iterates until there are no more conflicts.

In order to reduce the number of conflicts at each round, the coloring of the vertices is performed in *supersteps*. In each superstep, each processor colors a given number of its own vertices. Then it exchanges the colors of the boundary vertices colored in that superstep, if any, with its *neighbor* processors. If carried out synchronously, this communication mechanism ensures that two vertices can be in a conflict only if they are colored in the same superstep. The size of the superstep becomes an important parameter since a smaller size increases the number of messages exchanged on the network while a larger size is likely to increase the number of conflicts.

Coloring itself can be used to order vertices for *recoloring*. Culberson [18] showed that in recoloring, if the vertices belonging to the same color class (i.e., the vertices of the same color) in a previous coloring are colored consecutively, the number of colors will either decrease or stay the same. Zoltan currently provides three different heuristics to obtain new permutations of color classes for recoloring of distance-1 coloring: Reverse order of colors [18], non-increasing number of vertices, where the color classes are ordered in the non-increasing order of their vertex counts, and non-decreasing number of vertices, where the color classes are ordered in the non-decreasing order of their vertex counts.

There are five main parameters affecting the behavior of the framework in Zoltan and all are tunable by the user. These are superstep size, synchronous or asynchronous execution of supersteps, coloring order of the vertices, color selection strategy and recol-

oring (multiple passes of coloring). Several combinations of those were experimentally investigated to determine the best parameters for reducing the number of colors and the runtime. The studies [3,4,13,55] show that there is no single combination that outperforms the others both in terms of the number of colors and the runtime. In general, the best runtime is achieved by a single pass of greedy coloring, coloring internal vertices first and using asynchronous communication, whereas the best number of colors is achieved by multiple passes of recoloring using Smallest Last vertex visit ordering and synchronous communication.

2.4. Utilities

While the focus of Zoltan and Isorropia is on combinatorial algorithms, a few utilities related to and included in Zoltan and Isorropia functionality have proven useful. Two examples are described below.

2.4.1. Distributed data directories

Dynamic applications often need to locate off-processor information. After repartitioning, for example, a processor may need to rebuild ghost cells and lists of items to be communicated; it may know which items it needs, but may not know where they are located. After global ordering, data values such as the permuted global numbering associated with items may need to be shared among processors.

To allow applications to locate off-processor data, Zoltan includes distributed data-directory utilities based on a rendezvous algorithm [50]. Zoltan's distributed data directories have been used for updating ghost information and building communication maps in finite element and particle simulations.

Each data directory is distributed evenly across processes in a predictable fashion (through either a linear decomposition of the IDs or a hashing of IDs to processes). Within a process, directory entries are stored in hash tables to allow constant-time searches for specific entries. Processes register their owned items' IDs along with their process number and desired user data by calling `Zoltan_DD_Update`. Other processes can obtain the process number and user data associated with a given item by calling `Zoltan_DD_Find`, which, using the same distribution scheme, requests the information from the process holding the directory entry. In this way, distributed data directories provide memory efficient, constant time look-ups of off-process data. Total memory usage across all processes is proportional to the number of items. Distributing

the directory across processes avoids performance bottlenecks and memory limitations that would arise if the directory were stored on only one process. Using predictable, hash-based distribution and local-search schemes allows constant-time access to the data.

2.4.2. Probing

A special feature in Isorropia (not included in Zoltan) is the ability to perform *probing* on an operator. Often a linear operator is given only implicitly, not explicitly as a matrix. In some situations, it is necessary to build an explicit matrix that represents or approximates the operator. The probing technique was first suggested by Chan and Matthews [15] to approximate Schur complements in PDE solvers, but was restricted to banded approximations. An extension to general graphs was suggested in [57], where they use graph coloring as a tool. The probing method in Isorropia is based on this approach.

The input to an `Isoroppia::Prober` is an `Epetra_Operator` and the sparsity pattern given by a `Epetra_CrsGraph`, while the output is an `Epetra_CrsMatrix` that represents (approximates) the operator by using the given sparsity pattern. The use of probing is best explained by an example. Suppose a user has a matrix-free numerical method where the operator is given as a "black-box" but she wants to use an algebraic preconditioner (e.g., Ifpack or ML). Ifpack/ML needs a concrete sparse matrix, not an operator. A simple but inefficient way to solve this problem is to apply the desired operator to all the identity vectors, e_i, $i = 1, \ldots, n$, where n is the dimension of the operator. This would require n applications of the operator, which is usually too expensive to be practical. However, if the user knows the sparsity pattern (or can estimate it), then he can use the prober to construct the desired matrix much more efficiently.

Internally, the prober applies the operator to a carefully chosen set of vectors. To minimize the cost of probing, graph coloring is used to generate a minimal set of probing vectors. In fact, the probing vectors correspond to the seed matrix S as described in Section 2.3. Hence the main cost of the probing is to apply the operator c times, where c is the number of colors. (This is analogous to Jacobian compression.) Typically, $c \ll n$ so there is a substantial reduction of work. The coloring and reconstruction happen transparently to the user, and the coloring capabilities in Zoltan/Isorropia are leveraged.

3. Software design

The Trilinos packages Zoltan and Isorropia provide combinatorial algorithms to a wide range of applications. Zoltan is the base package, containing the partititioning, ordering and coloring algorithms described above. It requires users to map their data into the particular combinatorial model (e.g., graph, hypergraph, coordinates) to be used in partitioning, ordering and coloring. For users of Trilinos' Epetra [35], the most widely used matrix/vector classes in Trilinos, the Isorropia package performs this mapping from matrices to the combinatorial models and passes these models to Zoltan. Thus, the Isorropia interface is much simpler for Epetra users, and can provide richer algorithms (e.g., two-dimensional matrix partitioning) for matrix-based applications.

Both Zoltan and Isorropia assume an owner-computes model, where each process "owns" a distinct set of items and may optionally have copies of other needed items. Both can operate in serial; for parallel operation, they rely on MPI [47] to perform interprocess message-passing communication. Zoltan is written in C and provides C, C++ and Fortran90 interfaces; Isorropia's Epetra-based interface is written in C++, with C++ and Python interfaces (through Trilinos' PyTrilinos package [53]). Details of Zoltan and Isorropia's software design are below.

3.1. Zoltan

Zoltan is designed to provide parallel partitioning, coloring, and ordering to a wide range of parallel applications. Zoltan uses a distributed-memory programming model with the Message-Passing Interface (MPI) library [47] performing interprocess communication. Although it is a Trilinos package, it does not depend on any other Trilinos package or data structure, and, indeed, it can be built and used separately from Trilinos. Its callback-function user interface is "data-structure neutral" in that Zoltan users do not have to use a specific data structure in their applications, nor do they have to build a specific data structure for Zoltan. This interface has served well to support a variety of applications, including finite element methods with adaptive mesh refinement for multiphysics simulations, particle methods for accelerator and biological simulations, crash simulations with contact detection, circuit simulations, multigrid preconditioning, simulations of emerging computer architectures, and peridynamics simulations. Zoltan's design and implementation en-

Table 1
Examples of large-scale partitioning performed by Zoltan

Partitioning algorithm	Application	Problem size	Number of processes	Number of parts	Architecture	Source
Graph	PHASTA computational fluid dynamics	34M elements	16K	16K	BG/P	M. Zhou et al., RPI
Hypergraph	PHASTA computational fluid dynamics	1B elements	4096	160K	Cray XT/5	M. Zhou et al., RPI
Hypergraph	SPARTA load balancing toolkit	800M zones	8192	262K	Hera AMD Quadcore	K. Lewis, LLNL
Geometric (RCB)	Pic3P particle in cell	5B particles	24K	24K	Cray XT/4	A. Candel, SLAC
Geometric (RCB)	MPSalsa computational fluid dynamics	208M mesh nodes	12K	12K	Redstorm	P. Lin, SNL
Geometric (RCB)	Trilinos/ML multigrid in shock physics	24.6M rows; 1.2B non-zeros	24K	24K	Redstorm	J. Hu et al., SNL

ables it to be used for large scale simulations, partitioning for more than 200K processes; see Table 1 for some examples. Zoltan's design also supports research and development of new algorithms, such as its recently developed hypergraph and hierarchical partitioners.

In the set of applications that use Zoltan, data to be operated on include (but are not limited to) mesh elements and nodes, particles, matrix rows/columns/non-zeros, circuits, and agents. Rather than limit Zoltan's capabilities to a specific type of data, we consider each data entity to be merely an item on which Zoltan operates. Each item must have a globally unique identifier (ID) that can be represented as an array of unsigned integers. Examples of single-integer global identifiers include global element numbers and global matrix row numbers. Applications that do not support global numbering can use, e.g., a two-integer tuple consisting of the process rank of the process that owns the entity and a local entity counter in the process. Zoltan uses these global identifiers only to identify items; any unique naming scheme is acceptable. Zoltan also gives users the option of providing a local identifier (also an array of unsigned integers) for each item. These local identifiers allow applications to access data more directly in callback functions by avoiding a mapping from global identifiers to the local location of items. Examples of useful local identifiers include items' indices in data arrays and pointers to items' locations in memory.

Zoltan uses a callback-function interface, through which applications describe their data to Zoltan. Callback functions are small functions written by the user that access the user's data structures and return needed data to Zoltan. When applications invoke Zoltan's methods, Zoltan calls these user-provided callback functions to build the data structures (coordinates, graphs, hypergraphs, etc.) it needs for partitioning, coloring and ordering. Figure 5 contains an example of the geometric callback functions for a simple particle-based simulation. Callback function `num_geom` returns the dimension of the problem domain, in this case, three for a three-dimensional problem. Callback function `geom_multi` returns the coordinates for each on-process particle in arrays allocated by Zoltan and provided to the application. Pointers to these functions are registered with Zoltan through calls to `Zoltan_Set_Fn`. A complete example program using these functions is included in the Appendix.

The set of callback functions used in Zoltan has been kept quite small (see Table 2) and the same interface is used for partitioning, ordering, and coloring. Moreover, users need to implement only those callbacks required by the specific Zoltan capability they wish to use. At a minimum, users must provide callback functions that return the number of items owned by a process, and the unique global and optional local IDs for those items. Other callback functions needed depend on the operations to be performed by Zoltan. Geomet-

```
1   ///////////////  Application data for particle simulation ////////////
2   struct UserData {
3     int numMyParticles;
4     struct Particle {
5       int id;            // unique global name for particle
6       double x, y, z;    // geometric coordinates for particle
7       // ... solution values, etc. ...
8     } *myParticles;
9   };
10
11  ///////////  User-provided query functions for coordinates  ///////////
12  // Return dimension of problem.
13  int num_geom(void *data, int *ierr)
14  {
15    return 3;        // 3D problem
16  }
17
18  // Return coordinates for objects requested by Zoltan in globalIDs array.
19  void geom_multi(void *data, int nge, int nle, int numObj,
20                          ZOLTAN_ID_PTR globalIDs, ZOLTAN_ID_PTR localIDs,
21                          int dim, double *geomVec, int *ierr)
22  {
23    // Cast data pointer provided in Zoltan_Set_Fn to application data type.
24    struct UserData *myData = (struct UserData *) data;
25
26    // Copy coordinates for particle globalID[i] (with localID[i])
27    // into geomVec.
28    int i, j = 0;
29    for (i = 0; i < numObj; i++) {
30      geomVec[j++] = myData->myParticles[localIDs[i]].x;
31      if (dim > 1) geomVec[j++] = myData->myParticles[localIDs[i]].y;
32      if (dim > 2) geomVec[j++] = myData->myParticles[localIDs[i]].z;
33    }
34    *ierr = ZOLTAN_OK;
35  }
```

Fig. 5. Example of the geometric callback functions `Zoltan_Num_Geom_Fn` and `Zoltan_Geom_Multi_Fn` for simple particle-based application. A full program using these callbacks is included in the Appendix.

ric partitioning and ordering methods, for example, require two callback functions as in Fig. 5 to return the geometric dimension of the data and each item's geometric coordinates. Graph-based partitioning, coloring and ordering algorithms require graph-based callbacks that return information about edge connectivity between items. Hypergraph-based methods can use either graph-based callbacks (from which a hypergraph structure is constructed) or hypergraph-based callbacks that enable the full expressiveness of the hypergraph model to be used for, say, structurally non-symmetric problems.

Figure 6 shows the basic use of Zoltan in an application needing dynamic load balancing. The application begins as usual, reading input files and creating its data structures. Then it calls several Zoltan set-up functions. It initializes Zoltan by calling `Zoltan_Initialize`, which checks that MPI is initialized.

It also calls `Zoltan_Create` to allocate memory for Zoltan; a pointer to this memory is returned by `Zoltan_Create` and must be passed to all other Zoltan functions. Next, by calling `Zoltan_Set_Param`, the application selects the partitioning method it wishes to use and sets method-specific parameters. It registers pointers to callback functions through calls to `Zoltan_Set_Fn`. After the set-up is completed, the application computes a new partition by calling `Zoltan_LB_Partition` and moves the data to its new part assignments by calling `Zoltan_Migrate`. After migration, `Zoltan_LB_Free_Data` frees the arrays returned by `Zoltan_LB_Partition`. The application then proceeds with its computation using the newly balanced partition. Partitioning and computation can occur in many iterations of the application, with part assignments changing to adjust for changes in the computation. After the iterations are completed, the

Table 2
Zoltan callback functions and their return values

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

Note: Applications need to implement only the callbacks required by the specific capability (partitioning, coloring, ordering, migration) and methods (geometric, graph-based, hypergraph-based) the application is using.

application calls `Zoltan_Destroy` to free the memory allocated in `Zoltan_Create`, and completes its execution by returning the results of the computation.

The basic set-up of Zoltan – initializing, allocating memory, setting parameters, and registering callback functions, and freeing memory – is the same regardless of whether one uses Zoltan for partitioning, ordering or coloring. Only the operations in the iteration loop would change if ordering (`Zoltan_Order`) or coloring (`Zoltan_Color`) were needed.

3.2. Isorropia

Isorropia was first designed to provide load-balancing capabilities for Trilinos objects. It provided an easy and convenient way to perform partitioning and repartitioning on Epetra objects. Isorropia can be viewed as an interface between Trilinos' Epetra matrices and Zoltan callbacks. Starting with Trilinos v9, Isorropia provides access to most Zoltan features related to matrices; therefore, it is possible for Trilinos users to partition, color or order `Epetra_RowMatrix` objects without having to write any specific Zoltan callbacks. Isorropia is designed to also extend Zoltan, by applying higher level algorithms using Zoltan kernels.

As with most Trilinos packages, Isorropia is written in C++, and an effort has been made to obtain a convenient and robust interface for the user. Isorropia capabilities are described by interfaces (abstract classes) for which an Epetra implementation is provided. However, an implementation based on the templated matrix/vector classes in Tpetra [36] will use the same interface.

The interface by itself is divided in three main classes – `Isorropia::Colorer`, `Isorropia::Orderer` and `Isorropia::Partitioner` – which are designed to perform the associated task. As all these tasks have similarities, they all inherit from a `Isorropia::Operator` class. It provides a standard way to compute the task and also provides accessors to the result. Indeed, coloring, ordering or partitioning processes have almost identical signatures: they compute on a matrix (only the structure of the matrix is needed) and their result is an integer (color, number, part assignment) associated with every row and/or column. Having `Isorropia::Operator` as a common ancestor provides a standard and coherent way to access results. In our case, the bracket operator is the most natural way to access local results, but more advanced methods are also available.

Isorropia's interface is implemented for Epetra objects. An UML class diagram for the main classes is presented Fig. 7.
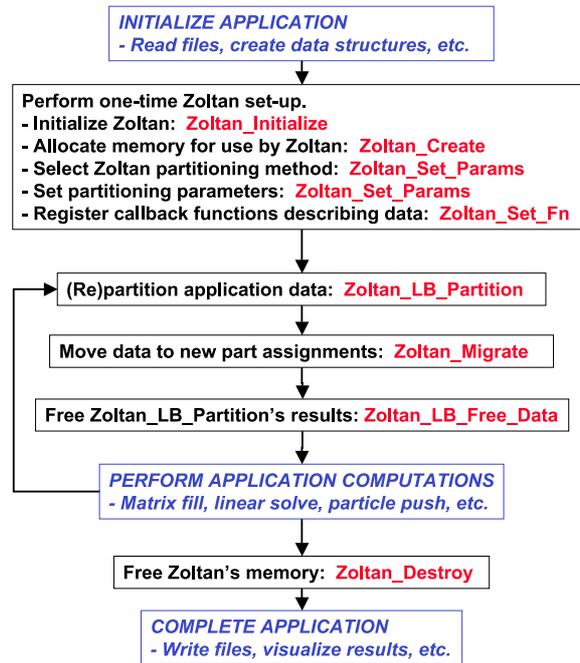
Fig. 6. Use of Zoltan partitioning in a typical dynamic application. Calls to Zoltan functions are shown in red; application operations are in blue. (The colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0342.)
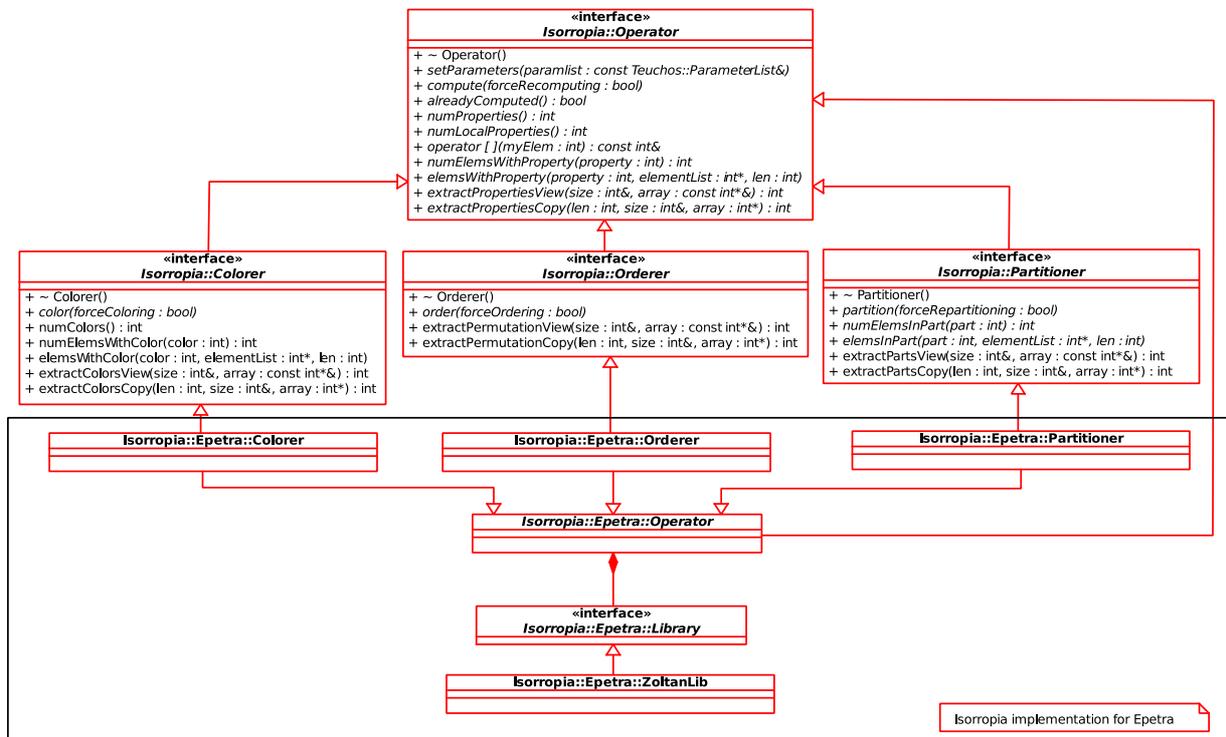

Fig. 7. Isorropia main UML class diagram. This architecture allows flexibility to support different combinatorial kernels (provided by Zoltan here) as well as object target types (here Epetra objects). (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0342.)

While `Isorropia::Operator` is used to share the methods for defining problems or for accessing results, its (partial) implementation `Isorropia::Epetra::Operator` is responsible for providing code used to access data.

For Epetra matrices, we use only Zoltan for combinatorial kernels; however, Zoltan calls are issued through a specific class `Isorropia::Epetra::ZoltanLib` which provides an implementation of the `Isorropia::Epetra::Library` interface. Thus, using another combinatorial toolkit requires mainly implementation of this library interface.

Another goal of Isorropia was to provide some highly specialized solutions for combinatorial problems on Trilinos objects. Thus, the architecture is designed to be more than a simple wrapper on Zoltan. For example, two-dimensional matrix partitioning in Isorropia is implemented as an extension of the partitioner class.

The examples below show that Isorropia greatly reduces the number of lines of source code for Epetra users. First, given an Epetra sparse matrix, one can obtain a row partitioning in one line:

```
// crsmatrix is an Epetra_CrsMatrix
Isorropia::Epetra::Partitioner
   partitioner(crsmatrix);
```

By default, the constructor also performs the partitioning. In more advanced use cases, the constructor just creates the object and the `partition()` method is explicitly called later:

```
// crsmatrix is an Epetra_CrsMatrix
Teuchos::ParameterList params;
params.set("PARTITIONING_METHOD",
   "GRAPH"); // default is HYPERGRAPH
params.set("IMBALANCE_TOL", "1.03");
// allow 3% imbalance
Isorropia::Epetra::Partitioner
   partitioner(crsmatrix, params,
               false);
// do some other work ...
partitioner.partition();
```

The result of the partitioning is stored in the `Partitioner` and can be accessed in several ways. A common use case is to redistribute the distributed object according to the computed partition. This is easily done using a redistributor:

```
// Assume partitioner was
   constructed as above
Isorropia::Epetra::Redistributor
  rd(partitioner);
// Redistribute the matrix
newmatrix =
   rd.redistribute(crsmatrix, true);
```

It is possible to reuse the redistributor to also distribute vectors or another matrix.

Isorropia has its own parameters defined by a `Teuchos::ParameterList` with options names that are more relevant for an Epetra user. However, it is still possible to pass advanced parameters directly to Zoltan.

To summarize, Isorropia provides solutions to combinatorial problems such as partitioning, coloring or ordering by an easy to use common interface which currently deals with Epetra objects. Isorropia also provides methods to apply these results so they can be used in complex Epetra-based codes with minimal effort.

4. Next generation: Zoltan2

We have begun a refactoring of Zoltan and Isorropia, embodied in the new Trilinos package Zoltan2. Zoltan2 combines features of Isorropia and Zoltan in a uniform abstract interface. It will continue to support a broad range of applications and provide the most commonly used features of Zoltan and Isorropia. It will also include new features and algorithms designed specifically for exascale computing, including support for heterogeneous computers, multicore algorithms, and scalability to million-way parallelism. This refactoring is motivated by a number of algorithmic and software engineering issues.

Zoltan2 enables future combinatorial algorithm research to better support exascale computing. As computer architectures evolve to include increasing heterogeneity, new combinatorial algorithms are required. Zoltan was designed strictly for distributed memory environments, but hybrid MPI+threads programming models are becoming common in parallel computers built from multicore processors. For applications using hybrid programming models, partitioning with awareness of the underlying computing architecture and memory hierarchies can increase data locality for

efficient computation, and ordering algorithms and iterators can improve data access within multicore architectures. Within the combinatorial algorithms themselves, adopting hybrid programming models can improve the scalability of partitioning, coloring and ordering. These features will be supported in Zoltan2 through incorporation of architecture models and system information from systems such as the OVIS resource analysis toolkit [5] and Portable Hardware Locality toolkit (hwloc) [6], as well as node-level programming primitives from the Trilinos' Kokkos node package [37].

Zoltan2 exploits recent advances in compiler technology and software design principles. Next-generation packages in Trilinos (e.g., the Tpetra [36] matrix/vector classes and Belos [38] linear solvers) rely on templating to enable computations on large-scale problems (i.e., those with more than two billion (maximum integer value on a 32-bit system) rows and non-zeros). Global and local row/column numbers are templated separately to accommodate integers, long integers, long long integers, etc. Scalar values are also templated to allow reduced, increased, or mixed precision computations. While Zoltan has been modified to allow more than two billion items to be partitioned, it does not yet support the full range of templating options available in next-generation Trilinos packages. In addition to templated ordinal types, Zoltan2 allows templated types for global and local IDs, analogous to the arbitrary IDs in Zoltan. Data types for weights are also templated.

Zoltan2 also provides a more natural interface for application developers. At the time of Zoltan's initial design, C++ compilers were immature, particularly on massively parallel computers such as ASCI Red. Thus, while using an object-oriented design, the Zoltan library is written in C. Isorropia simplifies use of Zoltan for applications with Epetra matrices, but other users must rely on the original C interface. To preserve backward compatibility for long-time users, Zoltan's interface is largely unchanged since its original design and can be improved by the user feedback and lessons we have learned through many years of Zoltan use. In particular, Zoltan2 separates the user's data model from the data model used in Zoltan2's algorithms, no longer requiring users to understand the details of graphs and hypergraphs to use partitioning, ordering and coloring algorithms.

Zoltan2 has an improved object-oriented design compared to Zoltan; the improvements are enabled by class hierarchies available in C++. We use a combina-

tion of templating and inheritance to abstract user data from the combinatorial model and algorithms. A high-level view of our design is shown in Fig. 8.

At the user level, users may provide parameters controlling the choice of combinatorial model and algorithms, and configuration options such as debugging levels and memory allocation routines. Users describe their application data through `InputAdapter` objects. Several types of `InputAdapter` objects – meshes, matrices, vectors, particles, networks – are defined to support different applications. These adapters allow users to describe their data in its native format, rather than mapping it to a particular model (e.g., graph, hypergraph) as in Zoltan. For example, the matrix `InputAdapter` has methods returning data about matrix rows and non-zeros; the mesh `InputAdapter` describes the elements-node adjacencies of a mesh. An `InputAdapter` also describes how to redistribute data after partitioning or ordering. In the future, users will be able to provide descriptions of the machine network and/or node architecture as well. While a number of input classes are shown, many are optional, allowing Zoltan2 to provide a novice interface suitable for the most common use cases (e.g., TPetra matrix partitioning).

The user's `InputAdapter`, parameters and configuration information are used to create a `Problem`, which may be a `PartitioningProblem`, `OrderingProblem`, or `ColoringProblem`. As in Isorropia, the `Problem` manages the requested operation (partitioning, ordering, coloring) and maintains state containing the result of the operation. Using the given input, the `Problem` creates an appropriate `Model` – the combinatorial representation of the input. This model describes how to use functions in the `InputAdapter` to build the representation needed for an algorithm, bridging the separation between the application data and the data structures needed by a combinatorial algorithm. This model is then passed to an `Algorithm` that performs the requested operation (partitioning, ordering, coloring); as in Zoltan and Isorropia, geometric, graph, and hypergraph-based algorithms will be supported. The `Algorithm` populates a `Solution` within the `Problem`. As with the `Problem` class, the `Solution` class has specific subclasses `PartitioningSolution` (containing import and export lists and part maps), `OrderingSolution` (containing permutations and iterators), and `ColoringSolution` (containing coloring maps and iterators).
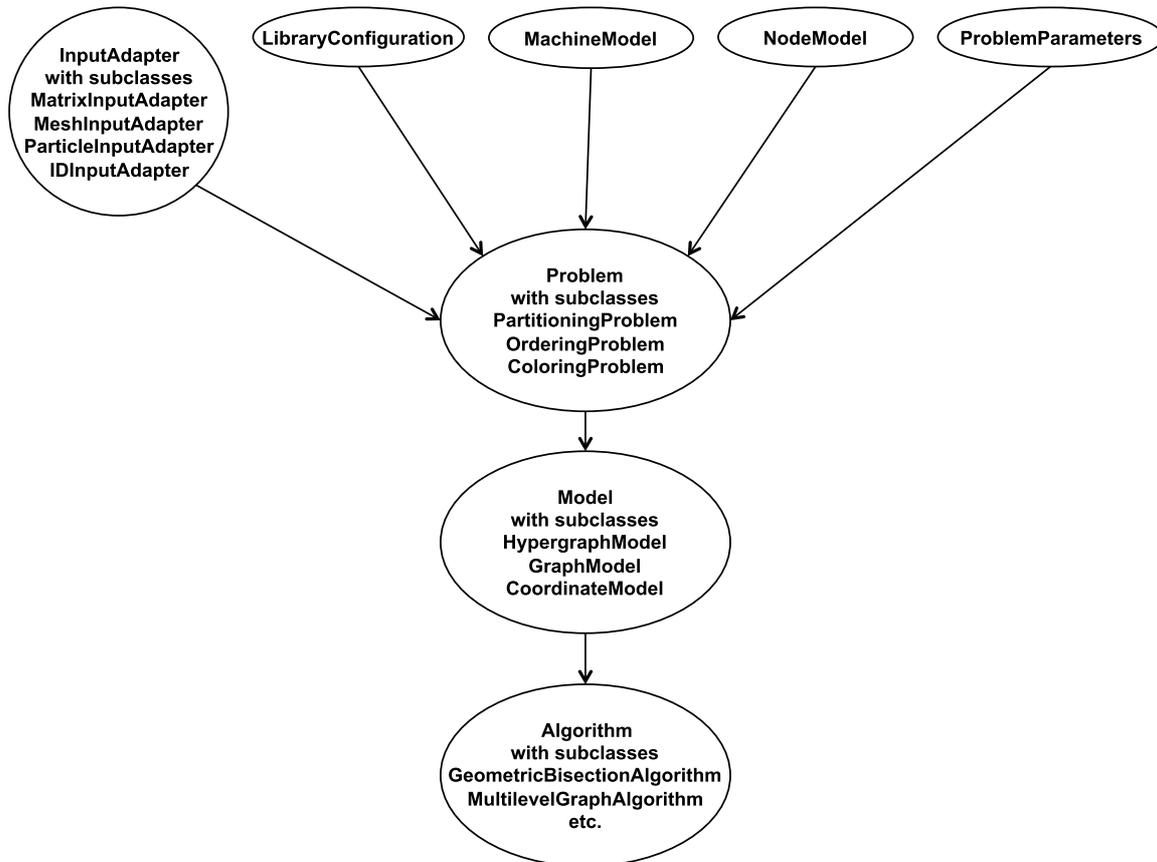
Fig. 8. High-level view of Zoltan2 design.

Initial release of Zoltan2 in Trilinos is scheduled for 2012. It will support only a small subset of Zoltan/Isorropia capabilities initially, but anticipated future development will provide richer capabilities in future Trilinos releases.

5. Conclusion and future work

We have described the design of two Trilinos packages focused on combinatorial scientific computing: the Zoltan toolkit of partitioning, ordering and coloring algorithms and the Isorropia toolkit of combinatorial algorithms for Trilinos' Epetra matrix/vector classes. These packages provide critical capabilities for a wide range of scientific computing applications, leading to improved application performance through better load balancing, data locality, memory usage, and algorithmic efficiency. We also introduced our next-generation package for combinatorial algorithms, Zoltan2, that will support more robustly the needs of exascale computing and integrate more closely with other Trilinos packages.

Our future efforts focus on Zoltan2. In addition to implementing the classes and algorithms that make up Zoltan2's base capability, we will pursue a number of research directions. We will examine architecture-aware partitioning strategies, accounting for the underlying machine architecture and resource state in making partitioning decisions. We will support multi-core architectures through improved data layouts via ordering and partitioning. And we will begin implementation of thread-enabled combinatorial algorithms. A greater amount of data can be made available to each task via shared memory than via distributed memory; we can exploit this fact in heuristic algorithms to make better decisions in partitioning, ordering and coloring. As with Zoltan and Isorropia, we will integrate and distribute Zoltan2 with the Trilinos toolkit.

Acknowledgements

Appendix

Below is a complete example of a program using Zoltan for geometric partitioning in a simple particle-based application. The application's data is stored in a simple structure `User_Data`, containing particles' unique IDs and their coordinates in space.

Four Zoltan query functions are needed to perform geometric partitioning. The `Zoltan_Num_Obj_Fn` `num_obj` returns the number of particles owned by the process. The `Zoltan_Obj_List_Fn` `obj_list` returns the global IDs of each of the particles owned by the process, as well as local IDs that are the indices into the local particles array data structure. This example does not use particle weights but if it did, they would also be returned in this function. The `Zoltan_Num_Geom_Fn` `num_geom` returns the geometric dimension of the application's domain – in this case, three. The `Zoltan_Geom_Multi_Fn` `geom_multi` returns the geometric coordinates of each of the particles. The local IDs provided in `obj_list` are used here to quickly locate the coordinates for each requested particle.

In the `main` program, the application initializes MPI and its particle data. It creates a `Zoltan_Struct` data structure by calling `Zoltan_Create`. This data structure contains state information needed by Zoltan; it is an input argument to subsequent Zoltan function calls. The program registers pointers to its callback functions through calls to `Zoltan_Set_Fn`. It also sets Zoltan parameters via `Zoltan_Set_Param`, in this case, selecting RCB to be the load-balancing method. It then calls `Zoltan_LB_Partition` to compute a new partition. The return arguments of `Zoltan_LB_Partition` are lists of particles to be imported to the process from other processes, and particles to be exported from the process to other processes. Both process ranks and part numbers are included in the import and export lists, to allow the number of parts to differ from the number of processes. These import and export lists are then input to a routine to actually move the particles to the new partition. In the example, we assume the user has provided his own function to perform the migration. However, the `Zoltan_Migrate` function can be used to move data. `Zoltan_Migrate` requires additional callback functions to specify the size of the particles to be moved, along with methods for loading particle data into communication buffers before migration and unloading it into the application data structures after migration. Since the import and export lists are allocated by Zoltan, the user should use `Zoltan_LB_Free_Part` to free the lists after migration. At this point, the application has a balanced decomposition that maintains geometric locality of the particles, and it is ready to compute. It can use this partition throughout its computation, or repartition using the same `Zoltan_Struct` and callbacks as needed. When the application no longer needs to repartition, it deallocates its `Zoltan_Struct` by calling `Zoltan_Destroy`.

```c
1  #include "zoltan.h"
2
3  ///////////////// Application data for particle simulation ////////////
4  struct UserData {
5    int numMyParticles;
6    struct Particle {
7      int id;           // unique global name for particle
8      double x, y, z;   // geometric coordinates for particle
9      // ... solution values, etc. ...
10   } *myParticles;
11 };
```

```
12
13  // //////////////// User−provided query functions for IDs  ////////////////
14
15  // Return number of particles on this process.
16  int num_obj(void *data, int *ierr)
17  {
18    // Cast data pointer provided in Zoltan_Set_Fn to application data type.
19    struct UserData *myData = (struct UserData *) data;
20    *ierr = ZOLTAN_OK;
21    return myData->numMyParticles;
22  }
23
24  // Return global and local ids for particles on this process.
25  // No particle weights used in this example.
26  void obj_list(void *data, int nge, int nle,
27                         ZOLTAN_ID_PTR globalIDs, ZOLTAN_ID_PTR localIDs,
28                         int wgtDim, float *objWgts, int *ierr)
29  {
30    // Cast data pointer provided in Zoltan_Set_Fn to application data type.
31    struct UserData *myData = (struct UserData *) data;
32
33    // Copy global ID for each particle into globalIDs array.
34    // Pass local index into particle array as the localID.
35    for (int i = 0; i < myData->numMyParticles; i++) {
36      globalIDs[i] = myData->myParticles[i].id;
37      localIDs[i]  = i;
38    }
39    *ierr = ZOLTAN_OK;
40  }
41
42  // ////////// User−provided query functions for coordinates  //////////
43  // Return dimension of problem.
44  int num_geom(void *data, int *ierr)
45  {
46    return 3;         // 3D problem
47  }
48
49  // Return coordinates for objects requested by Zoltan in globalIDs array.
50  void geom_multi(void *data, int nge, int nle, int numObj,
51                         ZOLTAN_ID_PTR globalIDs, ZOLTAN_ID_PTR localIDs,
52                         int dim, double *geomVec, int *ierr)
53  {
54    // Cast data pointer provided in Zoltan_Set_Fn to application data type.
55    struct UserData *myData = (struct UserData *) data;
56
57    // Copy coordinates for particle globalID[i] (with localID[i])
58    // into geomVec.
59    int i, j = 0;
60    for (i = 0; i < numObj; i++) {
61      geomVec[j++] = myData->myParticles[localIDs[i]].x;
62      if (dim > 1) geomVec[j++] = myData->myParticles[localIDs[i]].y;
63      if (dim > 2) geomVec[j++] = myData->myParticles[localIDs[i]].z;
64    }
65    *ierr = ZOLTAN_OK;
66  }
67
68  // //////////////////////// Main program    ////////////////////////
69  int main(int narg, char **arg)
70  {
71    // User creates and initializes his data here as appropriate.
72    MPI_Init(&narg, &arg);
```

```
73   struct UserData myData;
74   user_initialize(&myData);
75
76   // initialize Zoltan and create Zoltan data structure.
77   float version;
78   int ierr = Zoltan_Initialize(narg, arg, &version);
79   struct Zoltan_Struct *zz = Zoltan_Create(MPI_COMM_WORLD);
80
81   // register Zoltan callback functions.
82   ierr = Zoltan_Set_Fn(zz, ZOLTAN_NUM_OBJ_FN_TYPE,    (void (*)())num_obj,    &myData);
83   ierr = Zoltan_Set_Fn(zz, ZOLTAN_OBJ_LIST_FN_TYPE,   (void (*)())obj_list,   &myData);
84   ierr = Zoltan_Set_Fn(zz, ZOLTAN_NUM_GEOM_FN_TYPE,   (void (*)())num_geom,   &myData);
85   ierr = Zoltan_Set_Fn(zz, ZOLTAN_GEOM_MULTI_FN_TYPE, (void (*)())geom_multi, &myData);
86
87   // set Zoltan parameters: select RCB partitioning
88   ierr = Zoltan_Set_Param(zz, "LB_METHOD", "RCB");
89
90   // arguments returned by Zoltan_LB_Partition; arrays are allocated by Zoltan.
91   int anyChanges = 0;
92   int ngid, nlid, numImport, numExport;
93   ZOLTAN_ID_PTR importGIDs, importLIDs, exportGIDs, exportLIDs;
94   int *importProc, *importPart, *exportProc, *exportPart;
95
96   // call Zoltan to compute a new partition.
97   ierr = Zoltan_LB_Partition(zz, &anyChanges, &ngid, &nlid,
98                 &numImport, &importGIDs, &importLIDs, &importProc, &importPart,
99                 &numExport, &exportGIDs, &exportLIDs, &exportProc, &exportPart);
100
101   // migrate data as specified by import/export lists;
102   // users can use Zoltan_Migrate if desired, but it is not required.
103   if (anyChanges)
104     migrate(&myData, numImport, importGIDs, importLIDs, importProc, importPart,
105                   numExport, exportGIDs, exportLIDs, exportProc, exportPart);
106
107   // free data allocated by Zoltan.
108   Zoltan_LB_Free_Part(&importGIDs, &importLIDs, &importProc, &importPart);
109   Zoltan_LB_Free_Part(&exportGIDs, &exportLIDs, &exportProc, &exportPart);
110
111   // the Zoltan_Struct can be used for additional repartitioning, but
112   // when it is no longer needed, it should be destroyed.
113   Zoltan_Destroy(&zz);
114
115   // now the user can compute with a balanced partition with geometric
116   // locality for the particles.
117   user_computation(&myData);
118
119   MPI_Finalize();
120 }
```
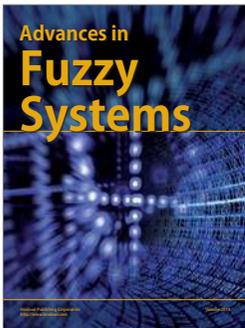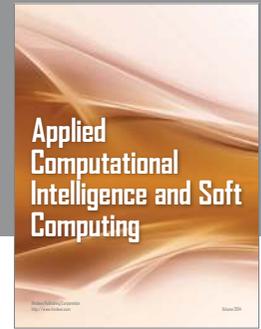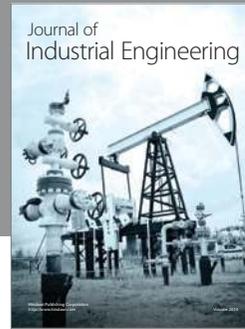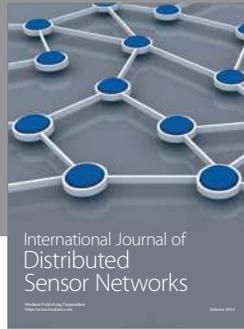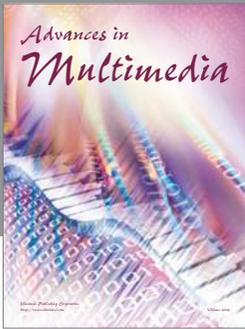
References

[1] C. Aykanat, A. Pınar and Ü.V. Çatalyürek, Permuting sparse rectangular matrices into block-diagonal form, *SIAM J. Sci. Comput.* **26**(6) (2004), 1860–1879.

[2] M.J. Berger and S.H. Bokhari, A partitioning strategy for nonuniform problems on multiprocessors, *IEEE Trans. Comput.* **C-36**(5) (1987), 570–580.

[3] D. Bozdağ, Ü.V. Çatalyürek, A.H. Gebremedhin, F. Manne, E.G. Boman and F. Özgünner, Distributed-memory parallel algorithms for distance-2 coloring and related problems in derivative computation, *SIAM J. Sci.Comput.* **32**(4) (2010), 2418–2446.

[4] D. Bozdağ, A.H. Gebremedhin, F. Manne, E.G. Boman and Ü.V. Çatalyürek, A framework for scalable greedy coloring on distributed memory parallel computers, *J. Parallel Distrib. Comput.* **68**(4) (2008), 515–535.

[5] J. Brandt, B. Debusschere, A. Gentile, J. Mayo, P. Pébay, D. Thompson and M. Wong, OVIS-2: a robust distributed architecture for scalable RAS, in: *Proc. 4th Workshop on System Management Techniques, Processes and Services at 22nd IEEE Int. Parallel and Distributed Processing Symposium*, Miami, FL, 2008.

[6] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault and R. Namyst, hwloc: a generic framework for managing hardware affinities in HPC applications, in: *Proc. 18th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Computing*, Pisa, 2010.

[7] T.N. Bui and C. Jones, A heuristic for reducing fill-in sparse matrix factorization, in: *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, SIAM, 1993, pp. 445–452.

[8] Ü.V. Çatalyürek and C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, *IEEE Trans. Parallel Dist. Syst.* **10**(7) (1999), 673–693.

[9] Ü.V. Çatalyürek and C. Aykanat, PaToH: a multilevel hypergraph partitioning tool, Version 3.0, Bilkent University, Department of Computer Engineering, Ankara, PaToH is available at: http://bmi.osu.edu/~umit/software.htm, 1999.

[10] Ü.V. Çatalyürek, C. Aykanat and E. Kayaaslan, Hypergraph partitioning-based fill-reducing ordering for symmetric matrices, *SIAM J. Sci. Comput.* **33** (2011), 1996–2023.

[11] Ü.V. Çatalyürek, C. Aykanat and B. Uçar, On two-dimensional sparse matrix partitioning: Models, methods and a recipe, *SIAM J. Sci. Comput.* **32**(2) (2010), 656–683.

[12] Ü.V. Çatalyürek, E.G. Boman, K.D. Devine, D. Bozdağ, R.T. Heaphy and L.A. Riesen, A repartitioning hypergraph model for dynamic load balancing, *J. Parallel Distrib. Comput.* **69**(8) (2009), 711–724.

[13] Ü.V. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar and A. Pothen, Distributed-memory parallel algorithms for matching and coloring, in: *Proc. Int. Symp. on Parallel and Distributed Processing, Workshops and PhD Forum, Workshop on Parallel Computing and Optimization*, IEEE Press, 2011, pp. 1966–1975.

[14] Ü.V. Çatalyürek, B. Uçar and C. Aykanat, Hypergraph partitioning, in: *Encyclopedia of Parallel Computing*, D. Padua, ed., Springer, 2011, pp. 871–881.

[15] T.F.C. Chan and T.P. Mathew, The interface probing technique in domain decomposition, *SIAM J. Matrix Anal. Appl.* **13** (1992), 212–238.

[16] C. Chevalier and F. Pellegrini, PT-SCOTCH: a tool for efficient parallel graph ordering, *Parallel Comput.* **34**(6–8) (2008), 318–331.

[17] T.F. Coleman and J.J. Moré, Estimation of sparse Jacobian matrices and graph coloring problems, *SIAM J. Numer. Anal.* **1**(20) (1983), 187–209.

[18] J.C. Culberson, Iterated greedy graph coloring and the difficulty landscape, Technical Report TR 92-07, University of Alberta, June 1992.

[19] A.R. Curtis, M.J.D. Powell and J.K. Reid, On the estimation of sparse Jacobian matrices, *J. Inst. Math. Appl.* **13** (1974), 117–119.

[20] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: *Proceedings of the 24th National Conference*, ACM, New York, NY, USA, 1969, pp. 157–172.

[21] T. Davis, J. Gilbert, S. Larimore and E. Ng, A column approximate minimum degree ordering algorithm, *ACM Trans. Math. Software* **30**(3) (2004), 353–376.

[22] J. DeBlasio, K. Ewing, A. Lawrence and M. Leece, Exploring the feasibility of 2D matrix partitioning, Technical report, Department of Computer Science, Harvey Mudd College, May 2011.

[23] K. Devine, E. Boman, R. Heaphy, B. Hendrickson and C. Vaughan, Zoltan data management services for parallel dynamic applications, *Comput. Sci. Eng.* **4**(2) (2002), 90–97.

[24] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling and Ü.V. Çatalyürek, Parallel hypergraph partitioning for scientific computing, in: *Proc. 20th Int. Parallel and Distributed Processing Symp.*, IEEE, 2006.

[25] I.S. Duff and G.A. Meurant, The effect of ordering on preconditioned conjugate gradients, *BIT* **29** (1989), 635–657.

[26] M. Garey, D. Johnson and L. Stockmeyer, Some simplified NP-complete graph problems, *Theoret. Comput. Sci.* **1** (1976), 237–267.

[27] A.H. Gebremedhin, F. Manne and A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, *SIAM Rev.* **47**(4) (2005), 629–705.

[28] A.H. Gebremedhin, D. Nguyen, M.M.A. Patwary and A. Pothen, ColPack: graph coloring software for sparse derivative matrix computation and beyond, *ACM Trans. Math. Software* (2012), to appear.

[29] J.A. George, Nested dissection of a regular finite element mesh, *SIAM J. Numer. Anal.* **10** (1973), 345–363.

[30] L. Grigori, E. Boman, S. Donfack and T. Davis, Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization, *SIAM J. Sci. Comp.* **32**(6) (2010), 3426–3446.

[31] B. Hendrickson and T.G. Kolda, Graph partitioning models for parallel computing, *Parallel Comput.* **26** (2000), 1519–1534.

[32] B. Hendrickson and R. Leland, A multilevel algorithm for partitioning graphs, in: *Proc. Supercomputing'95*, ACM, 1995.

[33] B. Hendrickson and E. Rothberg, Effective sparse matrix ordering: just around the bend, in: *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, Atlanta, GA, 1997.

[34] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring and A. Williams, An overview of Trilinos, Technical Report SAND2003-2927, Sandia National Laboratories, Albuquerque, NM, 2003.

[35] http://trilinos.sandia.gov/packages/epetra.

[36] http://trilinos.sandia.gov/packages/tpetra.

[37] http://trilinos.sandia.gov/packages/kokkos.

[38] http://trilinos.sandia.gov/packages/belos.

[39] M. Jones and P. Plassmann, Scalable iterative solution of sparse linear systems, *Parallel Comput.* **20**(5) (1994), 753–773.

[40] G. Karypis and V. Kumar, METIS: unstructured graph partitioning and sparse matrix ordering system, Technical report, Department of Computer Science, University of Minnesota, 1995, available at: http://www.cs.umn.edu/~karypis/metis.

[41] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comp.* **20**(1) (1998), 359–392.

[42] G. Karypis, K. Schloegel and V. Kumar, ParMETIS: parallel graph partitioning and sparse matrix ordering library, version 3.1, Technical report, Department of Computer Science, University of Minnesota, 2003, available at: http://www-users.cs.umn.edu/~karypis/metis/parmetis/download.html.

[43] S.O. Krumke, M. Marathe and S. Ravi, Models and approximation algorithms for channel assignment in radio networks, *Wireless Networks* **7**(6) (2001), 575–584.

[44] V.S.A. Kumar, M.V. Marathe, S. Parthasarathy and A. Srinivasan, End-to-end packet-scheduling in wireless ad-hoc networks, in: *Proceedings of the 15th Annual ACM–SIAM Symp. on Discrete Algorithms*, SIAM, 2004, pp. 1021–1030.

[45] F. Manne, A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices, in: *Proceedings of Para*, Lecture Notes in Computer Science, Vol. 1541, Springer, 1998, pp. 332–336.

[46] D.W. Matula, A min–max theorem for graphs with application to graph coloring, *SIAM Rev.* **10** (1968), 481–482.

[47] Message Passing Interface Forum, MPI: a message-passing interface standard, May 1994, available at: http://www.mpi-forum.org.

[48] F. Pellegrini, PT-SCOTCH 5.1 user's guide, Technical report, LaBRI, September 2008.

[49] J.R. Pilkington and S.B. Baden, Partitioning with spacefilling curves, CSE Technical Report CS94–349, Department of Computer Science and Engineering, University of California, San Diego, CA, 1994.

[50] A. Pınar, Combinatorial algorithms in scientific computing, PhD thesis, University of Illinois at Urbana–Champaign, 2001.

[51] S. Rajamanickam, E.G. Boman and M.A. Heroux, ShyLU: a hybrid–hybrid solver for multicore platforms, in: *Proc. of 26th IEEE Int. Parallel and Distributed Processing Symp.*, IEEE, 2012.

[52] Y. Saad, ILUM: a multi-elimination ILU preconditioner for general sparse matrices, *SIAM J. Sci. Comput.* **17** (1996), 830–847.

[53] M. Sala, W. Spotz and M. Heroux, PyTrilinos: high-performance distributed-memory solvers for Python, *ACM Trans. Math. Software* **34**(2) (2008), 1–33.

[54] M. Sala, K.S. Stanley and M.A. Heroux, On the design of interfaces to sparse direct solvers, *ACM Trans. Math. Software* **34**(9) (2008), 1–22.

[55] A.E. Sarıyüce, E. Saule and Ü.V. Çatalyürek, Improving graph coloring on distributed-memory parallel computers, in: *Proceedings of the 18th Annual International Conference on High Performance Computing*, Bangalore, India, 2011.

[56] K. Schloegel, G. Karypis and V. Kumar, Multilevel diffusion algorithms for repartitioning of adaptive meshes, *J. Parallel Distrib. Comput.* **47**(2) (1997), 109–124.

[57] C. Siefert and E. de Sturler, Probing methods for saddle-point problems, *Electr. Trans. Numer. Anal.* **22** (2006), 163–183.

[58] H.D. Simon, Partitioning of unstructured problems for parallel processing, *Comput. Syst. Eng.* **2** (1991), 135–148.

[59] M.M. Strout, L. Carter, J. Ferrante, J. Freeman and B. Kreaseck, Combining performance aspects of irregular Gauss–Seidel via sparse tiling, in: *LCPC*, W. Pugh and C.-W. Tseng, eds, Lecture Notes in Computer Science, Vol. 2481, Springer, 2002, pp. 90–110.

[60] V.E. Taylor and B. Nour-Omid, A study of the factorization fill-in for a parallel implementation of the finite element method, *Int. J. Numer. Meth. Eng.* **37** (1994), 3809–3823.

[61] J.D. Teresco, M.W. Beall, J.E. Flaherty and M.S. Shephard, A hierarchical partition model for adaptive finite element computation, *Comput. Methods Appl. Mech. Eng.* **184** (2000), 269–285.

[62] W.F. Tinney and J.W. Walker, Direct solution of sparse network equations by optimally ordered triangular factorization, *Proc. IEEE* **55** (1967), 1801–1809.

[63] M.S. Warren and J.K. Salmon, A parallel hashed oct-tree *n*-body algorithm, in: *Proc. Supercomputing'93*, Portland, OR, 1993.