Effecting Parallel Graph Eigensolvers Through Library Composition

Alex Breuer, Peter Gottschling, Douglas Gregor, Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{abreuer,pgottsch,dgregor,lums}@osl.iu.edu

***Abstract*— Many interesting problems in graph theory can be reduced to solving an eigenproblem of the adjacency matrix or Laplacian of a graph. Given the availability of high-quality linear algebra and graph libraries, one might expect that one could merely use a graph data structure within a eigensolver. However, conventional libraries are rigidly constructed, requiring conversion to library-specific data structures or using heavyweight abstraction methods that prevent efficient composition.**

The Generic Programming methodology addresses the problems of reusability and composability by careful factorization of a domain into efficient library abstractions. We describe the composition process that makes the data structures from a library supporting one domain usable with the algorithms of another library for a disjoint domain without conversion or heavyweight abstractions. To illustrate the process, we compose two separately-developed libraries, one for solving eigenproblems sequentially and the other for solving graph problems in parallel, effecting an efficient, scalable parallel graph eigensolver.

Keywords: Eigenvalues, Parallel Graph Libraries, Generic Programming

I. Introduction

Many problems in graph theory can be reduced to solving an eigenproblem of the adjacency matrix or the Laplacian of a graph. These features are useful in applications from information retrieval to engineering analysis. For example, eigenvalues and eigenvectors can be used to detect disconnected or nearly disconnected components in a graph [3], and for vertex clustering [9], [12]. Eigenvalues also have applications in estimating solutions to intractable problems, such as max-cut [10] and detecting large cliques in graphs [1].

Ideally, we would simply compute the eigenvalues of a graph using a container from a graph library and an algorithm from an eigensolver library. In this way, we would sacrifice nothing; we would have both the eigensolver and graph routines available for use on the same container. Unfortunately, this solution is not possible with conventional libraries: though there exist efficient algorithms for solving sparse eigenproblems for matrices and there exist scalable distributed graph libraries, composing a linear algebra eigensolver with graph containers is not possible in general. Conventional linear algebra libraries cannot operate on graph data types. Likewise, data stored in a linear algebra matrix container cannot be manipulated by graph library routines. There is no inherent reason why we cannot "have it all," that is, use both linear algebra library routines and graph library routines efficiently on the same container. In fact, sparse matrix representations from linear algebra and sparse graph representations are similar, but implementation details prevent ready exploitation of this duality. Graph libraries and matrix libraries use different data types, and despite structural similarities which may exist, explicit conversion and copying between graph containers and matrix containers becomes necessary. Such conversion introduces performance penalties and storage problems; some graphs are sufficiently large that the requirements of two simultaneous copies of data will exceed memory resources. Conversion and copying is even more difficult if the data is distributed throughout a cluster. These additional costs may be prohibitive when industrial-strength computation is required.

The Generic Programming paradigm provides a solution to this problem, allowing graph containers to be treated as a matrix within a linear algebra library. At the heart of the Generic Programming paradigm is the separation of data structures (containers) and algorithms. Algorithms assume a certain interface with a data structure, which specifies syntactic and semantic conditions. The set of these requirements is called a *concept*. An algorithm requiring concept `C` can be freely associated with any container implementing `C`. Figure 1 illustrates the way in which generic libraries interface with different kinds of containers, and contrasts that interface with conventional libraries. We can use a graph container in a generic linear algebra routine by providing a mapping to the concepts required by the linear algebra routine. Since sparse graph and sparse matrix representations have similar structure, it is possible to perform matrix operations on a graph container with the same complexity.

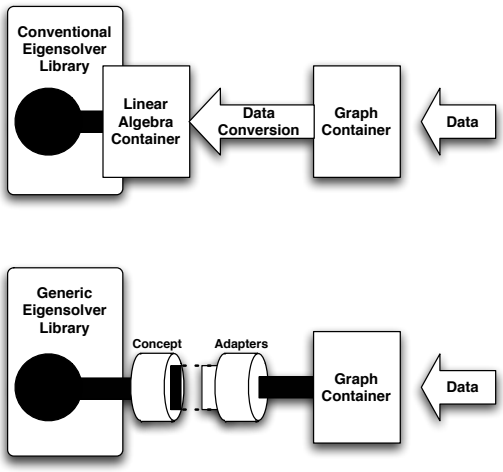


Fig. 1. Conventional libraries are hard-wired to a specific container, but generic libraries can interface with different kinds of containers through concepts.

Generic solutions need not be costly; if data structures are carefully matched with algorithms which exploit their complexity guarantees, the resulting implementation will not add any complexity to the algorithm. For example, an algorithm which performs n^2 random-accesses on a data structure will not have any additional complexity added when paired with any data structure which provides random access in constant time. Furthermore, when all genericity is evaluated at compile time, then there is no run-time overhead from our generic approach [8], [15].

Many generic libraries are available today, in particular we are concerned with the Parallel Boost Graph Library (Parallel BGL) [4], and the Iterative Eigensolver Template Library (IETL) [17], both implemented in C++ with templates. The IETL has several generic implementations of eigenvalue/eigenvector algorithms for sparse matrices, including the Lanczos method [2]. The Parallel BGL has distributed graph containers and tools for developing distributed graph applications. Though IETL and the Parallel BGL both are written in the generic style with templates, they were developed independently and with no collaboration. The Parallel BGL stores graphs distributed among processors, but the IETL contains sequential algorithms. Despite this, the generic style makes the task of composing the two libraries a matter of implementing matrix operators on graphs, and providing the syntactic adapters. No alteration of either library is necessary. And, since the Parallel BGL is a distributed graph library, the resulting eigensolver is distributed.

To this end, we introduce the reader to *concepts*, the means for reasoning about syntactic and semantic requirements of algorithms, and the methodology of Generic Programming. We discuss the concepts required by the IETL. We introduce containers from the Parallel BGL to model IETL concepts, and formulate linear algebra equivalent operations using those containers. Finally, we present performance results to illustrate the efficiency and scalability of our solution.

```
double sum(double *array, int n) {
  double s = 0;
  for ( int i = 0; i < n; ++i )
    s = s + array[ i ];
  return s;
}

float sum(node *first, node *last) {
  float s = 0;
  while (first != last ) {
    s = s + first → data;
    first = first → next;
  }
  return s;
}
```

Fig. 2. Two concrete implementations of the abstract sum() algorithm, which sum **double** values in an array or **float** values in a linked list, respectively.

II. Generic Programming

Generic Programming is a methodology for developing highly reusable, efficient software libraries. [11] At its core is a form of Platonic idealism, wherein the abstract form of an algorithm is held distinct from the many concrete implementations of that algorithm that appear as subroutines written in various programming languages. Generic programming seeks to discover the most abstract implementation of a given algorithm that is free from dependencies on particular data structures but can be translated into an efficient, concrete implementation.

A. Lifting

The Generic Programming process involves examining several concrete implementations of an algorithm, determining where similarities in the implementations exist, and then *lifting* the implementations to a more abstract formulation. Figure 2 illustrates two C++ functions that sum the values in a **double** array and a linked list storing **float**s, respectively. Although the two functions are rather different, the core algorithm is the same: they step through the sequence and accumulate a result by applying the + operator to each value in the sequence.

We can lift the implementations of Figure 2 into a single, more abstract implementation by introducing an abstraction for the underlying sequence. As in the C++ Standard Template Library [16], we adopt the iterator abstraction to permit enumeration of the elements in a sequence *without knowing how those elements are stored.* The resulting generic implementation, written as a C++ function template, is shown in Figure 3. This generic

```
template<typename Iterator, typename T>
T sum(Iterator first, Iterator last, T s) {
  while (first != last) {
    s = s + *first;
    ++first;
  }
  return s;
}
```

Fig. 3. A generic implementation of the sum() algorithm.

implementation can be automatically instantiated by the C++ compiler to produce efficient sum() implementations for arrays of **double**s, linked lists of **float**s, or any other data structure that can be traversed with an iterator.

B. Concepts

Generic algorithms such as sum() in Figure 3 are defined in terms of properties of types, not in terms of any particular type. The sum() algorithm can be applied to all types that meet certain requirements. For instance, a type used as the Iterator in sum() must support the operations ++, ∗, and !=.

We collect all of the requirements placed on a type—including syntactic, semantic, and complexity requirements—into a *concept*. For instance, the InputIterator concept [16] requires the aforementioned operations ++, ∗, != and specifies both their semantics (e.g., ++ steps to the next element) and their complexity (e.g., ++ must require only amortized constant time).

Concepts are used to express the requirements of a generic implementation. For instance, we say that any type used as the Iterator type parameter of sum() must meet the requirements of the InputIterator concept.

C. Models

When a type T meets the requirements of a concept C, we say that the type T *is a model of* C or, equivalently, T *models* C. Thus, a C++ pointer into an array of **double** values is a model of the InputIterator concept. Similarly, one can create a type list_iterator that iterates through the elements in a linked list and models InputIterator.

Generic implementations are reusable when the concepts they require are modeled by types unknown when the generic implementation was written. For instance, the Boost Graph Library (BGL) [14] contains a generic algorithm dijkstra_shortest_paths() that can be used with any model of the IncidenceGraph concept. The graph types from the LEDA graph library [6] can model the IncidenceGraph concept using a set of lightweight, non-intrusive adapters, allowing LEDA graph types to be used with BGL algorithms without any syntactic or performance overhead. Attempting to compose the libraries in the opposite direction, e.g., to use a BGL data structure with a LEDA algorithm, would require one to copy the BGL graph into a LEDA graph.

Expression	Meaning
ietl::generate(x,g)	fills the vector x with numbers from the generator g
std::swap(x,y)	swaps the two vectors x and y
ietl::dot(x,y)	calculates the scalar product of x and y
ietl::two_norm(x)	Euclidean norm of x
ietl::copy(x,y)	a deep copy y = x
y = x	a (possibly shallow) copy
x *= t	in-place scalar-vector multiplication
x /= t	in-place scalar-vector division
x += y	in-place vector-vector addition
x += t*y	in-place scaled vector-vector addition
x −= t*y	in-place scaled vector-vector subtraction
x = t*y	scalar-vector multiplication

Fig. 4. The IETL Vector concept

III. The Iterative Eigensolver Template Library

The IETL is a C++ library of algorithms for computing eigenvalues and eigenvectors of matrices. Therefore, IETL expects the containers on which it operates to behave like matrices. The IETL is written in a generic style; it is not bound to operating on only one kind of matrix container. IETL only requires that data containers model its necessary concepts.

Each generic algorithm has its own set of required concepts; since we are working with IETL, we are concerned with particular concepts it specifies. These concepts allow us to determine what additional adapters, if any, are needed to use Parallel BGL containers directly in IETL algorithms.

A. Vector

The IETL Vector concept, shown in Figure 4, represents a vector in linear algebra and contains primarily vector-vector operations, such as vector addition and the dot product. This concept captures all of the vector functionality required by IETL algorithms; any vector which implements this required functionality models the IETL Vector concept, and may be used in *any* IETL algorithm. The IETL Vector concept also contains an associated type describing the scalar type over which the vector is defined.

B. Matrix

The IETL Matrix concept, described in Figure 5, requires only a single operation: matrix-vector multiplication. Other operations common to matrix containers,

Expression	Meaning
ietl::mult(a,x,y)	calculates the matrix-vector product y=a∗x

Fig. 5. The IETL Matrix concept

such as mutation and transformation of the matrix itself, are not provided for two reasons. Eigenvalue computations which transform the input matrix tend to make sparse matrices dense. Additionally, matrix mutation is prohibitively expensive on many compressed sparse matrix formats, which would not be able to model the Matrix concept if it included mutation. Since it requires only matrix-vector multiplication, *any* matrix-like type with a corresponding vector-like type can model the IETL Matrix concept. Typically it is assumed that all results of matrix vector products lie in the same space as the multiplied vector. Some IETL algorithm allow to operate on a subspace of the vector space representable by vector data type. To assure that no vector lies outside the considered vector space, the matrix-vector product is projected into the vector space with project.

C. VectorSpace

The IETL VectorSpace concept, shown in Figure 6, ties together vectors and matrices that may be combined, e.g., through matrix-vector multiplication. A model of the VectorSpace concept also serves as a factory for vectors contained in its vector space. With the VectorSpace serving as a factory, algorithms do not need any information about how to create a vector: the VectorSpace can synthesize new vectors within the same vector space as the Matrix on which it operates. IETL VectorSpace objects must have a scalar type associated with them, and can only produce vectors capable of storing values of that scalar type. It is necessary to choose a VectorSpace whose scalar type space properly contains all possible scalar types which may be generated in one's calculations.

D. The Generic Power Method in IETL

For IETL, the Matrix, Vector and VectorSpace concepts allow for eigensolvers which are not bound to any specific representations; any input containers may be used so long as they model the IETL concepts. Thus, IETL algorithms never require alteration to allow a new container. To illustrate the use of concepts, consider Figure 7, the power method for calculating the principle eigenvector of a matrix m given a random start vector vec1. The iter object controls the iteration.

There is no type associated with the vectors used in the IETL power method. They could lie in a real or complex vector space, but the algorithm does not depend on the vector spaces which contain the vectors and matrix. They could be represented in a C array, a linked list, an STL vector or a FORTRAN array, provided that the IETL

Expression	Meaning
vector_type	the type of vectors in the vector space
scalar_type	the type of scalars in the vector space
magnitude_type	scalar type for storing norms
size_type	integral type to store the dimension of the vector space
new_vector(vs)	creates a new vector in the vector space
vec_dimension(vs)	dimension of the vector space
project(x,vs)	projects x into the vector space.

Fig. 6. The IETL VectorSpace concept

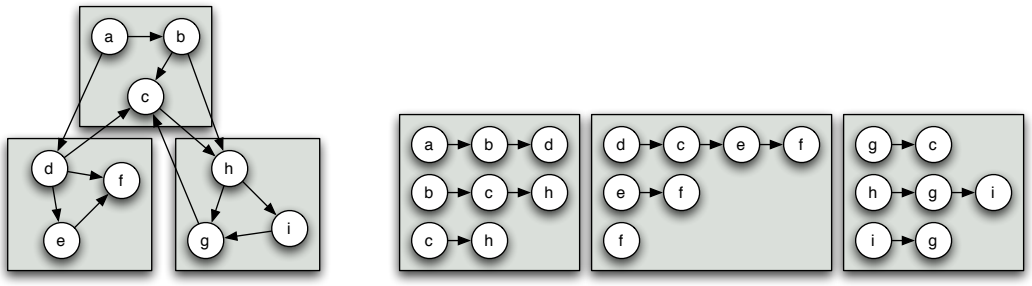
```
do {
  mult(m,vec1,vec2);
  lambda = dot(vec1,vec2);
  vec1 *= -lambda;
  vec1+=vec2;
  residual = ietl::two_norm(vec1);
  vec1=(1./ietl::two_norm(vec2))*vec2;
  ++iter;
} while(!iter.finished(residual,lambda));
```

Fig. 7. The power method as implemented in IETL.

concept is modeled for the container type. The algorithm will work with vectors and matrices from any vector space, as long as there are implementations of vector dot product, vector-scalar product, vector subtraction, Euclidean norm, and vector-scalar division. Likewise, the algorithm will work with any matrix type, as long as a matrix-vector product is implemented for the matrix and vector type. In other words, this version of the power method requires its vectors vec1, vec2 and matrix m to model concepts; its requirements would be satisfied by vectors modeling the IETL vector concept and a matrix modeling the IETL matrix concept.

IV. The Parallel Boost Graph Library

Recall that our purpose is to compute eigenvalues of a graph stored in a graph container. We have introduced the IETL as our eigensolver library, and now we introduce the distributed graph containers on which we will run IETL algorithms. The Parallel BGL provides distributed containers which are similar to the containers in the (sequential) Boost Graph Library (BGL) [14]. The vertices in a Parallel BGL graph are distributed among the various processes, with each process additionally owning all edges originating at each of its local vertices, and each process owning all properties of all local vertices or edges. Additionally, the Parallel BGL provides auxiliary



(a) Distributed graph

(b) Distributed adjacency list representation

Fig. 8. A distributed directed graph represented as an adjacency list across three processors.

data structures which map vertices to values, and a set of tools for writing distributed applications. We use the distributed_adjacency_list data type possibly enriched with numerical data associated with its edges to model the IETL Matrix concept, and use a iterator_property_map to model the Vector concept. We provide an entirely new data type to model the VectorSpace concept.

A. The Distributed Adjacency List

The distributed_adjacency_list from the Parallel BGL is based on the adjacency list from the Boost Graph Library. Its underlying storage arrangement is that of an adjacency list, which is similar to a sparse matrix; for each vertex v there is a list of all vertices u where there is an edge v, u in the graph.

The distributed_adjacency_list is automatically distributed among processors by vertices. The graph is divided into subgraphs, with each subgraph belonging to a single processor. Each processor owns all the vertices and edges in its subgraph, as well as all edges originating at local vertices, but terminating possibly at remote vertices. Adjacency list types are associated with a distribution type which provides information to resolve local vertex descriptors to global vertex descriptors.

There are three kinds of graph storage schemes; undirected, directed and bidirectional. Directed graphs only store outgoing edges, bidirectional graphs are directed graphs which store both in- and out-edges. Undirected graphs make no distinction between in- and out-edges. In a matrix context, this means graphs may be stored row-wise (for directed graphs) or *both* row-wise and column-wise. For bidirectional graphs, the transpose of the graph is available by reversing the graph, although this does not change the underlying storage scheme.

The distributed_adjacency_list can be augmented with built-in *property maps*, which associate extra values with the vertices or edges of a graph. Of particular interest is the edge weight property map, which associated scalar weights with each edge in the graph. We use the edge weight property map for storing non-zero entries in the adjacency matrix of the graph.

B. The Distributed Iterator Property Map

Conceptually, a property map associates values with the vertices or edges of a graph, but the underlying storage mechanism is left unspecified. Both the sequential and Parallel BGL provide various property map implementations. Of these, the iterator property maps are specially tuned to provide an efficient, random-access mapping of vertices or edges to values, and are therefore the primary mechanism for storing per-edge or per-vertex information external to the graph.

Distributed iterator property maps are iterator property maps that are distributed across several processes. Distributed iterator property maps use the same distribution scheme as the graph with which they are associated; a process owns all values which map to local vertices or edges. Values associated with remote vertices or edges are available through ghost cells, which are automatically generated as needed.

V. Composing the IETL and the Parallel BGL

To use IETL eigensolver routines directly on Parallel BGL containers, the Parallel BGL containers must model the IETL concepts. Thus, we determine how to perform graph operations equivalent to the linear algebra operations required by in IETL concepts. We then implement these operations via a set of adapters. Figure 9 illustrates the mapping from concepts in the IETL to Parallel BGL data types and concepts. This section describes the adapters needed to compose these two libraries.

A. Viewing Property Maps as Vectors

Any vector can be represented as a linear combination of base vectors $v = \sum_{i=1}^{n} x_i \mathbf{e}_i$, where the base vectors can be chosen orthonormal $\mathbf{e}_i^T \mathbf{e}_i = \delta_{ij} \ \ \forall i, j$. For a given basis, a vector can be uniquely represented by its coefficients $v \equiv [x_1, x_2, \ldots, x_n]^T$ omitting the base vectors. This numerical representation can be considered as a mapping from base vectors to values. For example, in two-dimensional space with Cartesian coordinates, the

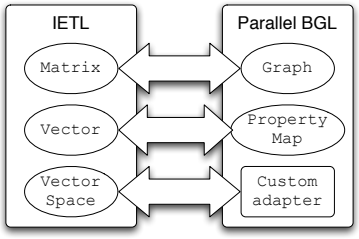


Fig. 9. Mapping between concepts in the IETL and the Parallel BGL

vector $v = [1, 3]^T$ maps the x direction to 1 and the y direction to 3.

In a graph context, vertices index the space's base vectors so that a vector representation with graph data types is a mapping of vertices to values, which can be directly implemented with vertex property maps. As a property map maps vertices to values, in a sense, it already is a vector. All that is required to extend it to model the IETL Vector concept are the vector operations, which are not defined in the Parallel BGL.

B. Viewing Adjacency Lists as Matrices

The adaptations needed for a Parallel BGL adjacency list type to model the IETL Matrix concept are lightweight and do not require significant implementation effort. As mentioned above, the only operation which the IETL Matrix concept requires is a matrix-vector product; now we consider how this may be implemented with a Parallel BGL distributed_adjacency_list. The graph analogue of a vector space is the set of vertices in the graph, and a vector is a mapping from vertices to scalar values; the definition of the graph analogue to a matrix-vector product is similar. We will adhere to the convention that if, in the $\mathbf{A}$ represented by graph G, there is a nonzero entry x in row i, column j, then there is an edge e in G from vertex i to vertex j with weight x. With this definition, we can easily extend any graph which models a concept similar to the Boost Graph Library's AdjacencyGraph to model its adjacency matrix. The result y of a graph adjacency enriched with weights $\mathbf{A}$ multiplied with a vector x is simply

$$y_v = \sum_{(v,w)\in E} weight(v, w)x_w$$

for each vertex v in E. As special case, the Laplacian of a graph is defined as all weights being -1 for $v \neq w$ and $deg(v)$ for edges (v, v). The weights do not need to be stored and the matrix vector product can be computed with implicit weights on the edges

$$y_v = deg(v)x_v + \sum_{\substack{(v,w)\in E \\ v\neq w}} -x_w$$

We use the Parallel BGL adjacency list container to model a IETL Matrix. The Lanczos algorithm implemented in IETL requires Hermitian matrices, which is in case of real values identical with symmetric matrices. This means that the graph must be symmetric $\forall v, w \in V\colon (v, w) \in E \Leftrightarrow (w, v) \in E$ and the weights must be also symmetric $\forall (v, w) \in E\colon weight(v, w) = weight(w, v)$. Naturally, for undirected graphs this property is always fulfilled.

```
template<typename Graph, typename EdgeMap,
         typename VMap1, typename VMap2>
inline void _gMult(Graph& g, EdgeMap& em,
                   VMap1& vin, VMap2& vout ) {
  using namespace boost;
  synchronize( vin );
  BGL_FORALL_VERTICES_T(u, g, Graph) {
    typename property_traits<VMap2>::value_type
      tmp = 0;
    BGL_FORALL_OUTEDGES_T(u, e, g, Graph)
      tmp += em[e] * vin[target(e, g)];
    vout[u] = tmp;
  }
}
```

Fig. 10. Our adjacency list matrix multiplication routine

Adjacency lists that contain an internal edge weight property map are weighted graphs, whose edge weights correspond to the non-zero elements in the matrix. Unweighted graphs correspond to 1-0 matrices.

The only requirement from IETL is that a matrix-vector multiplication must be defined, Figure 10 contains the adjacency list matrix multiplication routine. The EdgeMap type contains edge weights, vin and vout are vectors in the same vector space as the graph. The synchronize() is a Parallel BGL- specific interprocessor synchronization routine to guarantee data integrity.

C. A Graph Vector Space

In a graph analogue, a vector space is simply the set of weights associated with the vertices of the graph. Therefore, the dimensionality of the vector space is the size of the graph. Recall that IETL uses objects modeling the VectorSpace concept as factories for vectors, so we hard-wire our VectorSpace model to generate instances of the property map type, which models the IETL Vector concept. We also store a reference to the graph which defines the vector space. Finally, we provide a vector for scratch storage for intermediate results of vector operations. Figure 11 shows our vector space class.

VI. Evaluation

To illustrate the efficiency and scalability of our solution, we tested our adapters on two classes of random graphs: small world graphs, which exhibit good locality and a somewhat regular ring-like structure, and Erdös-Renyi graphs, which typically distribute poorly and have

```
template<typename T, typename Graph>
class g_vectorspace {
public:
  typedef iterator_property_map_wrapper<T, Graph,
    g_vectorspace<T, Graph> > vector_type;
  typedef T scalar_type;
  typedef typename std::vector<T>::size_type size_type;
  g_vectorspace( size_type n, Graph& g );
  size_type vec_dimension() const;
  vector_type new_vector() const;
  void project(vector_type&) const;
  vector_type *_scratch;
};
```

Fig. 11. A model of the IETL VectorSpace concept for Parallel BGL graphs and property maps

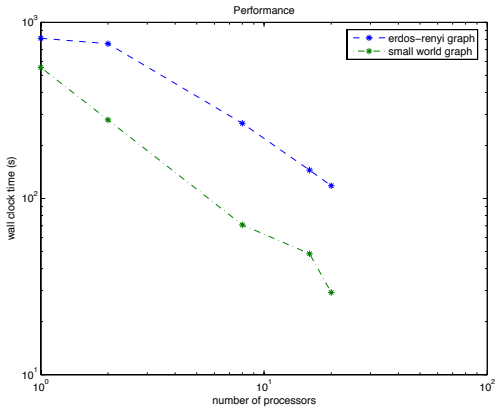


Fig. 12. Parallel Speedup of Lanczos Method using Parallel BGL containers. Graphs are on 10^6 vertices and have 12×10^6 edges

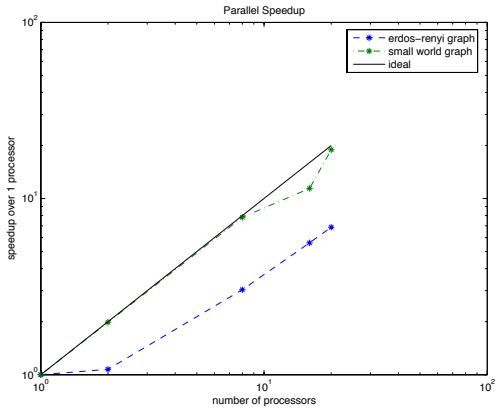


Fig. 13. Execution times of Lanczos Method using Parallel BGL containers. Graphs are on 10^6 vertices and have 12×10^6 edges

no internal structure. For scalability evaluations, we generated undirected graphs from both models with 10^6 vertices and an average vertex degree of 12. We ran the Lanczos method on these graphs for 100 iterations to illustrate scalability objectively between the two graph types, irrespective of their convergence properties in the algorithm. The Lanczos method was included as it is the most powerful method for calculating eigenvalues in IETL. Figure 12 and 13 give parallel speedup and wall clock times, respectively, of our tests. For both random graph models, scalability and performance is quite good, and we achieve near-linear scalability with small-world graphs. The poorer scalability of the Erdös-Renyi graphs is due to the greater number of edges which span processors in the graph distribution; this indicates that graph structure is an important determining factor in the scalability of this eigensolver.

All performance results were collected on a Linux cluster, which consists of 128 compute nodes connected with gigabit ethernet. Each node has two 2GHz AMD Opteron processors, with 4GB RAM, but for our tests we only used one processor per node. The Parallel BGL tests were compiled with Boost 1.33.0 (for the sequential BGL) and the current version of the Parallel BGL. All programs were compiled with version 3.4.4 of the GNU C++ compiler, using optimization level $-$O3 and LAM/MPI 7.1.1.

For means of comparison with non-generic eigensolvers, we ran tests against SLEPc [18]. These preliminary results are not conclusive in regards to overall superiority; for small world graphs, our solution performed much faster than SLEPc but for typical FEM problems, SLEPc was more efficient. On 16 processors, 200 Lanczos iterations on a Laplacian matrix associated with a 500×500 grid required 2.5s with SLEPc, while the same computation with IETL/Parallel BGL took 23.8s. A case study more interesting for our investigation was a small world graph with 2×10^5 vertices and 2.4×10^6 edges where the configuration was the same as the scaling tests above. For the small world graph, 100 Lanczos iterations with SLEPc ran in 65s; with IETL/Parallel BGL the same number of iterations required 9.65s.

As mentioned above, these run time behaviors are subject to further investigations. In any case, our experiments have shown that it is more efficient to use the generic version if only few iterations are desired. This is a result of the expensive copy operations for large distributed data sets, especially in C implementations of matrix assemblies involving many function calls without inlining. In addition, we are working on graph formats with more compression, so that our future implementations will be significantly faster.

VII. Related work

ARPACK [13] is an iterative eigensolver library for both symmetric and non-symmetric matrices; it contains

implementations of the Lanczos tridiagonalization and its generalization for non-symmetric matrices. It is written in FORTRAN77, and offers reverse communication, which allows some choice of user defined containers. This solution differs from ours in that the user must dispatch method calls on user defined containers, and this dispatch happens at run-time. Our solution has the compiler perform all method dispatch, and the dispatch happens at compile time. SLEPc [18] is another parallel library which extends PETSc [5] with eigensolvers.

Lee and Lumsdaine [7] have explored similar techniques for composition of generic iterative solvers written in C++ and various array libraries. They demonstrated how a simple layer of adapters is sufficient to compose a generic iterative solver library with a third-party array library, and study the efficiency of this solution relative to equivalent implementations using conventional libraries.

VIII. Conclusions

There are strong theoretical ties between linear algebra and graph theory that allow algorithms discovered in one domain to be reused in the other. In particular the solution of eigenproblems on graphs has many interesting applications for clustering and connectedness computations. Unfortunately, conventional libraries from these domains cannot be readily composed, even though the structure of the core data types in each domain—adjacency lists and sparse matrices—is quite similar.

The generic programming methodology addresses the problem of intra-domain and cross-domain reusability and library composability by implementing algorithms against *concepts*. Library composition then reduces to mapping from a domain-specific data structure or concept to the concept of yet another domain. We have illustrated this principle by mapping the graphs and property maps of the Parallel Boost Graph Library into the concepts of the Iterative Eigensolver Template Library, effecting an efficient parallel graph eigensolver. With this mapping the whole is far more than the sum of its parts: we have built an efficient graph eigensolver and parallelized it, without the need to explicitly introduce parallelism into the eigensolver.

Although the IETL and Parallel BGL are generic libraries, we have only truly exploited the genericity of the IETL. We could have composed any graph library with the IETL—sequential or parallel—to effect a graph eigensolver. However, the genericity of the Parallel BGL allows further composition: any data structure that can model the Parallel BGL concepts will then model the IETL concepts, thus expanding the capabilities of each library and promoting sharing of ideas and implementations across once-disparate domains.

Acknowledgments

This work was supported in part by NSF grant EIA-0131354 and a grant from the Lilly Endowment.

References

[1] Noga Alon, Michael Krivelevich, and Benny Sudakov. Finding a large hidden clique in a random graph. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 594–598, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[2] J.K. Cullum and R.A. Willoughby. *Lanczos algorithms for Large Symmetric Eigenvalue Computations. Volume 1, Theory*. Birkhäuser, 1985.

[3] Chris H. Q. Ding, Xiaofeng He, and Hongyuan Zha. A spectral method to separate disconnected and nearly-disconnected web graph components. In *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 275–280, New York, NY, USA, 2001. ACM Press.

[4] Douglas Gregor, Nick Edmonds, Brian Barrett, and Andrew Lumsdaine. The Parallel Boost Graph Library. `http://www.osl.iu.edu/research/pbgl`, 2005.

[5] W. D. Gropp and B. Smith. PETSc: Portable extensible tools for scientific computation. Technical report, Argonne National Laboratory, Argonne, IL, 1994.

[6] C. Uhrig K. Mehlhorn, S. Näher. Leda: Library of efficient datatype and algorithms. http://www.mpi-sb.mpg.de/LEDA/, 1998.

[7] Lie-Quan Lee and Andrew Lumsdaine. Generic programming for high performance scientific applications. In *Proceedings of the 2002 Joint ACM Java Grande – ISCOPE Conference*, pages 112–121. ACM Press, 2002.

[8] Lie-Quan Lee, Jeremy Siek, and Andrew Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *ISCOPE'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[9] Bin Luo, Richard C. Wilson, and Edwin R. Hancock. Spectral feature vectors for graph clustering. In *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 83–93, London, UK, 2002. Springer-Verlag.

[10] B. Mohár. *Some applications of Laplace eigenvalues of graphs*. 1997.

[11] David R. Musser and Alexander A. Stepanov. Generic programming. In P. (Patrizia) Gianni, editor, *Symbolic and algebraic computation: ISSAC '88, Rome, Italy, July 4–8, 1988: Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, 1989. Springer Verlag.

[12] Saeko Nomura, Satoshi Oyama, Tetsuo Hayamizu, and Toru Ishida. Analysis and improvement of hits algorithm for detecting web communities. In *SAINT '02: Proceedings of the 2002 Symposium on Applications and the Internet*, pages 132–140, Washington, DC, USA, 2002. IEEE Computer Society.

[13] C. Yang R. B. Lehoucq, D. C. Sorensen. Arnoldi package. `http://www.caam.rice.edu/software/ARPACK/`.

[14] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[15] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, Nov/Dec 1999.

[16] Alexander A. Stepanov and Meng Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

[17] Mathias Troyer and Prakash Dayal. The Iterative Eigensolver Template Library. `http://www.comp-phys.org:16080/software/ietl/`.

[18] Andrés Tomás Vincent Vidal Vincente Hernández, José E. Roman. Scalable library for eigenvalue problem computations. `http://www.grycap.upv.es/slepc/`.