Lifting Sequential Graph Algorithms for Distributed-Memory Parallel Computation

Douglas Gregor
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
dgregor@cs.indiana.edu

Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
lums@osl.iu.edu

ABSTRACT

This paper describes the process used to extend the Boost Graph Library (BGL) for parallel operation with distributed memory. The BGL consists of a rich set of generic graph algorithms and supporting data structures, but it was not originally designed with parallelism in mind. In this paper, we revisit the abstractions comprising the BGL in the context of distributed-memory parallelism, *lifting* away the implicit requirements of sequential execution and a single shared address space. We illustrate our approach by describing the process as applied to one of the core algorithms in the BGL, breadth-first search. The result is a generic algorithm that is unchanged from the sequential algorithm, requiring only the introduction of external (distributed) data structures for parallel execution. More importantly, the generic implementation retains its interface and semantics, such that other distributed algorithms can be built upon it, just as algorithms are layered in the sequential case. By characterizing these extensions as well as the extension process, we develop general principles and patterns for using (and reusing) generic, object-oriented parallel software libraries. We demonstrate that the resulting algorithm implementations are both efficient and scalable with performance results for several algorithms.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Algorithms, Design

Keywords

Generic programming, parallel graph algorithms, distributed computing

1. INTRODUCTION

Significant advances have been made in large-scale computational science and engineering applications because of the availability of high-performance software libraries. Communication libraries like PVM [12] and MPI [25] enabled performance-portable parallel computing. Higher-level libraries (such as PETSc [15], which is built on top of MPI) have similarly enabled the (comparatively) rapid development of complicated applications. Using such libraries allows programmers to focus on the specific needs of their applications rather than the details of numerical linear algebra and parallelism (issues which, for the purposes of many applications, have already been solved).

It is clear, however, that MPI is not the final answer for high-performance parallel programming. Even using MPI, parallel code is still limited by having to express a particular type of parallelism (message passing) in a particular way (MPI library calls). An important principle in software engineering is that of *separation of policy and mechanism.* Ideally, in developing parallel code, we would like to separate the policy of parallelism from the mechanism of parallelism and, if possible, separate the policy of parallelism from algorithms that do not *per se* require parallelism.

As illustrated in libraries such as the Standard Template Library [34], Matrix Template Library [31], and Iterative Template Library [32], the emerging paradigm of generic programming is in some sense based on the idea of separating policy from mechanism. For example, consider the generic conjugate gradient algorithm in the Iterative Template Library:

```
template<class LinearOperator, class HilbertSpaceX,
         class HilbertSpaceB>
int cg(const LinearOperator& A, HilbertSpaceX& x,
       const HilbertSpaceB& b );
```

This function template is parameterized by the type of its arguments for the matrix $\boldsymbol{A}$ as well as vectors $\boldsymbol{x}$ and $\boldsymbol{b}$. For this algorithm to operate correctly, we must be able to apply the linear operator type to a vector type (policy)—we don't specify how that application is accomplished (mechanism). How the application is accomplished is not important to correct functioning of the conjugate gradient algorithm.

For example, the ***cg()*** algorithm above can be used with

Matrix Template Library classes in the following way.

```
typedef  matrix<double, rectangle<>,
                        array< compressed<> >,
                        row_major >::type Matrix;
typedef  dense1D<double> Vector;

Matrix A;          // Create matrix
Vector x, b;       // Create vectors

cg(A, x, b);       // Solve system with CG
```

Because it is polymorphic, the same ***cg()*** algorithm can be used with arbitrary types—provided an object of ***A***'s type can be applied to an object of ***x***'s type. In fact, ***A*** need not be a matrix at all for its application as a linear operator to be valid within this algorithm. So called "matrix-free" algorithms can be implemented with this algorithm by using appropriate types for the arguments [5, 22].

The separation of policy from mechanism has some far-reaching and profound effects for high-performance computing. Because the ***cg()*** algorithm did not depend on the details of the application of ***A*** to ***x***, *the same* ***cg()*** *algorithm can be used to realize a parallel CG solver,* simply by using distributed linear algebra objects [21].

```
ParMatrix A;       // Create parallel   matrix
ParVector x, b;    // Create parallel   vectors

cg(A, x, b);       // Solve parallel   system with CG
```

For completeness we note that it is also necessary that the inner-product and norm functions make sense for the types of ***x*** and ***b***. However, again because policy and mechanism are separated, this is easily handled for the distributed case as well without changing the ***cg()*** algorithm.

That the same algorithm can be reused for sequential and parallel computations may be somewhat surprising but it makes perfect sense in the context of the separation of policy and mechanism. When a generic algorithm is instantiated by applying it to sequential types, the generic operations are realized as sequential ones and we correspondingly realize a sequential algorithm. Similarly, when the algorithm is instantiated by applying it to parallel types, the generic operations become parallel and we realize a parallel algorithm.

To make reusable software libraries more generally applicable in high-performance parallel computing, it is important to understand the relationships between sequential, parallel, and generic versions of algorithms and how to realize parallel algorithms based on generic ones. Correspondingly, it is important to understand how to write generic algorithms so that they can be parallelized and how to write parallel data types with which such algorithms can be composed.

To shed light on these issues, this paper describes the process we used in extending the Boost Graph Library (BGL) [30, 33] (formerly the Generic Graph Component Library [23]) for distributed memory operation. The BGL was designed to be generic, but was not originally designed with parallelism in mind. However, it does include a rich set of carefully factored abstractions to support the problem domain of graph algorithms. To parallelize the BGL, we analyzed these abstractions and removed sequential assumptions in order to realize distributed memory operation of existing BGL algorithms. By characterizing these extensions as well as the extension process, we develop general principles and patterns for using (and reusing) generic high-performance parallel software libraries. We also demonstrate the suitability of the generic programming paradigm for implementing such libraries.

Sections 2 and 3 provide an overview of generic programming and the Boost Graph Library. We present our approach to parallelizing the BGL in Section 4. We demonstrate the efficiency of the resulting Parallel BGL with performance results in Section 5. We review related work in Section 6 and present some conclusions and directions for future work in Section 7.

2. GENERIC PROGRAMMING AND GENERIC LIBRARIES

Generic programming has recently emerged as an important paradigm for the development of highly-reusable software libraries [27]. The term "generic programming" is perhaps somewhat misleading because it is about much more than simply programming *per se*. Fundamentally, generic programming is a systematic approach to classifying entities within a problem domain according to their underlying semantics and behaviors. The attention to semantic analysis leads naturally to the *essential* (i.e., minimal) properties of the components and their interactions. Basing component interface definitions on these minimal requirements provides *maximal* opportunities for re-use.

2.1 Lifting

Concrete implementations are evolved into generic implementations through a process known as *lifting*, in which unnecessary requirements on types are removed from an implementation (the abstraction level of the implementation is "lifted"). Consider the following two implementations. The first computes the sum of doubles stored in an array; the second computes the sum of elements in a linked list.

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

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

Abstractly, both implementations are doing the same thing: traversing a collection of elements and summing up the values. However, the implementations also impose additional requirements (and ones that are unnecessary for the purposes of summation). In the first implementation, the elements must be doubles stored in an array. In the second implementation, the elements must be of type ***node**** with doubles stored in the ***data*** field.

Summing a collection of elements only requires that we be able to visit all of the elements in the collection and extract the corresponding values. A generic algorithm should therefore be able to work correctly with any collection of elements supporting traversal and element access. For instance, one could define a generic implementation of ***sum()*** as:

```
template <typename InputIterator, typename T>
```

```
T sum(InputIterator first, InputIterator last, T s) {
  while (first != last)
    s = s + *first++;
  return s;
}
```

This algorithm is implemented as a function template, parameterized on ***InputIterator*** and ***T***. The algorithm can be used with any type substituted for ***InputIterator***, as long as that type supports the $++$ operation for moving from one element to another and the $*$ operation for accessing a value. Similarly, the type bound to ***T*** must support assignment and addition. Note that although we have specified a particular syntax for these parameterized types (we have to write the algorithm down somehow), we have only specified policy—we have not specified *how* these operations must be carried out. In fact, the ***sum()*** algorithm can be used with arrays, linked lists, or any other type that meets the requirements for ***InputIterator*** and ***T***.

2.2 Concepts

Generic algorithms are defined in terms of properties of types, not in terms of any particular type. We use the term *concept* to mean a collection of abstractions, membership in which is defined by a set of requirements. These requirements are typically syntactic and semantic and are augmented by performance guarantees. A generic algorithm can then be defined as one that works correctly and efficiently for every abstraction in a concept. The ***sum()*** example above is defined in terms of the Input Iterator concept. In the C++ Standard Template Library (STL), generic algorithms for the Input Iterator concept include ***copy()***, ***for_each()***, ***equal()***, ***transform()***, ***accumulate()***, etc.

Some sequence algorithms require their parameters to exhibit additional properties to behave correctly. For example, by adding the requirement of multi-pass traversal to the Input Iterator concept we obtain the STL Forward Iterator concept, which enables algorithms such as STL's ***merge()***, ***fill()***, ***replace()***, ***remove()***, ***rotate()***, etc. The relation between Input Iterator and Forward Iterator is an example of *concept refinement*, and the graph of this refinement relation is known as a conceptual taxonomy. Fewer types will meet the requirements of the Forward Iterator than Input Iterator, but those types can be used in more powerful algorithms. When a data type meets all requirements of a concept, it is said to *model* that concept. When a type models a given concept, it models all concepts that concept refines.

Concepts in generic programming are similar in purpose to interfaces or abstract classes within object-oriented programming. Continuing the analogy, concept refinement is similar to inheritance of interfaces and a class could be said to model any interface that it implements. Thus, our discussion of generic programming and lifting applies equally well to the compile-time polymorphism of C++, the run-time, bounded polymorphism of C# and Java generics, or the use of abstract base classes or interfaces in object-oriented languages without generics.

3. THE BOOST GRAPH LIBRARY

The Boost Graph Library (BGL) is a library of graph algorithms and supporting data structures written in C++. The BGL is part of the Boost C++ libraries [4], a collection of open-source, peer-reviewed C++ libraries that have driven the evolution of library development in the C++ community and ANSI/ISO C++ standard committee. Developed with the generic programming paradigm, one of the primary contributions of the BGL is its generic interface framework for accessing graph data structures and for separating the policy and mechanism of such access. As with other generic libraries, this interface framework is intended to support composition; any graph library that implements the interface can use the generic BGL algorithms and any other algorithms that also use this interface, without loss of efficiency [23]. BGL also provides some general-purpose graph classes that conform to this interface, but they are not meant to be the only such graph classes; different applications may need or want different graph classes.

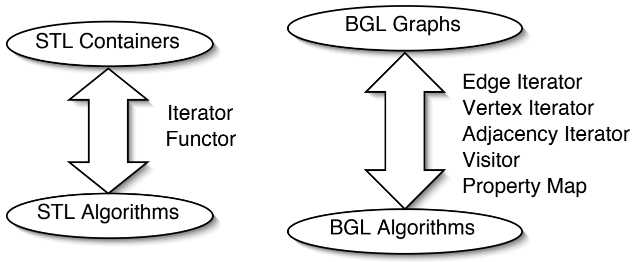


Figure 1: Analogy between generic interface formalisms in the C++ Standard Library and the Boost Graph Library.

3.1 BGL Concepts and the BGL Interface

Although the generic programming paradigm is applicable to the domain of graph algorithms, separating policy from mechanism in the graph domain is more involved than for containers in the C++ Standard Template Library (STL), and a richer interface is required. The analogy between the STL and the BGL is shown in Figure 1.

Like the STL, the BGL uses iterators to define the interface for data structure traversal. There are three distinct graph traversal patterns, each having separate iterators: traversal of vertices in the graph, traversal through the edges, and traversal along the adjacency structure of the graph (from a vertex to each of its neighbors). This generic interface allows functions such as ***breadth_first_search()*** to be usable with a large variety of graph data structures.

To make BGL algorithms extensible, the BGL provides visitors for important graph algorithms. In graph algorithms there are often several key "event points" at which it is useful to insert user-defined operations. The visitor object has a different method that is invoked at each event point. The particular event points and corresponding visitor methods depend on the algorithm. They often include methods like ***start_vertex()***, ***discover_vertex()***, ***examine_edge()***, ***tree_edge()*** and ***edge_relaxed()***.

Typical graph algorithms need to associate values with the vertices and edges of the graph (we call an associated value a *property*). In addition, it is often necessary to associate multiple properties with each vertex and edge. Vertex and edge properties are accessed via a *property map*, of which there is one for each property. Properties may be stored internally to graph data structures or externally. Internal properties are typically supplied to the BGL graph classes via template parameters (e.g., an instance of the ***City*** class in each vertex), whereas external properties are often stored in a vector (e.g., the distance to a vertex from a particu-

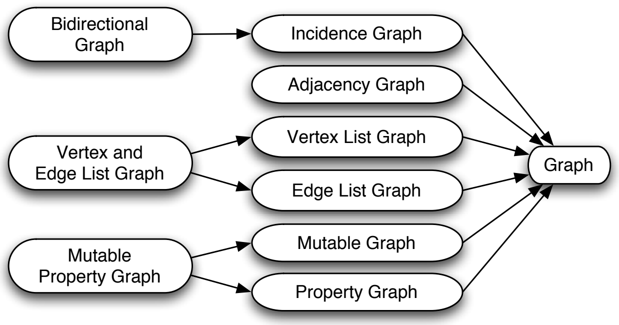


Figure 2: Graph concepts and their refinement relationships in the BGL.

lar source vertex). This is another important separation of policy from mechanism: it is not important to the graph algorithms where or how the properties are stored, the algorithm just needs to be able to access and update the values of those properties efficiently.

To allow generic graph algorithms to be written to minimal requirements, the BGL provides a rich set of graph concepts, as shown in Figure 2. The BGL implements a large set of graph algorithms, including all of the well-known algorithms from graph theory.

3.2 Generic Breadth-First Search

As an illustrative example, Figure 3 shows the generic sequential breadth-first search implementation from the BGL. Note that this implementation is already generic with respect to the type of graph (*g*), type of queue (*Q*), storage of the vertex color (*color*), and visitor (*vis*).

The arguments ***Q***, ***vis***, and ***color*** are optional, provided so that ***breadth_first_search()*** can be customized and form the basis for other algorithms. For example, the relationship between breadth-first search and Dijkstra's algorithm is well known. This relationship is manifest in BGL: Dijkstra's shortest-path algorithm is implemented in the BGL by invoking ***breadth_first_search()*** with appropriate ***vis*** and ***Q*** objects.

3.3 BGL Graph Classes

The BGL provides two graph class templates (***adjacency_list*** and ***adjacency_matrix***) and an edge list adaptor (***edge_list***). The class template ***adjacency_list*** is a highly-parameterized graph class that can be optimized for different situations: the graph can be directed or undirected, allow or disallow parallel edges, can make efficient access to just the out-edges or also to the in-edges, support fast vertex insertion and removal at the cost of extra space overhead, etc. For instance, in the BGL, one might declare an adjacency list type in this manner:

```
typedef adjacency_list<
            /* edge list = */ listS,
            /* vertex list = */ vecS,
            /* directedness = */ bidirectionalS,
            property<vertex_distance_t, double>,
            property<edge_weight_t, double> > Digraph;
```

The parameters to the class template ***adjacency_list*** are (in order):

- The type of storage to be used for the edge lists. ***listS*** indicates the use of a standard ***std::list***. Other options include ***vecS***, ***setS***, and ***hashS***.
- The type of storage to be used for the vertices in the graph. ***vecS*** indicates the use of a standard ***std::vector***, and the options are the same as with edge lists.
- The kind of edges the graph will store. ***bidirectionalS*** indicates that the graph is directed and that all vertices have access to both their outgoing edges and their incoming edges. Other options are ***directedS***, for a directed graph storing only the outgoing edges from each vertex, or ***undirectedS*** for an undirected graph.
- The properties associated with each vertex. In this case, we state that a distance value (a ***double***) will be stored with each vertex. Multiple properties may be placed into vertices, and any property stored in a vertex may be accessed via a property map.
- The properties associated with each edge. As with vertex properties, edges may have properties (such as a weight) associated with them.

4. PARALLELIZING FOR DISTRIBUTED MEMORY

Parallelizing an algorithm for distributed memory involves lifting the requirements of a shared address space and a single thread of execution. For an embarrassingly parallel algorithm, lifting these requirements completely is sufficient, but for the vast majority of algorithms we will have to introduce new requirements needed to ensure proper parallel execution in a distributed memory environment. These requirements often involve addressing (e.g., what data are required to be stored locally), communication (e.g., limits on message frequency, size, and delivery time), and data-race resolution. We describe the abstraction of the Breadth-First Search (BFS) algorithm from the BGL to a distributed BFS implementation, enumerating requirements as we find them. Given these requirements, we present our implementation of the generic, distributed breadth-first search algorithm and detail the abstractions required to ensure correct and efficient execution. The resulting generic algorithm should then be instantiable to a concrete algorithm competitive with existing parallel graph implementations.

4.1 Distributed Breadth-First Search

Breadth-first search traverses a graph from a specified starting vertex and proceeds by first visiting the vertex, then its neighbors, then its neighbors' neighbors, etc. The vertices are therefore organized into *levels*, where each level contains all vertices that are the same number of steps away from the source. Figure 4 illustrates the levels in a breadth-first search starting at vertex s.

A sequential BFS will visit all vertices in a level before proceeding to the next level, but does not specify in what order the vertices within a level will be visited. Taking advantage of this, a parallel BFS visits all vertices within a given level concurrently, but preserves the invariant that no vertex at level l will be visited before a vertex at any level $< l$. A distributed BFS retains the behavior of the parallel BFS, but each processor may only visit the vertices that reside in its memory space. The goal of the lifting process for

```
1  template<class IncidenceGraph, class Buffer, class BFSVisitor, class ColorMap>
2  void
3  breadth_first_search (const IncidenceGraph& g,
4                        typename graph_traits<VertexListGraph>::vertex_descriptor s,
5                        Buffer& Q, BFSVisitor vis, ColorMap color)
6  {
7    put(color, s, Color::gray());                       vis.discover_vertex (s, g);
8    Q.push(s);
9    while (! Q.empty()) {
10     Vertex u = Q.top(); Q.pop();                      vis.examine_vertex(u, g);
11     for (tie (ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
12       Vertex v = target(*ei, g);                      vis.examine_edge(*ei, g);
13       ColorValue v_color = get(color, v);
14       if (v_color == Color::white()) {                vis.tree_edge (*ei, g);
15         put(color, v, Color::gray());                 vis.discover_vertex (v, g);
16         Q.push(v);
17       } else {                                        vis.non_tree_edge (*ei, g);
18         if (v_color == Color::gray())                 vis.gray_target (*ei, g);
19         else                                          vis.black_target (*ei, g);
20       }
21     } // end for
22     put(color, u, Color::black());                    vis.finish_vertex (u, g);
23   } // end while
24 }
```

Figure 3: Generic implementation of the (sequential) breadth-first search algorithm in the BGL. The algorithm resides in the left column and the associated event points are written in the right column.

BFS is to effect the behavior of the distributed BFS using an implementation originally designed for sequential BFS.

The lifting process begins with an existing algorithm and a set of requirements we wish to abstract away, in this case the requirements of a single address space and parallel execution. When lifting the sequential BFS implementation in Figure 3, we consider six sets of related operations: traversing out-edges, retrieving vertices to process, queuing vertices, determining termination, tracking visited vertices, and invoking customization points.

Traversing out-edges. The ***for*** loop on line 11 of Figure 3 iterates over the edges outgoing from the vertex ***u***, which are returned from the ***out_edges()*** function. For each of these edges, we need to be able to retrieve the target vertex of the edge. Within the sequential BGL, these two requirements are encapsulated in the Incidence Graph concept. The Incidence Graph concept also requires that the ***out_edges()*** function operate in constant time, which is essential for the efficient execution of this algorithm. The distributed BFS therefore includes this constant-time requirement and introduces an additional requirement that iteration over the outgoing edges of a vertex stored locally must involve no communication. To permit communication within ***out_edges()*** would exclude certain parallel computation models, such as Bulk-Synchronous Parallel (BSP) [36].

Retrieving vertices to process. The zero-communication requirement on the ***out_edges()*** function restricts the vertices that can be handled by any given processor to its own local vertices. Since all vertices passed to ***out_edges()*** must initially be returned by the queue's ***pop()*** function on line 10, ***pop()*** is required to return only local vertices.

Queuing vertices. Although all vertices ***u*** are required to be local by the ***pop()*** function, the targets of each edge are not required to be local (that would require the entire graph to be local!). Since any vertex (local or remote) can be pushed into the queue at the locations on lines 8 and 16 in Figure 3, but only local vertices may be popped from the queue, the queue must be distributed and remote vertices must be routed to the owning processor. Furthermore, pushing a remote vertex must not involve blocking the sender or transferring a large amount of data, so the ***push()*** operation is limited to sending $\mathcal{O}(1)$ messages with constant size.

Determining termination. The sequential BFS terminates when the queue is completely empty. In the distributed BFS, termination occurs when the distributed queue is empty (i.e., all local queues are empty) and there are no additional ***push()*** operations outstanding. The ***empty()*** operation on line 9 is therefore required to determine when all local queues are empty, making it a synchronization point for the processors.

Tracking visited vertices. The BGL breadth-first search keeps track of the vertices that have been visited by associating a color with each vertex. White vertices have not yet been discovered, gray vertices have been discovered but not yet processed, and black vertices have already been processed. The operations on lines 7, 13, 15, and 22 involve getting and setting the color of a particular vertex. In all cases, the vertex may be either local or remote, and may therefore require communication. The property map ***get()*** operations attempt to retrieve a value associated with a vertex that may have been stored on another processor. These ***get()*** operations may not block (otherwise, the algorithm could deadlock), and therefore must return a "safe" value even for unknown remote vertices: in the case of BFS, this value is ***white***. Note that this behavior introduces the possibility that the same vertex could be pushed onto the queue multiple times by different processors. We therefore revise

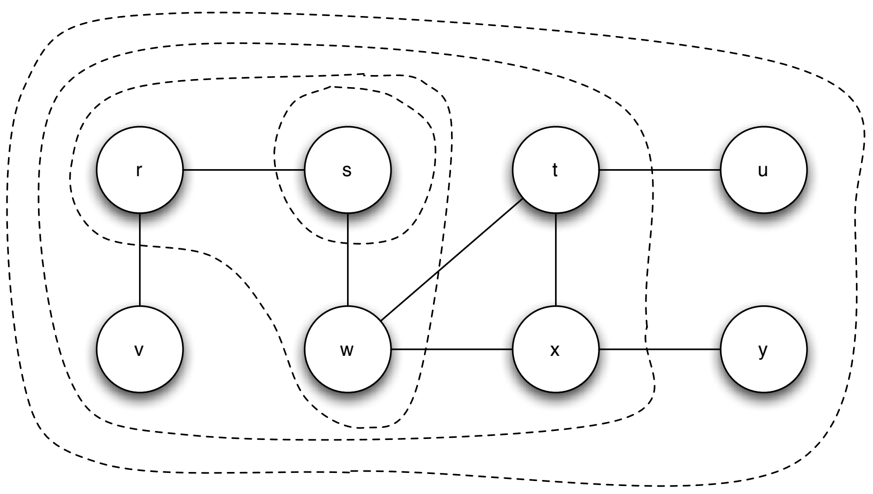


Figure 4: Breadth-First Search levels when starting at the vertex ***s*** and moving outward.

the requirements on the distributed queue to state that it must filter duplicate vertices to ensure that vertices are visited only once. The ***put()*** operations, which also may not block, may require communication, but that communication must be limited to $\mathcal{O}(1)$ messages of constant size to ensure efficient parallel execution. Additionally, several ***put()*** operations may execute concurrently, writing different values for the same vertex. To avoid a data race due to concurrent ***put()*** operations, BFS requires a specific resolution protocol where merging the existing color for a vertex with the color placed by a ***put()*** operation always resolves to the darker color.

Invoking customization points. The BGL breadth-first search is customizable with a visitor object whose members are invoked at certain points in the algorithm (shown in the right-hand column of Figure 3). With the distributed BFS, the visitor must naturally be distributed because certain events (such as ***examine_vertex()*** and ***examine_edge()***) are only executed on a particular processor. In addition, the data races inherent to the distributed property maps cause some events to be unreliable. For instance, many processors may "discover" the same vertex, therefore executing the ***discover_vertex()*** operation several times for the same vertex. Similarly, the classification of tree and non-tree edges is no longer valid, as some non-tree edges may be labeled tree edges, resulting in calls to ***tree_edge()*** when a call to ***non_tree_edge()*** is expected. The visitor must be aware of these potential races to operate correctly in a distributed setting.

The requirements we have derived for the implementation of a distributed breadth-first search involve restrictions on the graph, queue, property map, and visitor, in some cases narrowly defining their behavior. From these requirements we derive our distributed BFS algorithm.

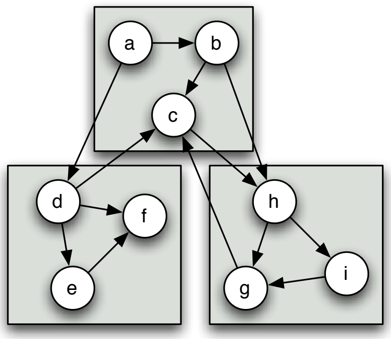
4.2 Distributed BFS Implementation

The declaration of ***breadth_first_search()*** shown in Figure 3 permits customization of the graph ***g***, buffer (queue) ***Q***, color property map ***color***, and visitor ***vis***. Abstracting each of these parameters has proven valuable within the sequential BGL, but they serve a dual purpose because the requirements we have derived while lifting to the distributed BFS all apply to these parameters. *Thus, the "implementation" of distributed breadth-first search in the Parallel BGL is merely a call to the sequential BFS using a distributed graph, queue, visitor, and color map.* Implementing a generic, distributed BFS therefore reduces to implementing these distributed data structures, paying careful attention to the concepts derived from the generic BFS implementation. The following data structures are implemented in the Parallel BGL to support distributed graph algorithms such as BFS and Dijkstra's shortest paths. Note, however, that the implementations provided are not the only implementations possible: any implementation meeting the requirements stated by the Parallel BGL concepts would suffice.

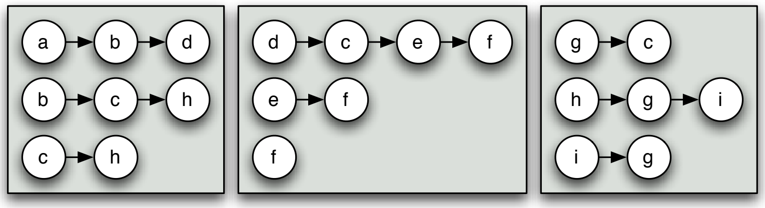
Distributed graph. The Parallel BGL provides a distributed adjacency list class template that meets the requirements of distributed BFS. Each vertex in the graph is owned by a single process, and all edges outgoing from that vertex and properties associated with that vertex are stored with the vertex itself. Figure 5 illustrates the distribution of a directed adjacency list across three processors.

The distributed adjacency list itself is built upon the non-distributed adjacency list already a part of the Boost Graph Library, and almost completely mimics its interface. In the distributed adjacency list, the only change is in the storage of the vertices. The Parallel BGL therefore retains the interface but introduces an additional, parameterized selector ***distributedS*** that indicates that vertices should be distributed. For instance, the following type is a distributed form of the non-distributed adjacency list previously described:

```
typedef adjacency_list<
          /* edge list = */ listS,
          /* vertex list = */
            distributedS<mpi::bsp_process_group,vecS>,
          /* directedness = */ bidirectionalS,
          property<vertex_distance_t, double>,
          property<edge_weight_t, double> >
  DistributedDigraph;
```



(a) Distributed graph



(b) Distributed adjacency list representation

Figure 5: A distributed directed graph represented as an adjacency list across three processors.

The ***distributedS*** selector takes two parameters. The first parameter is a Process Group that defines the medium over which the vertices will be distributed and is described later in this paper. The second parameter is a vertex list selector that determines how the list of vertices is stored on each processor.

The distributed adjacency list supports many of the same operations as the non-distributed adjacency list, and is implemented as a distributed wrapper over a non-distributed adjacency list (one on each processor, storing the graph for local edges). Thus, as the non-distributed adjacency list grows (for instance, by supporting additional data structures), the distributed adjacency list also gains these features.

Whenever a new data structure is introduced for use with a generic library such as the (Parallel) BGL, it is important to determine which of the existing concepts the new data structure will model. For instance, which of the graph concepts in Figure 2 does the distributed adjacency list model? Although a complete diagnosis is beyond the scope of this paper, several interesting concepts require consideration:

- Graph: The Graph concept requires the data type to provide the types of its vertices and edges. The distributed adjacency list can trivially model this concept.

- Incidence Graph: The distributed adjacency list implements the ***out_edges()*** function to provide the outgoing edges for a given vertex, as required by this concept. However, there is one caveat: the distributed ***out_edges()*** can only be applied to vertices local to the executing processor. Thus, we may state that the distributed adjacency list models the Incidence Graph concept subject to a restriction on the domain of the ***out_edges()*** function.

- Vertex List Graph: A model of Vertex List Graph provides a function ***vertices()*** that enumerates all vertices in the graph. However, with a distributed adjacency list, no single processor stores a list of all of the vertices in the graph. Therefore, the distributed adjacency list does not model the Vertex List Graph concept. However, one can define a slightly more general concept that the distributed adjacency list does model. This concept, named Distributed Vertex List Graph in the Parallel BGL, requires that the function ***vertices()*** return a set of vertices on each processor, such that the union of the sets from all processors is the complete set of vertices. Note that the Vertex List Graph concept is now a refinement of the Distributed Vertex List Graph concept.

Distributed queue. The distributed queue in the Parallel BGL is implemented as an adaptor over a sequential queue that stores values in the local queue. The ***pop()*** operation pulls from this local queue, while the ***push()*** operation sends a message containing the vertex to the vertex's owner. Most of the implementation effort is in the ***empty()*** routine, which must repeatedly exhaust the local queue and synchronize with all other processors to determine if the algorithm should terminate. Note that messages are only received during synchronization, so that messages sent at a particular level in the BFS tree will not be received until all processors have finished handling vertices at that level. This formulation of the distributed queue adaptor therefore implements a level-synchronized breadth-first search.

Distributed visitor. The distributed visitor is the caller's responsibility, and therefore requires no changes in the of the Parallel BGL implementation. Figure 6 contains a breadth-first search visitor that computes the distance from the source node to every other node along a breadth-first traversal. Since the source of a tree edge is always local to the process executing ***tree_edge()***, the source distance is correct and the target distance can be updated by sending a message to its owner. Thus, this BFS visitor is correct for both sequential and distributed BFS, requiring only that the property map ***distance*** be distributed in the distributed case and non-distributed in the sequential case.

Distributed property map. The distributed property map is the most interesting component of the Parallel BGL. It is implemented as an adaptor consisting of four parts:

1. Local property map

```
template<class DistanceMap>
struct bfs_discovery_visitor : bfs_visitor<>
{
  bfs_discovery_visitor(DistanceMap distance)
    : distance(distance) { }

  template<class Edge, class Graph>
  void tree_edge(Edge e, const Graph& g)
  {
    std::size_t new_distance =
      get(distance, source(e, g)) + 1;
    put(distance, target(e, g), new_distance);
  }

 private:
  DistanceMap distance;
};

// Constructor function
template<class DistanceMap>
inline bfs_discovery_visitor<DistanceMap>
make_bfs_discovery_visitor(DistanceMap distance)
{
  return bfs_discovery_visitor<DistanceMap>(distance);
}
```

Figure 6: BFS visitor that calculates the distance from the source node to every other node and stores the result in the property map ***distance***.

2. Ghost cells
3. Process group
4. Data race resolver

The local property map stores the values associated with local entities (vertices or edges), and is equivalent to a property map for a non-distributed BGL graph. For values associated with remote entities, the distributed property map maintains a set of ghost cells that are automatically created for any ***put()*** or ***get()*** request and store the most recent value known to this processor. The process group is the communication medium, which is used by the distributed property map to broadcast ***put()*** requests and receive updated values into ghost cells. Finally, the data race resolver provides the default value for an entity when the first remote ***get()*** operation is invoked and decides among various ***put()*** messages sent to the distributed property map.

The distributed property map adaptor is used in many places throughout the Parallel BGL, although many of its uses are transparent to the user. Properties stored within distributed graphs are accessed via automatically-generated distributed property maps. Additionally, we have provided specializations and overloads for several of the common sequential *external* property maps that silently generate distributed property maps when a distributed graph is involved. These specializations and overloads are the C++ compile-time equivalent of an abstract factory, producing data types that model particular concepts and are well-suited to the execution environment (sequential or distributed), but are otherwise unspecified. For instance, the following call to BFS will generate a distributed ***iterator_property_map*** when ***g*** is distributed a non-distributed ***iterator_property_map*** when ***g*** is not distributed:

```
std::vector<int> distances(num_vertices(g));

vertex_descriptor s = vertex(A, g);

breadth_first_search
  (g, s,
   visitor(make_bfs_discovery_visitor
             (make_iterator_property_map
                (distances.begin(),
                 get(vertex_index, g)))));
```

The preceding call to ***breadth_first_search()*** is interesting because it need not change when the type of ***g*** changes. The algorithms only require ***g*** to model the Incidence Graph concept and the visitor to model the BFS Visitor concept. Since the ***breadth_first_search()*** implementation uses factories to provide default values for the other parameters—the queue and color property map—the algorithm operates on distributed data structures when ***g*** is distributed, therefore realizing a distributed algorithm. When ***g*** is not distributed, sequential data structures are employed and a sequential algorithm is realized. This functionality is entirely encapsulated in the abstract factories themselves, so neither user code nor the BFS implementation need be modified.

4.3 Process Groups

Reusing the sequential BFS implementation relies on communication through various distributed data structures supplied to the algorithm, such as the distributed queue and distributed property map. The set of communicating data structures for a given algorithm invocation is not fixed: for instance, a user may introduce additional distributed property maps via the visitor, e.g., to compute the level in the BFS tree. Process groups tie together these disparate data structures within a single, underlying communication model that synchronizes the data structures concurrently.

Like Incidence Graph and BFS Visitor in the Boost Graph Library, Process Group is a concept that specifies the behavior of the communication layer but not its precise type or implementation. Figure 7 illustrates the process group concept taxonomy, containing both shared-memory and distributed-memory process group concepts. The best understood process group concept within the Parallel BGL is the Messaging Process Group, which requires several operations for a process group ***pg***:

1. ***construct(other, handler)***: Constructs a new process group object related to the existing ***other*** process group and install the given handler. Each data structure installs its own handler in the process group, which gives it a separate block of message tags.

2. ***send(pg, dest, tag, value)***: Send the given ***value*** in a message marked with the given numerical ***tag*** to the process with identifier ***dest***. Messages with a given ***(source, dest)*** pair are guaranteed to be received in the order sent.

3. ***receive(pg, source, tag, value)***: Receive a message containing ***value*** from process ***source*** with the given ***tag***. The ***receive()*** operation blocks until such a message is available.

4. ***probe(pg)***: Immediately returns a ***(source, tag)*** pair if a message is available, or a no-message indicator.

5. ***synchronize(pg)***: Collectively waits until all messages sent by any process are stored in a buffer at their destinations. All messages sent prior to synchronization may be immediately received after synchronization.

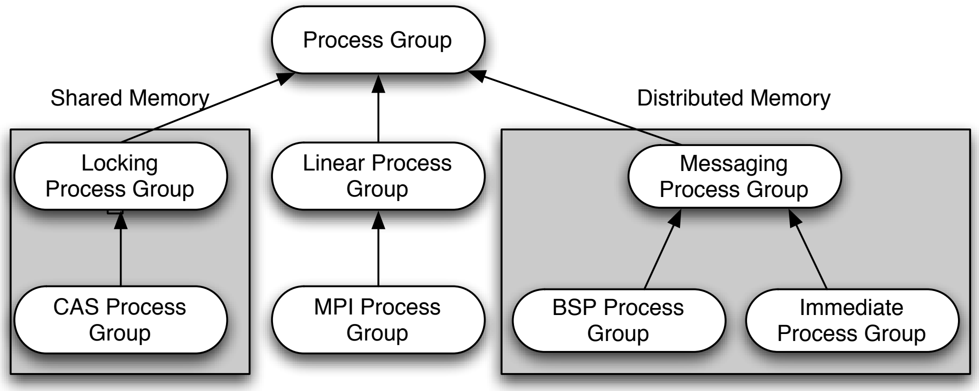


Figure 7: Breadth-First Search levels when starting at the vertex *s* and moving outward.

The Messaging Process Group requirements are intended to provide a suitable abstraction of message-based communication against which distributed algorithms and data structures can be written without overly constraining the implementation. There is sufficient slack within the requirements to permit different memory consistency models to fit within the framework and, in many cases, use the same generic algorithms. For instance, the ***synchronize()*** operation guarantees only that any messages sent before it is invoked can be received after it returns, but does not specify whether messages may be received before the synchronization call. The two concepts that refine Messaging Process Group, Immediate Process Group and BSP Process Group, provide stronger guarantees. The former guarantees that a message will be transmitted immediately and may be received prior to the call to ***synchronize()***. The latter, however, guarantees that messages sent before the synchronization will not be received until after synchronization, i.e., the ***synchronize()*** call in effect delineates supersteps in a BSP algorithm. The Parallel BGL contains two process group implementations based on MPI, one modeling the Immediate Process Group concept and the other modeling BSP Process Group. With a few exceptions, either process group can be used with any algorithm in the Parallel BGL.

4.4 Distributing Dijkstra's Algorithm

Dijkstra's algorithm computes shortest paths from a given source vertex. The algorithm maintains a set of vertices whose shortest distances have already been found, and in each step expands this set to include a new vertex that is closest to the source. Dijkstra's algorithm can therefore be implemented as a breadth-first search using a priority queue to order the vertices by distance and performing a *relax* operation to update the distance to vertices in the queue when an edge is crossed. Within the sequential BGL, Dijkstra's algorithm is implemented as a call to ***breadth_first_search()*** that provides a priority queue and a special Dijkstra visitor to implement the required functionality.

Parallelizing Dijkstra's algorithm is similar to parallelizing breadth-first search: the priority queue is distributed into priority queues on each processor, with ***push()*** operations passing messages to the owner of the vertex. Following the naïve formulation of parallel Dijkstra's algorithm [14, §10.7.2], after each level of the tree is processed an all-reduce operation is performed to determine the (global) minimum distance to any vertex in the graph (this can be performed by the distributed queue's ***empty()*** operation), and at each step the processors may only consider vertices whose distance is equal to that minimum. Parallelism occurs when the distance from the source to two or more vertices is equivalent, and these vertices are stored on different processors so that they will be considered concurrently. Since the implementation of distributed Dijkstra's algorithm can be effected given a distributed priority queue and a special BFS visitor, it can be implemented trivially as a call to the generic (sequential or distributed) ***breadth_first_search()***. Moreover, the BFS visitor implemented by Dijkstra's algorithm in the sequential BGL is itself generic, and can be reused for both sequential and distributed Dijkstra's algorithm. Thus, *the implementation of Distributed Dijkstra's algorithm requires only a distributed priority queue.*

Increasing the number of vertices at each level in the graph is the key to extracting more parallelism from this formulation of parallel Dijkstra's algorithm. Various heuristics exist to identify the set of vertices to process in each step [9, 26], indicating that several distinct implementations of the algorithm may be required. However, this is not the case: the heuristics can be completely encapsulated in the distributed priority queue, therefore permitting the variations on Dijkstra's algorithm to be implemented as an invocation of ***breadth_first_search()*** with a customized visitor and queue. For sequential graphs, the queue is merely a relaxed heap, whereas for distributed graphs the queue is one of several different distributed priority queues. As with Breadth-First Search, the generic implementation of Dijkstra's algorithm within the (sequential) BGL has been lifted for distributed-memory computation without modifying the algorithm itself.

4.5 Distributed Depth-First Search

The generic breadth-first search implementation could be abstracted for distributed-memory parallelism, producing an efficient algorithm that also serves as the basis for an efficient, distributed implementation of Dijkstra's algorithm. Figure 8 illustrates the BGL implementation of another core graph algorithm: depth-first search.

```
1   template <class IncidenceGraph, class DFSVisitor, class ColorMap>
2   void depth_first_search
3     (const IncidenceGraph& g,
4      typename graph_traits<IncidenceGraph>::vertex_descriptor u,
5      DFSVisitor vis,   ColorMap color)
6   {
7     put(color , u, Color:: gray());           vis . discover_vertex (s, g);
8
9     for (tie (ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
10      Vertex v = target(*ei, g);              vis . examine_edge(*ei, g);
11      ColorValue v_color = get(color , v);
12      if (v_color == Color::white()) {        vis . tree_edge (*ei, g);
13        depth_first_search (g, v, vis , color );
14      } else if (v_color == Color::gray())   vis . back_edge(*ei, g);
15      else                                    vis . forward_or_cross_edge (*ei, g);
16    }
17    put(color , u, Color:: black ());         vis . finish_vertex (u, g);
18  }
```

Figure 8: Generic implementation of the recursive, sequential depth-first search algorithm in the BGL. The algorithm resides in the left column and the associated event points are written in the right column.

Analysis of the DFS implementation reveals several constructs that are similar to the BFS implementation, such as property map accesses (lines 7, 11, and 17) and iteration over the out-edges of a vertex (line 9), which can be externally parallelized. However, the recursive call at line 13 presents a problem, because the vertex ***v*** may be non-local. One potential solution would be to make the call to ***depth_first_search()*** a remote procedure call, which is automatically routed to the owner of the vertex ***v***. While this does make the algorithm applicable in a distributed context, it does not expose any parallelism: the algorithm will still proceed as if it were sequential.

An alternative solution involves recasting the DFS implementation as an iterative algorithm by storing the vertices on a stack. The resulting algorithm (which serves as the default DFS implementation in the BGL) is similar to BFS. Would it then be possible to implement distributed depth-first search by distributing the stack? It is possible, but again this result does not parallelize well: no processor can pop the top vertex from the distributed stack until it is certain that no more vertices will be pushed onto the stack.

Neither formulation of depth-first search can be lifted into an efficient distributed algorithm. However, by applying the lifting process to these algorithms one is able to rapidly determine that the resulting parallel algorithms would not be efficient.

5. EXPERIMENTAL RESULTS

No technique for producing parallel or distributed algorithms from sequential ones is useful unless the resulting parallel or distributed algorithm is both efficient and scalable. To determine if the lifting approach is in fact useful, we evaluated the performance of the Parallel BGL on random graphs generated with three models. The results demonstrate that the Parallel BGL achieves good performance and scalability.

5.1 Random graph models

We have selected three very different graph models: Erdös-Renyi, small-world, and scale-free. Each exhibits a unique structure that impacts the performance of graph algorithms.

The Erdös-Renyi model is the oldest model of graphs. A $G(n, p)$ Erdös-Renyi graph has n vertices and a probability p that an edge exists between any two vertices. In this model, vertex degrees tend to fall on a normal distribution centered around pn, the graph tends to be connected when $p \geq \log n/n$ and has a small diameter of 2 when $p \geq 2/\sqrt{n}$. However, the clustering coefficient of an Erdös-Renyi graph tends to be very low, as there is very little internal structure.

Watts' small world model constructs graphs that have a large clustering coefficient and small diameter, as is common with various real-world social networks. A $G(n, k, p)$ small-world graph has n vertices (typically arranged in a ring) with each vertex connected to its k nearest neighbors in the ring. With probability p, each edge may be "rewired" to another vertex randomly selected from outside its closest k neighbors. The degree distribution is a normal distribution centered on k with a small standard deviation unless p is large.

Scale-free graphs exhibit a power-law degree distribution, which occurs in many real-world networks (e.g., web site connectivity and actor collaboration graphs). These graphs tend to have a small diameter, because a vertex can reach many other vertices through the high-degree "hubs". To generate scale-free graphs we use the Power-Law Out Degree (PLOD) algorithm [29], which generates scale-free graphs that generally exhibit low clustering.

5.2 Experimental setup

We demonstrate the performance and scalability of the implementation of the two algorithms successfully lifted in this paper, breadth-first search and Dijkstra's algorithm, along with a third common graph algorithm: connected components. Additional performance results for these and other algorithms in the Parallel BGL [3, 9, 11, 13, 35] (see complete list in Figure 9) are provided at `http://www.osl.iu.edu/research/pbgl/performance`.

We performed our performance evaluation on Odin, which consists of 128 compute nodes connected via Infiniband. Each node contains two 2GHz AMD Opteron processors with 4GB RAM, but for our tests we have left one processor idle on each node. The Parallel BGL tests were compiled using Boost 1.32.0 [4] (containing the sequential BGL) and the current development version of the Parallel BGL.

Breadth-first Search	Depth-first Search
Crauser et al's shortest paths	Eager Dijkstra's shortest paths
Boruvka Minimum Spanning Tree	Dehne & Götz's Minimum Spanning Tree
Connected Components	Strongly Connected Components
Page Rank	Boman et al's Graph Coloring

Figure 9: Algorithms implemented in the Parallel BGL

All programs were compiled with version 3.4.2 of the GNU C++ compiler using optimization level $-\boldsymbol{O2}$ and LAM/MPI 7.1.2b17.

5.3 BFS performance

For the breadth-first search tests, we performed a breadth-first search from node zero (arbitrarily selected) and supplied a visitor that computes the distance from the source node to each node (see Figure 6), thus requiring an additional distributed property map.

Figure 10 illustrates the performance of distributed breadth-first search for a fixed problem size with a relatively dense graph containing $n = 100,000$ vertices and $m = 15,000,000$ edges. The small-world graphs used a rewriting probability of $p = 0.04$ and the scale-free graphs used a constant falloff of $\alpha = 0.5$. The graphs were small enough to avoid swapping on the single-processor tests, and we see good scalability up to 64 processors (on 64 nodes). As expected, the performance on small-world graphs is better than for scale-free or Erdös-Renyi graphs due to better load balancing.

Figure 11 illustrates the performance of distributed BFS on more sparse graphs, with $n = 1,000,000$ vertices and $m = 15,000,000$ edges. Here we see a greater disparity between small-world performance and the performance on the other two graph types. Scalability is still reasonable in all cases, but the small-world graphs are exhibiting optimal scalability.

5.4 Dijkstra performance

For the Dijkstra's single-source shortest paths tests, we attached edge weights chosen from a normal distribution over $[0, 1)$ and computed the shortest paths to every vertex from an arbitrary start vertex. The random graphs generated for these experiments are identical to the random graphs employed in the evaluation of breadth-first search. We present results using two different Dijkstra heuristics. The first heuristic permits the algorithm to process all vertices whose distances are at most a constant factor h greater than the global minimum distance. For these experiments, we let $h = 0.1$. The second heuristic is the adaptive heuristic of Crauser et al. [9], which computes the set of vertices that can be processed based on the weights of the incoming and outgoing edges for each vertex.

Figure 13 illustrates the performance of the two Dijkstra heuristics—(C) for Crauser et al. and (E) for the "Eager", constant-lookahead version–on relatively sparse graphs, with an average vertex degree of 30. We note that the two heuristics produce very different results depending on the underlying structure of the graph. Figure 12 demonstrates similar results on a much denser graph (average vertex degree of 300), although the differences between the two heuristics are more pronounced. These results lead us to two interesting conclusions. First, we can achieve good scalability and performance from non-trivial distributed algorithms implemented by lifting generic, sequential algorithms. Second, by providing the implementation of distributed Dijkstra's algorithm as a generic algorithm, we permit the user to select from a predefined set of heuristics or supply new heuristics, just by replacing the distributed priority queue. Since the performance characteristics of the two heuristics are so radically different, this customization is essential for a software library.

5.5 Connected Components performance

The Parallel BGL implements a variation of the connected components algorithm by Goddard, Kumar, and Prins [13] using some optimizations from Johnson and Metaxas [19] and adapted for distributed memory. We have opted to provide performance results comparing our connected components implementation to that of CGM*graph* [7]. CGM*graph* is an object-oriented parallel graph library written in C++ and based on the Course-Grained Multicomputer (CGM) [10] model of computation implemented via MPI.

Figure 14 illustrates the performance of the Parallel BGL relative to CGM*graph*, using Erdös-Renyi graphs with 96,000 vertices and 10 million edges[1]. The CGM*graph* results shown in the left-hand plot exhibit the same trends as the results presented in [7], although our cluster is slightly faster. On the same graphs, the Parallel BGL exhibits the same scalability but performs significantly faster. The right-hand plot shows the speedup of the Parallel BGL over CGM*graph*, which varies between 17 and 30.

6. RELATED WORK

The idea of extracting a parallel or distributed algorithm from an efficient sequential algorithm is not new. Parallel algorithm designers have long drawn inspiration from existing sequential algorithms, often considering several algorithms before selecting a particular one to parallelize. However, the emphasis was on creating *new* parallel or distributed algorithms, whereas our emphasis is on abstracting the fundamental underlying algorithm to produce a single generic version that encompasses both sequential and distributed computation. Lee et al. [21] have found that the generic algorithms in the Iterative Template Library can instantiate to either parallel or sequential algorithms, using distributed vectors and matrices for distribution. Jézéquel [18] describes the use of Eiffel inheritance to build distributed algorithms and data structures from their sequential counterparts within the Eiffel Parallel Execution Environment (EPEE) [17], focusing on two core algorithms: a ***map()*** operation to transform one sequence into another and a ***reduce()*** operation to compute a single result from a sequence of values.

Nitsche describes a transformation from sequential functional programs to parallel "skeletons" [28], i.e., a primitive set of high-level parallel routines based on message-passing.

[1]These graphs are equivalent to the largest graphs reported by Chan and Dehne [7].

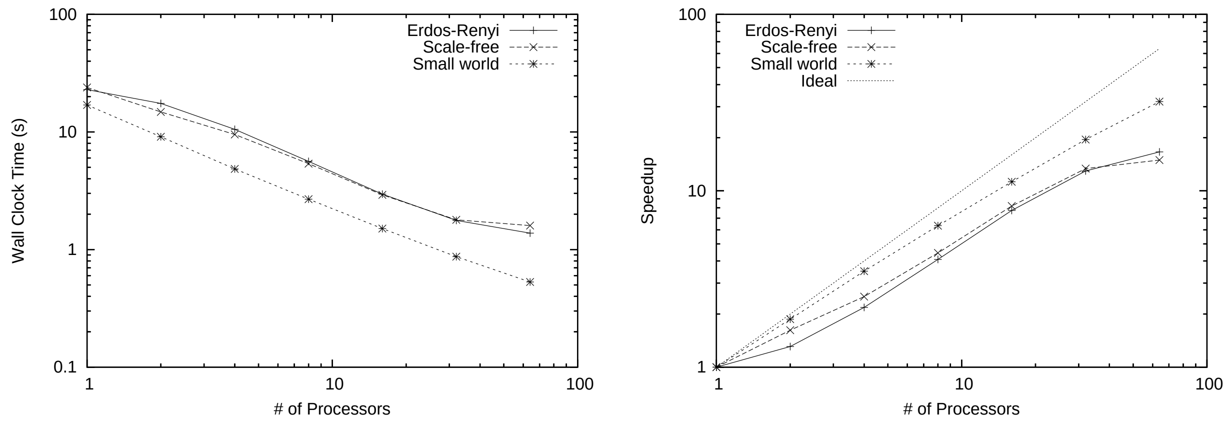


Figure 10: Performance of distributed breadth-first search on random graphs with $100k$ vertices and $\approx 15M$ edges

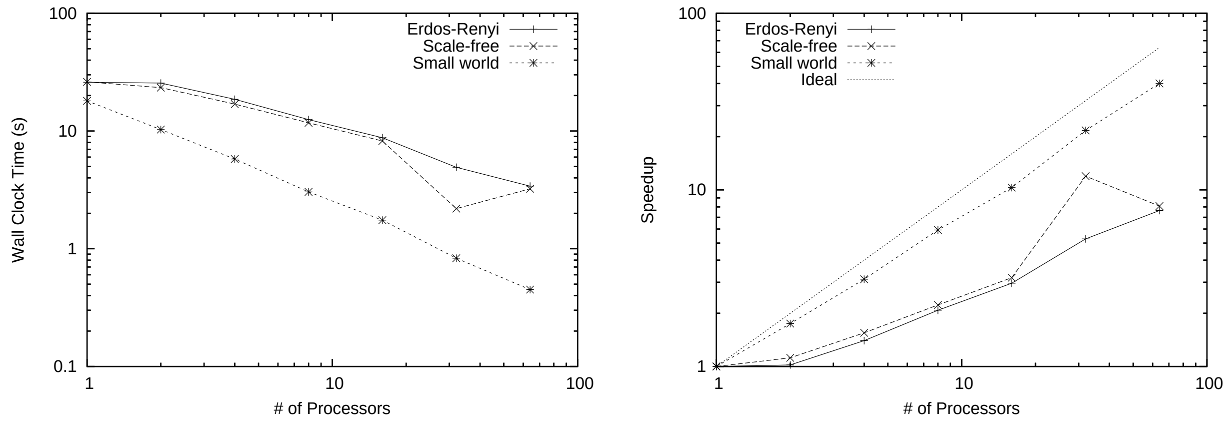


Figure 11: Performance of distributed breadth-first search on random graphs with $1M$ vertices and $\approx 15M$ edges

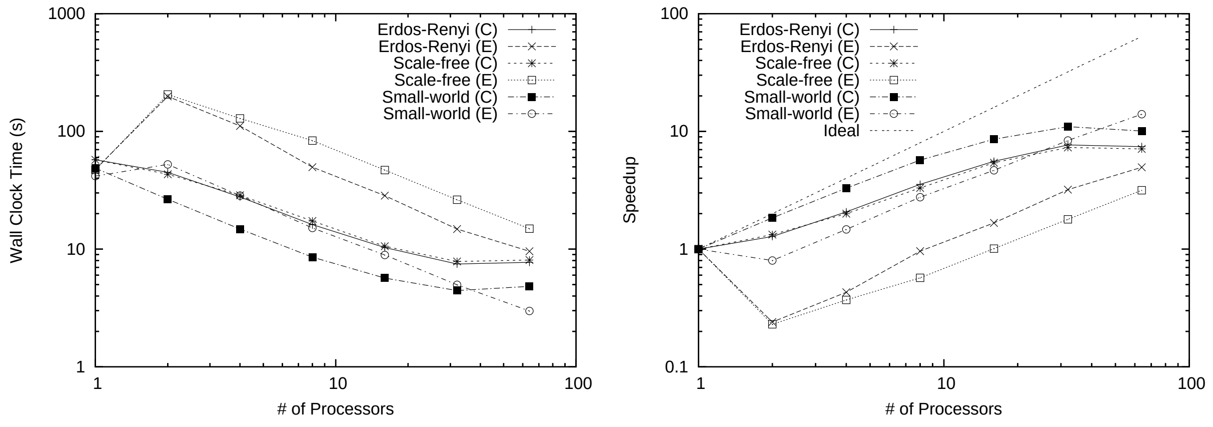
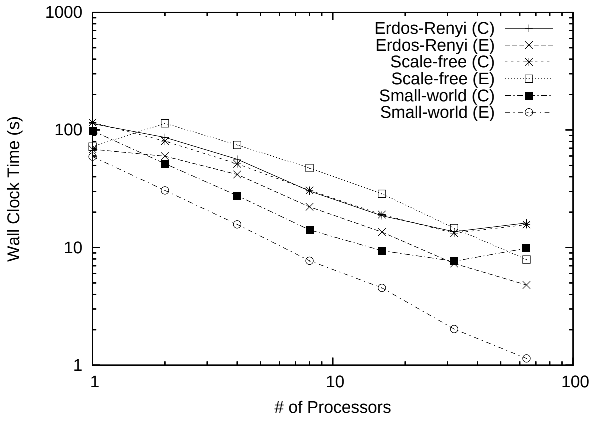


Figure 12: Performance of various distributed Dijkstra formulations on random graphs with $100k$ vertices and $\approx 15M$ edges.



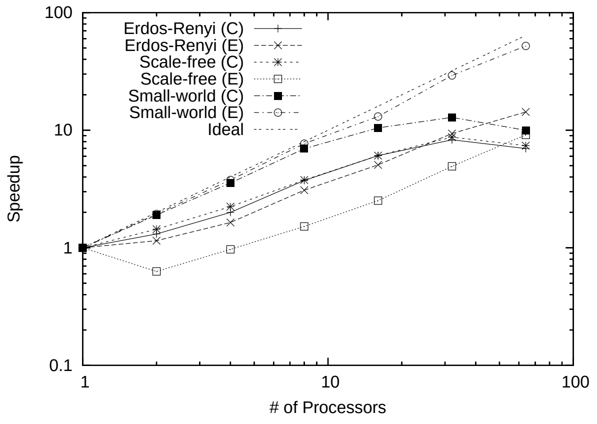
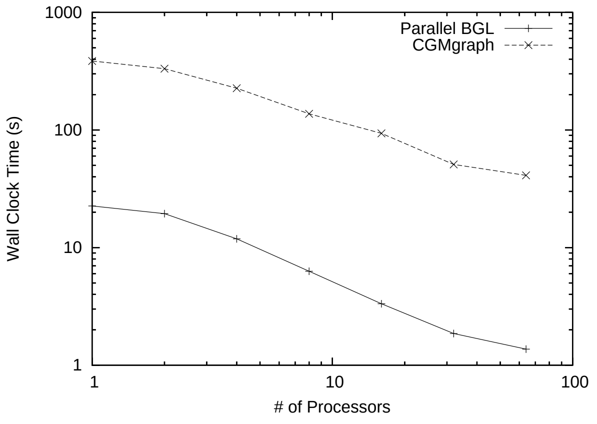


Figure 13: Performance of various distributed Dijkstra formulations on random graphs with $1M$ vertices and $\approx 15M$ edges.



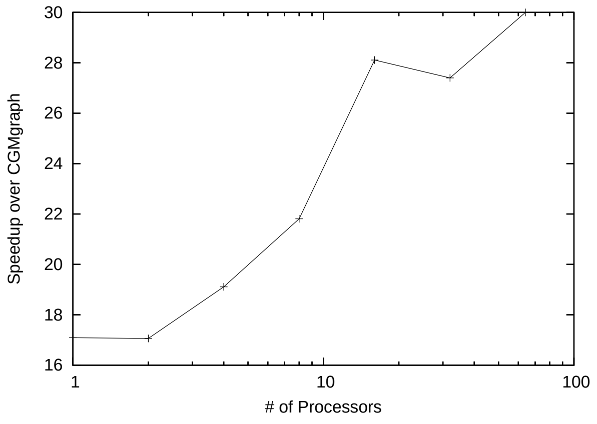


Figure 14: Performance of distributed connected components for the Parallel BGL and CGM*graph*.

The transformation requires a description of the topology of the data structure, called a "cover." As with the manual translations from parallel to distributed algorithms, this approach is again a translation that produces explicitly parallel algorithms, not generic algorithms that can be sequential or parallel.

Languages and libraries that support data-parallel programming [1, 2, 6, 20, 24] permit the implementation of algorithms that will be efficient in a sequential, parallel, or distributed setting. In many ways the algorithm implementations produced by lifting away the sequential execution and single address space requirements will resemble the algorithms as written in these languages. One could apply the lifting process to a sequential algorithm and then use the data-parallel constructs of these libraries or languages in the resulting implementation, *if* the algorithm mandates the proper structure and the requirements of the algorithms match well with the facilities provided by the library or language.

There are several parallel libraries providing graph algorithms and data structures. The ParGraph library [16], also built on top of the sequential BGL, provides generic max-flow and breadth-first search implementations, although it opts for a more explicit representation of communication that does not permit reuse of, e.g., breadth-first search in the parallel context. The CGM*graph* library [7, 8] is an object-oriented library for distributed graph computation. The Standard Template Adaptive Parallel Library (STAPL) [1] is a generic, parallel library modeled after the Standard Template Library and providing distributed data structures and parallel algorithms, including a graph data type. Where the Parallel BGL has opted to encode distribution and parallel communication information within the data types and selects efficient algorithms at compile time, STAPL delays such decisions until run-time to adapt to the current execution environment.

7. CONCLUSION AND FUTURE WORK

The generic programming process emphasizes lifting away unnecessary requirements to produce maximally reusable algorithms without sacrificing performance. While generic programming has traditionally focused on lifting sequential algorithms operating in a single address space, we have applied the approach to lift sequential algorithms for parallel execution with distributed memory. We have shown this abstraction process to be successful for two graph algorithms, breadth-first search and Dijkstra's single-source shortest paths, which were lifted to distributed algorithms via externally-supplied distributed data structures but exhibited good performance and scalability. We have also shown that not all generic algorithms can be parallelized in this manner: attempting to do so with depth-first search immediately uncovered performance problems with the resulting distributed algorithm.

Large scale distributed applications are more difficult to develop than sequential applications because of the additional requirements of communication and coordination. To make development of such applications manageable while also achieving high levels of performance, it is important to leverage high-quality sequential libraries—and important for sequential libraries not to impose unnecessary requirements. Generic programming is a paradigm that allows this to be done in a systematic and high-performance way that is implementable in mainstream object-oriented programming languages such as C++, Java, and C#.

Our research with the Parallel BGL is progressing in several directions. We are applying the generic lifting process to additional algorithms from the (sequential) BGL to produce distributed, generic graph algorithms and experimenting with the lifting process for extracting fine-grained parallelism. The most interesting results may come from combining fine- and course-grained parallelism in generic graph algorithms.

The Parallel BGL is open source and available under a BSD-like license at `http://www.osl.iu.edu/research/pbgl`.

Acknowledgments

This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment. The Odin cluster used in the experiments was funded, in part, by NSF Grant EIA-0202048. The authors also thank Brian Barrett for his help with performance analysis and Chris Mueller and Ronald Garcia for valuable suggestions.

8. REFERENCES

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. In *Int. Wkshp on Adv. Compiler Technology for High Perf. and Embedded Processors*, page 10, July 2001.

[2] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-93-129, April 1993.

[3] Erik G. Boman, Doruk Bozdag, Umit Catalyurek, Assefaw H. Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. Preprint.

[4] Boost. *Boost C++ Libraries*. `http://www.boost.org/`.

[5] Peter N. Brown and Alan C. Hindmarsh. Matrix-free methods for stiff systems of ODE's. *SIAM J. Numer. Anal.*, 23(3):610–638, June 1986.

[6] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *Software Engineering*, 26(3):197–211, 2000.

[7] Albert Chan and Frank Dehne. CGM*graph*/CGM*lib*: Implementing and testing CGM graph algorithms on PC clusters. In *PVM/MPI*, pages 117–125, 2003.

[8] Albert Chan and Frank Dehne. cgmLIB: A library for coarse-grained parallel computing. `http://lib.cgmlab.org/`, 2004 December.

[9] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In Lubos Brim, Jozef Gruska, and Jirí Zlatuska, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[10] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of*

the ninth annual symposium on Computational geometry, pages 298–307. ACM Press, 1993.

[11] Frank Dehne and Silvia Götz. Practical parallel algorithms for minimum spanning trees. In *Symposium on Reliable Distributed Systems*, pages 366–371, 1998.

[12] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: A Parallel Virtual Machine*. Scientific and Engineering Computation Series. MIT Press, 1994.

[13] Steve Goddard, Subodh Kumar, and Jan F. Prins. Connected components algorithms for mesh connected parallel computers. In Sandeep N. Bhatt, editor, *Parallel Algorithms*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–58. American Mathematical Society, 1997.

[14] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing, Second Edition*. Addison-Wesley, 2003.

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

[16] Florian Hielscher and Peter Gottschling. ParGraph. `http://pargraph.sourceforge.net/`, 2004.

[17] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 1993.

[18] J.-M. Jézéquel. Transparent parallelisation through reuse: Between a compiler and a library approach. In O. M. Nierstrasz, editor, *ECOOP'93 proceedings*, number 707 in Lecture Notes in Computer Science, pages 384–405. Springer Verlag, July 1993.

[19] Donald B. Johnson and Panagiotis Takis Metaxas. A parallel algorithm for computing minimum spanning trees. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 363–372, 1992.

[20] Elizabeth Johnson and Dennis Gannon. HPC++: Experiments with the Parallel Standard Template Library. In *International Conference on Supercomputing*, pages 124–131, 1997.

[21] Lie-Quan Lee, Lixin Ge, Marc Kowalski, Zenghai Li, Cho-Kuen Ng, Greg Schussman, Michael Wolf, and Kwok Ko. Solving large sparse linear systems in end-to-end accelerator structure simulations. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[22] 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.

[23] Lie-Quan Lee, Jeremy Siek, and Andrew Lumsdaine. The Generic Graph Component Library. In *Proceedings of OOPSLA'99*, 1999.

[24] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 96–114, 1993. `ftp://ftp.-cs.washington.edu/pub/orca/papers/lcpc93.ps`.

[25] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[26] Ulrich Meyer and Peter Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA '98: Proceedings of the 6th Annual European Symposium on Algorithms*, pages 393–404. Springer-Verlag, 1998.

[27] 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.

[28] Thomas Nitsche. Lifting sequential functions to parallel skeletons. *Parallel Processing Letters*, 12(2):267–284, June 2002.

[29] Christopher R. Palmer and J. Gregory Steffan. Generating network topologies that obey power laws. In *Proceedings of GLOBECOM '2000*, November 2000.

[30] 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.

[31] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[32] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. Generic programming for high performance numerical linear algebra. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[33] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *Boost Graph Library*. Boost, 2001. `http://www.boost.org/libs/graph/doc/index.html`.

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

[35] Y. H. Tsin. Some remarks on distributed depth-first search. *Information Processing Letters*, 82(4):173–178, 2002.

[36] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.