The Parallel BGL: A Generic Library for Distributed Graph Computations

Douglas Gregor and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor,lums}@osl.iu.edu

Abstract

This paper presents the Parallel BGL, a generic C++ library for distributed graph computation. Like the sequential Boost Graph Library (BGL) upon which it is based, the Parallel BGL applies the paradigm of generic programming to the domain of graph computations. Emphasizing efficient generic algorithms and the use of concepts to specify the requirements on type parameters, the Parallel BGL also provides flexible supporting data structures such as distributed adjacency lists and external property maps. The generic programming approach simultaneously stresses flexibility and efficiency, resulting in a parallel graph library that can adapt to various data structures and communication models while retaining the efficiency of equivalent hand-coded programs. Performance data for selected algorithms are provided demonstrating the efficiency and scalability of the Parallel BGL.

1 Introduction

The widespread use of software libraries improves research by providing a common infrastructure that amortizes the costs of implementing widely-used algorithms and data structures over all researchers and practitioners using the library. Libraries provide a language that improves dissemination of research results and simplifies comparison of alternatives when the library is used as common ground. Popular software libraries serve as a single point of optimization, where library optimizations can benefit a great number of existing and future applications without requiring changes in those applications. Perhaps more importantly, a single widely-used implementation is more likely to be reliable and correct.

The MPI [23, 36] and PVM [22] libraries have been very successful at abstracting the parallel execution environment, so that a parallel program written with either library can be easily ported to other environments, often retaining its performance characteristics. Moreover, many vendors provide implementations of MPI or PVM optimized for particular platforms, enabling better performance for any algorithm written using these libraries.

Libraries that abstract certain application domains, such as linear algebra or graphs, can be built on top of lower-level libraries. In the former domain, popular sequential libraries such as BLAS [32] and LAPACK [5] have spawned parallel variants [2, 9], simplifying the parallelization of programs written to the sequential interfaces. On the other hand, while there are several sequential graph libraries, such as LEDA [35], Stanford GraphBase [31], and JUNG [51] there are relatively few attempts at parallel graph libraries [3, 14, 29] and none that provide the flexibility needed in a general-purpose library.

The sequential Boost Graph Library (BGL) [43, 44], formerly the Generic Graph Component Library [34], is a high-performance generic graph library that is part of the Boost library collection [11]. The Boost libraries are a collection of open-source, peer-reviewed C++ libraries that have driven the evolution of library development in the C++ community [1] and ANSI/ISO C++ standard committee [6].

The Parallel Boost Graph Library (Parallel BGL) [26] provides data structures and algorithms for parallel, distributed computation on graphs. The Parallel BGL retains much of the interface of the (sequential) BGL on which it is based, simplifying the task of porting programs from the sequential BGL to the Parallel BGL. The Parallel BGL, like the BGL, is a generic library written in C++ with a dual focus on efficiency and flexibility. Following the principles of generic programming [39, 45] and written in a style similar to the C++ Standard Template Library (STL) [38, 46], data types provided by the Parallel BGL are parameterized by the underlying storage types (allowing one to create customized graphs with no performance penalty) and algorithms provided by the Parallel BGL are parameterized by the data types they operate on (allowing one to use any suitable graph type with the algorithm).

Section 2 briefly describes the generic programming methodology. Section 3 describes the syntax of the sequential and Parallel BGL, highlighting the similarities and differences between the two interfaces. An evaluation of the performance of several algorithms in the Parallel BGL, including a comparison against the CGM*graph* library [14], is given in Section 4. Section 5 discusses related work with conclusion and future work in Section 6.

2 Generic Programming

Generic programming has recently emerged as an important paradigm for the development of highly-reusable software libraries [7, 39]. 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 Generic Algorithms

Generic algorithms place the minimum requirements on the data types received as arguments. For instance, 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. A generic version of ***sum()*** could be defined this way.

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

This algorithm is 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. The ***sum()*** function (a simple version of ***accumulate()*** from the C++ Standard Template Library) can be used with arrays, linked lists, or any other type that meets the requirements for ***InputIterator*** and ***T***. For example:

```
double x[10];                        double a =  sum(x, x+10, 0.0);
vector <int> y(10);                  int  b =  sum(y.begin(),  y. end(), 0);
list  <complex<double>> z(10);       complex<double> c = sum(z.begin(), z. end(),  complex<double>(0.0,0.0));
```

2.2 Concepts

A characteristic aspect of generic programming is illustrated here: generic algorithms are defined in terms of properties of types, not in terms of any particular types. 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: valid expressions, associated types, semantics, and complexity 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 STL, generic algorithms for the Input Iterator concept include ***accumulate()***, ***copy()***, ***for_each()***, ***equal()***, ***transform()***, etc.

Some algorithms, however, need additional properties of the types to which they can be applied. When we add these requirements to those of the Input Iterator concept, we obtain a new concept that has fewer types but where each type is more specialized and computationally useful because more algorithms can be applied to it. For example,

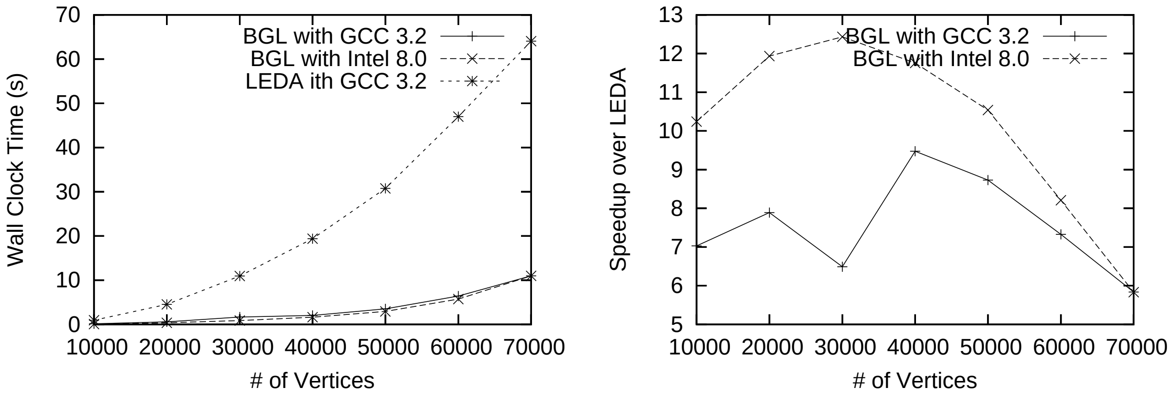


Figure 1: Sequential connected components performance on Erdös-Renyi graphs with an edge probability of 0.01.

adding the requirement that values in an iteration sequence can be visited more than once, we obtain the Forward Iterator concept, which enables algorithms such as STL's ***merge()***, ***fill()***, ***rotate()***, ***remove()***, ***replace()***, 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. 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.

2.3 Performance

Generic programming focuses on abstractions that can be eliminated by the compiler, so that instantiating a generic algorithm with a set of concrete types results in an algorithm as efficient as one written specifically for those types. Figure 1 illustrates that the generic connected components algorithm from the sequential Boost Graph Library performs significantly better than its counterpart in the non-generic LEDA library [35]. In fact, a prior version of the Boost Graph Library was found to be competitive even with highly-tuned Fortran code for sparse matrix ordering [33].

3 The (Parallel) Boost Graph Library

The parallel and sequential BGL provide efficient and flexible graph data structures and algorithms. The focus of both libraries is on graph algorithms that are generic with respect to the underlying graph representation and the properties of vertices and edges that guide the algorithms. Throughout this discussion, "BGL" refers to the sequential BGL, "Parallel BGL" refers to the Parallel BGL, and "(Parallel) BGL" refers to both the sequential and Parallel BGL.

Algorithms in the (Parallel) BGL typically accept several required parameters and have many optional, named parameters. For instance, the following is a typical usage of Dijkstra's single-source shortest paths algorithm:

```
dijkstra_shortest_paths(G, s,
                        weight_map(get(edge_weight, G)).
                        predecessor_map(predecessor).
                        distance_map(distance));
```

Here, ***G*** is a graph of unspecified type and ***s*** is the starting vertex (in graph ***G***) for the algorithm. Section 3.1 describes the allowable types for the parameter ***G***. The other three parameters are passed via the BGL named parameters mechanism, which associates ***get(edge_weight, G)*** with the ***weight_map*** parameter, ***predecessor*** with the ***predecessor_map*** parameter, and ***distance*** with the ***distance_map*** parameter.[1] The latter three arguments are *property maps*, which associate properties—such as the distance from the source or the weight of an edge—with vertices or edges in the graph but do not dictate a concrete representation. Property maps are discussed in Section 3.3.

The (Parallel) BGL implementation of Dijkstra's algorithm is a great deal more customizable than is shown here. For instance, one may replace the representations of zero and ∞, the manner in which distances are compared and

[1]The period separating the named arguments is not an error; it is used by the named parameters simulation code.

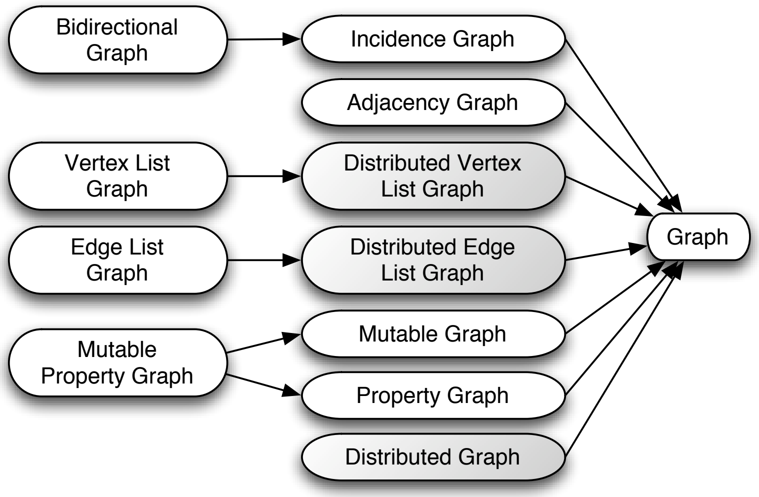


Figure 2: (Parallel) Boost Graph Library concept taxonomy with parallel-only concepts shaded.

added, introduce a visitor for key events such as examining or relaxing an edge, or change the property map associating weights with edges. The implementation of Dijkstra's algorithm in the Parallel BGL is built on the sequential implementation from the BGL, by applying the generic programming process of lifting [28].

3.1 Graph Concepts

Algorithms in the (Parallel) BGL are generic with respect to the graph type and representation. However, each algorithm necessarily places certain requirements on the graph type and other argument types. Following the generic programming approach, the requirements have been extracted from each algorithm and documented as graph concepts. Great care has been taken to ensure that the concepts of the Parallel BGL correspond as closely to the concepts of the BGL as possible, so that applications using the BGL can be ported to the Parallel BGL without a great deal of effort.

Figure 2 illustrates the (Parallel) BGL graph concept taxonomy. The top-level Graph concept introduces only a few simple requirements: the type must have associated types that name vertices and edges (called vertex and edge *descriptors*), along with some additional identification information. Other concepts of interest include:

- Incidence Graph: Requires that the set of edges outgoing from a given vertex be accessible in constant time. This concept is essential for algorithms that traverse the graph according to the adjacency structure, such as breadth-first- and depth-first—search.

- Bidirectional Graph: Requires that the set of edges incoming to a given vertex be accessible in constant time, in addition to the outgoing edges. This concept allows one to reverse the edges of a graph as an adaptor (called ***reverse_graph*** in the BGL) without any performance penalty.

- Distributed Edge List Graph: Requires that the set of edges local to a process be accessible in constant time. The union of the edge sets returned on all processes must be the set of all edges and the pairwise intersection of these edge sets must be the empty set. The distributed minimum spanning tree algorithms of Dehne and Götz [18] require a Distributed Edge List Graph.

- Edge List Graph: Requires that the set of edges in the graph be accessible in constant time. The Bellman-Ford single-source shortest paths algorithm, for instance, requires an edge list but does not require access to the in- or out-edges of a given vertex. Alternatively, one could define this concept as the Distributed Edge List Graph concept but adding the requirement that the number of processes be one.

- Distributed Graph: Imposes additional restrictions on the vertex and edge descriptor associated types defined by the Graph concept. In particular, these descriptors model the Global Descriptor concept, which permits one to retrieve the owning process of any descriptor and the local identifier (valid only in the owning process) of that descriptor.

Different graph representations meet the requirements of different concepts within the taxonomy. A directed adjacency list, for instance, will model Graph, Incidence Graph, Adjacency Graph, Vertex List Graph, and Edge List Graph. It may also model Bidirectional Graph, but only if in-edges are stored in addition to out-edges. If the adjacency list is then distributed, it will no longer model Vertex List Graph and Edge List Graph but will instead model Distributed Vertex List Graph and Distributed Edge List Graph. An undirected (potentially distributed) adjacency list would have different characteristics—and, therefore, a different set of concepts—as would an adjacency matrix. The next section describes the primary graph data structure of the (Parallel) BGL. However, it is important to realize that the Parallel BGL can be applied to graph data structures not provided by the library: any distributed graph-like data structure can be augmented to support the graph concepts described above, permitting distributed graph computations using the Parallel BGL to be applied to existing distributed data structures.

3.2 The *adjacency_list* Class Template

The adjacency list is the most widely used graph representation for the (Parallel) BGL, but it is not the only graph representation. The BGL contains an adjacency matrix, a simple vector-as-graph adaptor, and adaptors that permit usage of the LEDA graph data structures [35] with BGL algorithms. The adjacency list generator provides efficient storage for all but very dense graphs but also meets the requirements for many of the graph concepts and, therefore, many of the generic graph algorithms. However, the adjacency list isn't a single, concrete representation: it is a type generator that gives users several degrees of freedom when selecting the underlying data structures for storing edge and vertex lists (e.g., a vector representation for efficient access or a linked list for efficient insertion and deletion) or determining what extra information should be associated with edges and vertices (such as edge weights and vertex labels). For instance, the following ***typedef*** generates a particular type of adjacency list:

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

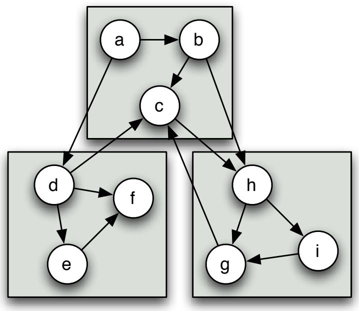
The ***listS*** and ***vecS*** tags indicate that the edge lists for each vertex should be stored in a linked list and that the vertices in the graph be stored in a vector, respectively. Other options include ***setS*** to use a set data structure (and thereby eliminate parallel edges automatically) or ***multisetS*** to use a multiset data structure that permits parallel edges but keeps edge lists sorted. The ***bidirectionalS*** tag indicates that the edges of the graph are directed and that the adjacency list should also store incoming edges; the resulting type ***Graph*** therefore models the Bidirectional Graph concept.

The two ***property<...>*** instantiations attach properties to the vertices and edges of the graph, respectively. For instance, ***property<vertex_distance_t, double>*** attaches a distance to each vertex (stored as a ***double***). This distance can be accessed using the tag type ***vertex_distance_t*** (or the canonical value of that type, ***vertex_distance***) via a *property map*. Multiple properties can be attached to any graph by nesting ***property<...>*** instantiations, permitting any number of user-defined properties. Property maps will be discussed further in Section 3.3.

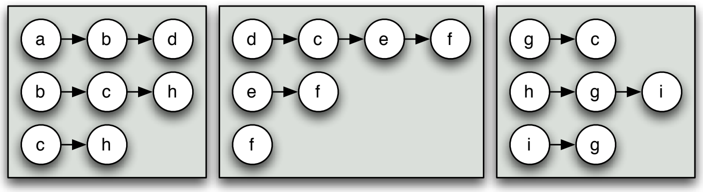
In the Parallel BGL, the vertices of the graph can be distributed across processes using the ***distributedS<...>*** selector for the vertex list. For instance, the following graph is a distributed version of the previous graph:

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

Note that the only change here is that the vertex storage selector has been changed from ***vecS*** to ***distributedS<mpi::bsp_process_group, vecS>***. The first parameter to the ***distributedS*** selector is the *process group*



(a) Distributed graph



(b) Distributed adjacency list representation

Figure 3: A distributed directed graph represented as an adjacency list across three processors.

over which the graph will be distributed; in this case, the graph uses MPI for communication with the Bulk Synchronous Parallel (BSP) model [47, 48]. Process groups are discussed in greater detail in Section 3.4. The second parameter is the kind of storage to be used for vertices local to each process; in this case, they are stored within a vector.

Figure 3 illustrates how the vertices and edges of the distributed adjacency list are stored. The vertices are distributed among the processes and all edges outgoing from an vertex are stored with that vertex. This layout ensures that the distributed graph models the Incidence Graph concept (important for, e.g., breadth-first and depth-first search), subject to the restriction that out-edges are available only in the process owning the vertex. In addition, the properties attached to vertices and edges (distance and weight, respectively) are stored along with the vertices or edges. Thus, the distance property of a vertex is stored in the process that owns the vertex.

Once defined, graphs can be constructed in several ways. The functions ***add_vertex()*** and ***add_edge()*** add vertices and edges to the graph, but with the Parallel BGL the user is required to manage the distribution manually, e.g., vertices and edges can only be added locally.[2] However, when the number of vertices is known in advance one can construct the graph with an iterator-based interface. For instance, the following builds an Erdös-Renyi [19] graph with ***n*** vertices, edge probability ***p***, and draws random numbers from the random generator ***gen***:

```
Graph G(erdos_renyi_iterator<RandomGen, Graph>(gen, n, p),
        erdos_renyi_iterator<RandomGen, Graph>(),
        n);
```

Changing the instances of ***Graph*** to ***DistGraph*** will create a distributed graph that will automatically distribute the vertices and edges according to a block distribution. Alternative distributions can be provided as arguments to the graph constructor.

(Local) vertices and edges in the graph can be accessed via the ***vertices()*** and ***edges()*** functions of the adjacency list and by following edges in the adjacency list. The types of vertex and edge descriptors, which are used to name vertex or edges in the graph, are defined in an auxiliary traits structure [40] called ***graph_traits***. One can access the vertex descriptor type of the graph ***Graph*** with:

```
typedef graph_traits<Graph>::vertex_descriptor Vertex;
```

In both distributed and non-distributed graphs, one can access the i^{th} vertex in the graph (globally) via the ***vertex()*** function. For instance, the invocation of Dijkstra's algorithm could begin at vertex 0 in the graph:

```
Vertex s = vertex(0, G);
```

[2]This restriction is required to ensure local completion semantics.

3.3 Property Maps

Property maps associate additional information with the vertices or edges of a graph and allow algorithms to be decoupled from the representation of graph attributes. Property maps can be used to access the internal properties described in the previous section, such as the weight of an edge, but can also associate external data with graphs. Internal properties of a graph can be accessed via the ***get()*** function and their type can be determined using the ***property_map*** class template. For instance, the following code creates a property map that can access the distance of any vertex in the graph ***G***:

```
property_map<Graph, vertex_distance_t>::type distance = get(vertex_distance, G);
```

When the graph ***G*** is distributed, the code is unchanged but the semantics differ slightly. The distance of all local vertices will be immediately available, but the distance of a remote vertex will be inaccessible until after it is requested. The discussion of the semantics of distributed property maps is deferred to the end of this section.

Properties can be stored externally to the graph via any number of data structures. One need only provide a suitable property map to access these data structures. The (Parallel) BGL provides several of these adaptors, the most common of which is the ***iterator_property_map***, which accepts a random access iterator pointing to the beginning of a data structure along with a mapping from keys to indices within the data structure. For instance, we can define the mapping from vertices to predecessors as follows:

```
// A mapping from vertices to indices
typedef property_map<Graph, vertex_index_t>::type VertexIndexMap;

// Predecessor vector and property map
vector<Vertex> pred_vec(num_vertices(G));
iterator_property_map<vector<Vertex>::iterator, VertexIndexMap> predecessor(pred_vec.begin(), get(vertex_index, G));
```

The initial ***typedef*** extracts the type of the built-in property map that associates vertices in the graph with integers in the range ***[0, num_vertices(G))***. The second line of code constructs a vector with enough storage to contain predecessors for each vertex, and the final line of code constructs a property map to access these predecessors. The same code operates similarly on a distributed graph, where ***num_vertices(G)*** returns the number of vertices stored locally, so that each process stores only data for local vertices. Ghost cells can then be created to cache values for vertices in other processes.

Although distributed property maps mimic the interface of non-distributed property maps and model the same concepts, the semantics of the primary property map operations, ***get()*** and ***put()***, differ slightly to ensure local completion semantics:

- ***get(p, k)***: Retrieves the value associated with key ***k*** from the property map ***p***. For local keys and non-distributed property maps, this value will be stored in a local property map and can be accessed directly. For remote keys, the value will be present in a ghost cell. If no such ghost cell exists, one will first be created and filled with a default value. The origin of the default value will be discussed shortly.

- ***put(p, k, v)***: Replaces the value associated with key ***k*** with the new value ***v***. For local keys and non-distributed property maps, this involves copying ***v*** into the local property map. For remote vertices, a message will be sent to the owner of ***k*** containing the new value.

A process may receive several messages with updated values for a given key, corresponding to concurrent ***put()*** operations on different processes. Distributed property maps contain a *resolver* that combines the values received via messages with the values stored locally. The combination method differs based on the use of the property map: for instance, a property map containing shortest distances will always select the smaller value, while a property map used to store ***visited*** flags would perform a logical OR of the values. Resolvers are also responsible for providing a default value when a ghost cell is created by a ***get()*** operation, e.g., ∞ for shortest distances or ***false*** for ***visited*** flags.

Distributed property maps can be constructed with the class template ***distributed_property_map*** given the process group of the graph, a local property map, and a key type that is a model of the Global Descriptor concept. Thus, any non-distributed property map can be adapted to a distributed property map with little effort. However, the Parallel BGL provides automatic adaptors (by way of class template partial specialization) that permit code using property maps to remain unchanged when moving from a non-distributed graph to a distributed graph. These adaptors follow the Execution Instance Ordering pattern [27].

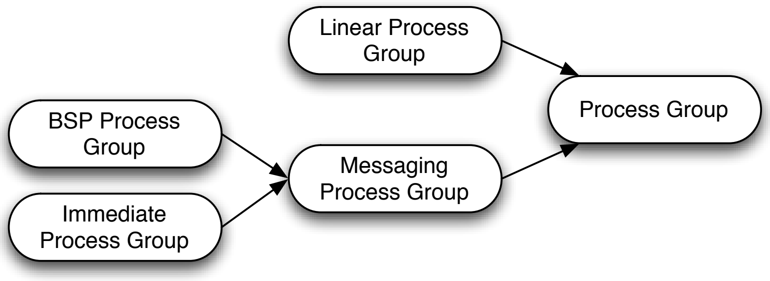


Figure 4: Partial Parallel BGL process group concept taxonomy.

Property maps are an important part of the BGL, abstracting access to data away from the storage of that data. The Parallel BGL enriches this abstraction, permitting both internal and external property maps to be distributed across processes. Moreover, the shift to distributed property maps has little or no effect on the program source code: all program code presented this paper will execute efficiently on either distributed or non-distributed graphs.

3.4 Process Groups

Process groups abstract the notion of several processes cooperating to perform some computation. The Parallel BGL defines a concept taxonomy for process groups (see Figure 4), where concepts differ based on the topology of processors (e.g., free-form, linear, or an array) and the method of communication (e.g., message-passing vs. shared memory). Process group concepts provide the crucial separation between the requirements an algorithm places on its communication layer and the implementation of those requirements. For instance, an algorithm's semantics may not be dependent on the order in which messages are delivered, so long as it can detect when all messages have been received. Instantiating the algorithm with different process groups results in algorithms with different communication behavior, consistency models, and performance, although the algorithm remains correct. Thus, the algorithms can be provided with process groups that are optimized for the hardware on which the algorithm will be executed. At present, we are primary interested in the Messaging Process Group concept, which requires the existence of several communication operations:

- ***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.
- ***receive(pg, source, tag, value)***: Receive a message containing ***value*** from process ***source*** with the given ***tag***.
- ***probe(pg)***: Immediately returns a ***(source, tag)*** pair if a message is available, or a no-message indicator.
- ***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.

The requirements of the Messaging Process Group concept are similar to the BSP parallel communication model, in that messages received are not guaranteed to be received until after ***synchronize()*** is invoked. Note, however, that this definition permits messages to be delivered *before* a synchronization. Thus, the Messaging Process Group concept will be modeled by a process group type that immediately sends messages but includes synchronization. The Parallel BGL also defines a BSP Process Group concept that refines Messaging Process Group but adds the requirement that messages not be delivered before ***synchronize()*** is called. Finally, the Immediate Process Group refines Messaging Process Group by requiring that ***receive()*** block while waiting for messages and that receives be matched to sends without an intervening ***synchronize()***.

The Parallel BGL implementation of Dijkstra's algorithm is an example of an algorithm that is specified in terms of the Messaging Process Group concept, but has different behavior and performance characteristics depending on the message delivery time. When messages are delivered only on calls to ***synchronize()*** boundaries, as with process groups that model the BSP Process Group concept, Dijkstra's algorithm proceeds via a series of supersteps. Within each superstep, the processors communicate to find the global minimum shortest path from the source vertex to any vertex not already completed. Each processor relaxes the edges outgoing from any vertex that has a distance "close enough" to the global minimum distance, as determined by various heuristics [16, 37]. The processors then communicate relaxed edges and enter the next superstep.

When Dijkstra's algorithm is provided with a process group that immediately delivers messages (e.g., a process group that models the Immediate Process Group concept), the notion of a superstep changes from the BSP model. The algorithm is roughly the same, but instead of queuing updates that arise from relaxing edges, the updates are sent and processed immediately. Thus, when the communication medium is very fast, each superstep may process more vertices, leading to fewer supersteps. With fast interconnects and sparse graphs, this formulation may prove more efficient than the BSP model. This decision is left to the user, who may choose what kind of process group is appropriate for her environment and instantiate the implementation of Dijkstra's algorithm accordingly.

4 Experimental Results

We evaluated the performance of the Parallel BGL on graphs generated with various graph models: Erdös-Renyi [19], small-world [50], and scale-free [8, 20]. Erdös-Renyi graphs are the best-studied random graphs and are often used for theoretical evaluations of algorithm complexity. Small-world graphs maintain a high level of clustering, small diameter, and tend to distribute well; we typically expect the best performance and scalability with these graphs and use a rewiring probability of 0.04 to ensure a small diameter. Scale-free graphs, which we generate with the Power-Law Out Degree (PLOD) algorithm [42], have power-law degree distributions (we fix the power at -0.5 for our tests) that reflect those of real-world networks such as the web graph or actor collaboration networks [8, 20]. Algorithms that rely on traversing the edges incident on each vertex may exhibit poorer performance on scale-free graphs due to the large standard deviation in vertex degrees. For these experiments, we generated all of the graphs randomly but did not partition the graphs before invoking each algorithm.

We performed our performance evaluation on Odin, which consists of 128 compute nodes connected via Infiniband. Each node contains 4GB main memory with two 2.0GHz AMD Opteron processors, but for our tests we have left one processor idle on each node. The Parallel BGL tests were compiled using a prerelease version of Boost 1.33.0 [11] (containing the sequential BGL) and version 0.5.0 of the Parallel BGL [26]. Comparisons against CGM*graph* [14] used version 0.9.5 beta[3], available from the CGM*lib* web site [15]. All programs were compiled with version 3.4.3 of the GNU C++ compiler [21] using optimization level −***O3*** and LAM/MPI 7.1.1 [12].

In this paper we report results for several algorithms implemented in the Parallel BGL. More complete performance data for Parallel BGL algorithms is available on the project's web page [26].

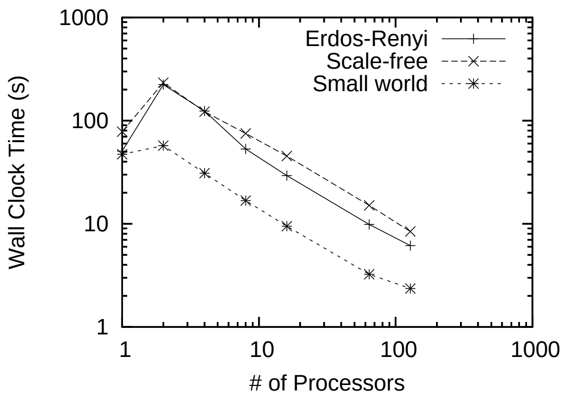
4.1 Dijkstra's Algorithm

Dijkstra's algorithm computes shortest paths from a given source vertex to every vertex in the graph. The implementation of Dijkstra's algorithm in the Parallel BGL uses the naïve formulation of parallel Dijkstra's algorithm [25, §10.7.2], but augments it with priority queues that use a variety of heuristics to increase the number of vertices that can be processed in a given superstep. The Eager implementation of Dijkstra's implementation looks ahead a constant distance to find vertices near the global minimum distance in each superstep [37]. Alternative heuristics consider the weights of incoming and outgoing edges to each vertex, as described by Crauser et al [16].

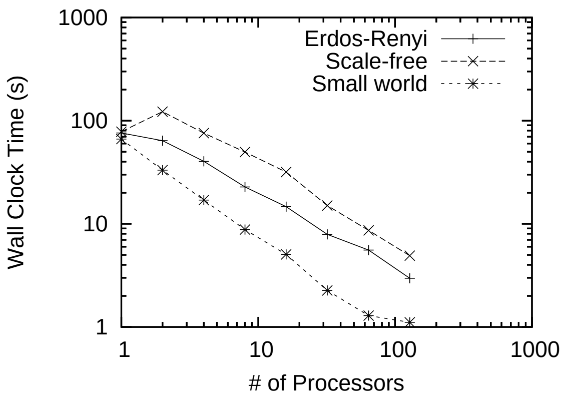
Figure 5 illustrates the performance of the Eager implementation using a constant lookahead of 0.1. For the dense graph (left), the lookahead value 0.1 is much too large, resulting in an immediate performance degradation because the algorithm must revisit the same vertices multiple times. As the number of processors increases, the vertices are more spread out and the negative effects of the over-eager lookahead are not as pronounced. The lookahead of 0.1 was manually tuned for the sparse graphs on the right-hand side of the figure, where it provides excellent scalability and performance.

Figure 6 illustrates the performance of the Parallel BGL implementation of Crauser et al.'s heuristics for Dijkstra's algorithm [16]. The performance scales slightly better on dense graphs than on sparse graphs, but is reasonable in both cases up to 32 processors. Beyond 32 processors, the heuristics cannot expose sufficient parallelism to keep all

[3]We made minor changes to the source code to facilitate newer C++ compilers than the code was originally developed with, but no changes to any part that is included in timing tests.



(a) $100k$ vertices, $15M$ edges



(b) $1M$ vertices, $15M$ edges

Figure 5: Eager Dijkstra's single-source shortest paths algorithm on randomly-generated graphs with edge weights uniformly distribution in $[0, 1)$ and a constant lookahead of 0.1.

processors busy. However, this algorithm does have the advantage that it does not need to be tuned by a lookahead parameter. By default, the ***dijkstra_shortest_paths()*** implementation applies the Crauser et al. heuristics; supplying an explicit lookahead value employs the Eager Dijkstra algorithm.

4.2 Connected Components

The Parallel BGL implements a variation of the connected components algorithm by Goddard, Kumar, and Prins [24] using some optimizations from Johnson and Metaxas [30] and adapted for distributed memory. We evaluated our implementation on graphs generated with all three graph models.

Figure 7 illustrates the performance of our implementation on graphs with 100,000 vertices and roughly 15 million undirected edges (for an average degree of around 300), with execution time on the left and speedup on the right. We note that scalability is quite good up to 64 processors and that the algorithm performs slightly better on small-world graphs than scale-free or Erdös-Renyi graphs. We have also evaluated the performance of the algorithm on much more sparse graphs (average degree of around 30), where a load imbalance due to large connected components results in poor scalability past 16 processors; we are currently investigating these problems (which have been noted by other researchers [13]) to determine if they are due to the algorithm itself or the implementation.

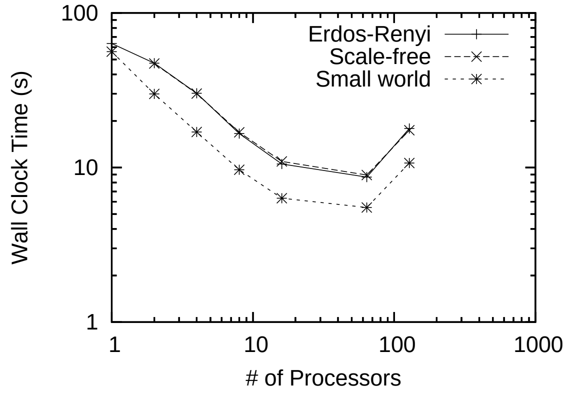
Figure 8 illustrates the performance of the Parallel BGL relative to CGM*graph*, using Erdös-Renyi graphs with 96,000 vertices and 10 million edges[4]. The CGM*graph* results shown in the left-hand plot exhibit the same trends as the results presented in [14], although our cluster is faster. On the same graphs, the Parallel BGL exhibits better scalability and performs significantly faster. The right-hand plot shows the speedup of the Parallel BGL over CGM*graph*, which varies between 17 and 30. Inspection of the CGM*lib* and CGM*graph* source code leads us to believe that the use of virtual function calls within the communication infrastructure is causing the poor performance.[5]

4.3 Minimum Spanning Tree

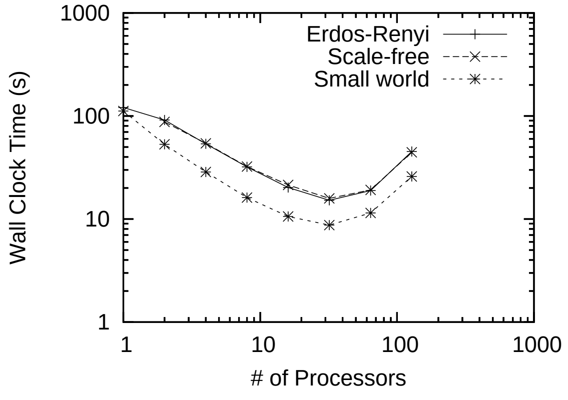
The Parallel BGL contains implementations of four algorithms that solve the distributed minimum spanning tree (MST) problem due to Dehne and Götz [18]. Figure 9 illustrates the performance and speedup of each algorithm for the three random graph models, for both dense (left) and sparse (right) graphs. On the dense graphs, the "Merge MSF" algorithm scales more poorly than the others, but the scalability of all algorithms is reasonable, with the "Then-Merge" algorithm slightly faster than the others. It is interesting to note that the simple parallelization of Borůvka's algorithm

[4]These graphs are equivalent to the largest graphs reported by Chan and Dehne [14].

[5]We have seen similar effects—with similar speedups—when comparing the (sequential) BGL to Java graph libraries.

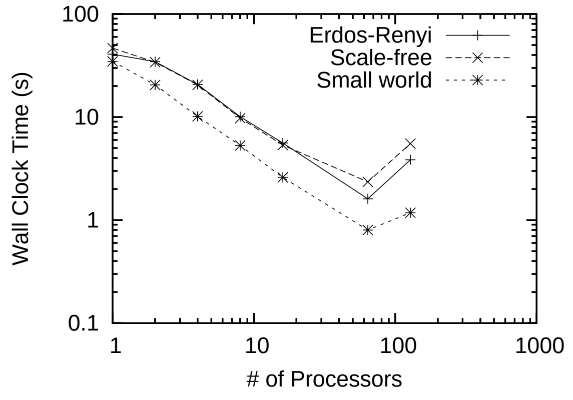


(a) $100k$ vertices, $15M$ edges



(b) $1M$ vertices, $15M$ edges

Figure 6: Crauser et al.'s single-source shortest paths algorithm on randomly-generated graphs with edge weights uniformly distribution in $[0, 1)$.



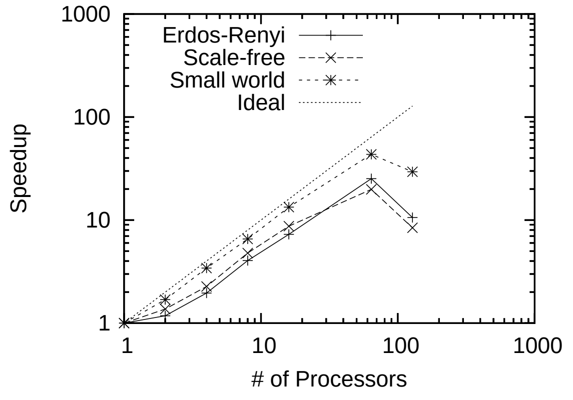


Figure 7: Connected Components performance on randomly-generated graphs with $100k$ vertices and $15M$ edges.

performs as well as any of the other, theoretically better algorithms. Although the plots do not convey scale, the minimum spanning tree algorithms are roughly four times slower than the connected components algorithm.

The right-hand side of Figure 9 illustrates the performance of the MST algorithms on sparse graphs with an average degree of around 30. Here we see that the algorithms do not scale well, with performance leveling off around 32 processors. This is expected, as it confirms the theoretical results presented by Dehne and Götz [18]: the algorithms are only expected to be efficient when $m/n \geq p$, i.e., the number of processors does not exceed the average degree. The parallel Borůvka and Then-Merge algorithms still perform reasonably well on these graphs. Both algorithms appear to be good choices in practice.

4.4 Graph Coloring

The Parallel BGL implements the vertex coloring algorithm due to Boman et al. [10]. The algorithm has its roots in the standard sequential vertex coloring algorithm, which visits the vertices in the graph in some specified order and, for each vertex, selects the lowest-numbered color that creates a valid coloring. The Boman et al. algorithm has the same sequential step, but after each superstep processors exchange vertex colors with their neighbors to resolve any conflicts.

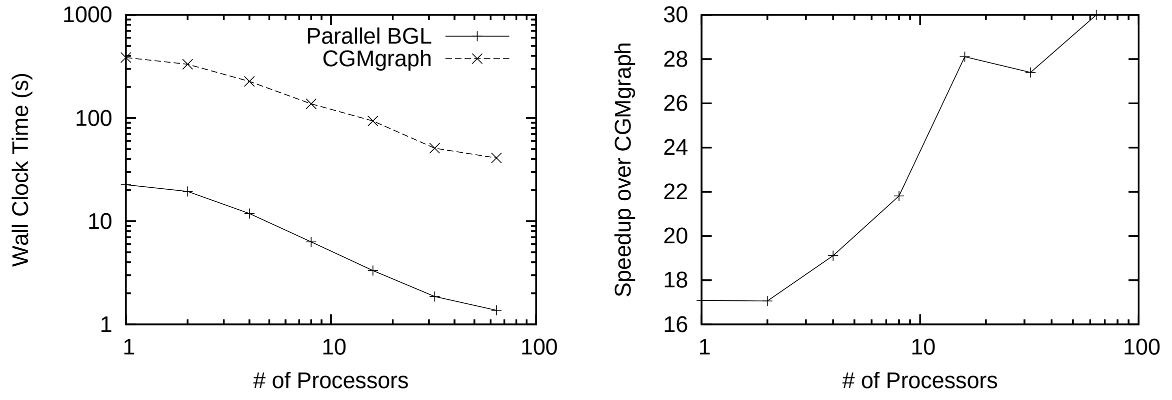


Figure 8: Connected Components performance of the Parallel BGL and CGM*graph* on Erdös-Renyi graphs with $96k$ vertices and $10M$ edges.

Figure 10 illustrates the performance of the Parallel BGL implementation of the vertex coloring algorithm. On small-world graphs, where the majority of the neighbors of each vertex are locale, the implementation provides excellent scalability, as expected. Also, sparse graphs tend to scale much better than dense graphs, since the density of graphs directly effects the number of colors required to represent the graph and the number of conflicts that occur in the communication phase.

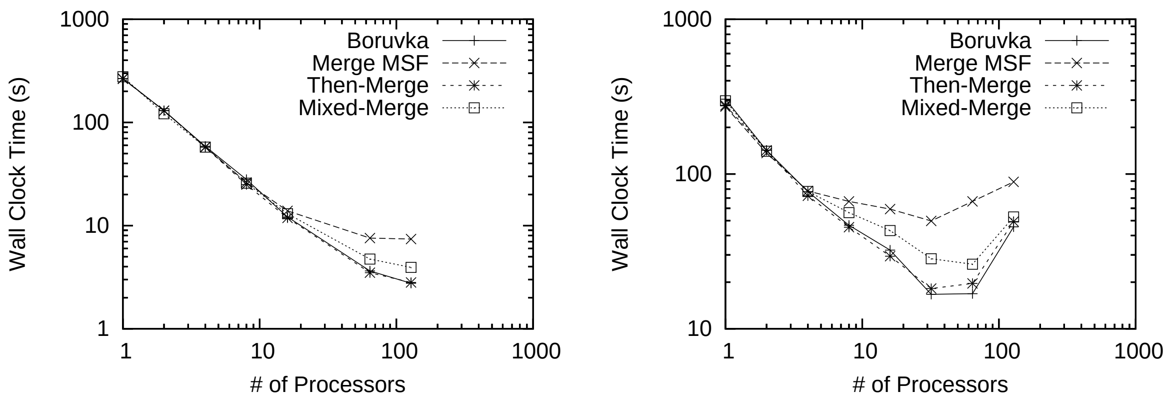
4.5 Scalability on Large Graphs

All of the performance results we have shown previously illustrate the speedup provided by algorithms in the Parallel BGL on relatively small graphs that fit in the memory of a single compute node. Figure 11 illustrates the performance and scalability of selected Parallel BGL algorithms on small-world graphs with $546,875$ vertices per processor ($70M$ vertices on 128 processors) and an average degree of 30 ($> 1B$ edges on 128 processors). Ideally, the plots for each algorithm in Figure 11 would be nearly horizontal lines, as the number of vertices and edges is scaled linearly with the number of processors. Breadth-First Search, Eager Dijkstra's shortest paths (with lookahead 0.1), and Connected Components all scale reasonably well up to 128 processors. However, the vertex coloring and Crauser et al. shortest paths algorithms do not fare as well. Notably absent from these performance results are the minimum spanning tree algorithms described in Section 4.3: by design, these algorithms require each processor to store information about each vertex in the graph. While this is feasible for smaller graphs, it is not possible to store this information for each processor with a $70M$-vertex graph, so we have chosen to omit partial results.

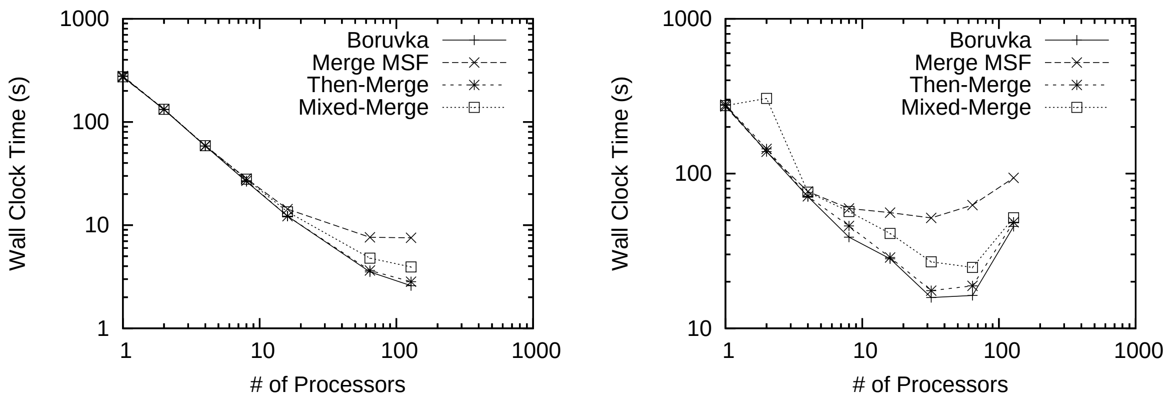
5 Related Work

The CGM*graph* library [14, 15] implements several graph algorithms, including Euler tour, connected components, spanning tree, and bipartite graph detection. It uses its own communication layer built on top of MPI and based on the Course Grained Multicomputer (CGM) [17] model, a variant of the BSP model. From an architectural standpoint, CGM*graph* and the Parallel BGL adopt different programming paradigms: the former adheres to Object-Oriented principles whereas the latter follows the principles of generic programming. We have provided comparative results in Figure 8 for the only algorithm the two libraries have in common at this time.

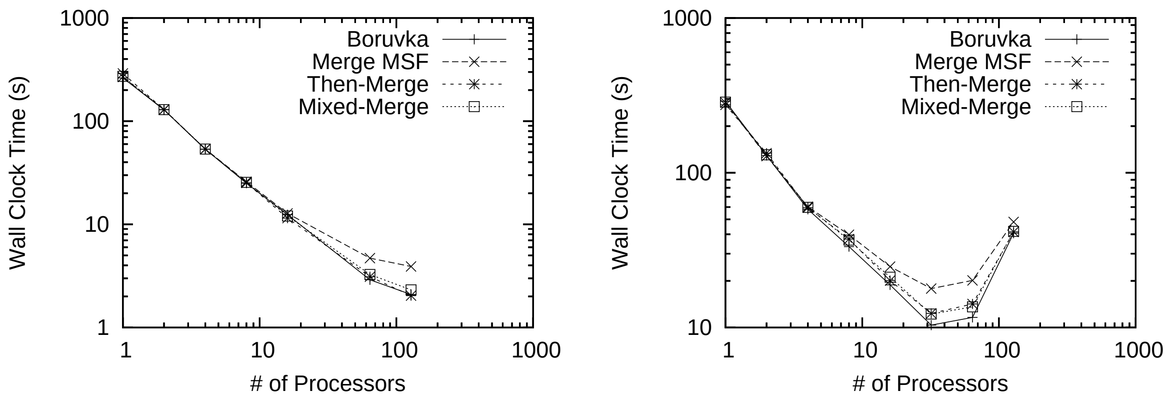
The ParGraph library [29] shares many of its goals with the Parallel BGL. Both libraries are based on the sequential BGL and aim to provide flexible, efficient parallel graph algorithms. More importantly, both libraries directly address the problem of distributed property maps. The libraries differ in their approach to parallelism: the Parallel BGL stresses source-code compatibility with the sequential BGL, eliminating most explicit communication. ParGraph, on the other hand, represents communication explicitly, which may permit additional optimizations. Since ParGraph provides only maximum-flow and breadth-first search implementations, so we have not provided a performance comparison.



(a) Erdös-Renyi

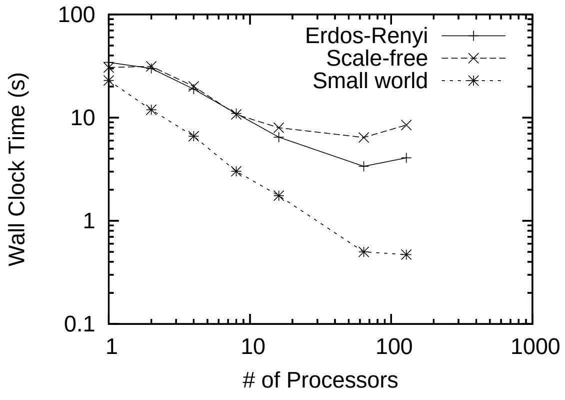


(b) Scale-free

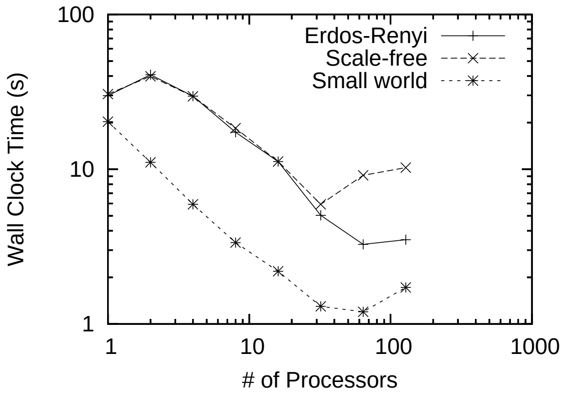


(c) Small world

Figure 9: Minimum Spanning Tree performance on randomly-generated graphs with edge weights uniformly distributed in $[0, 1)$. The left-hand charts represent dense graphs ($100k$ vertices, $15M$ edges) and the right-hand charts represent sparse graphs ($1M$ vertices, $15M$ edges).



(a) $100k$ vertices, $15M$ edges



(b) $1M$ vertices, $15M$ edges

Figure 10: Vertex coloring performance on randomly-generated graphs.

The Standard Template Adaptive Parallel Library (STAPL) [3, 4] is a generic parallel library providing data structures and algorithms whose interfaces closely match those of the C++ Standard Template Library. STAPL and Parallel BGL both share the explicit goal of parallelizing an existing generic library, but their approach to parallelization is quite different. STAPL is an adaptive library, that will determine at *run time* how best to distribute a data structure or parallelize an algorithm, whereas the Parallel BGL encodes distribution information (i.e., the process group) into the data structure types and makes parallelization decisions at *compile time*. Run time decisions potentially offer a more convenient interface, but compile time decisions permit the library to optimize itself to particular features of the task or communication model (an *active library* [49]), effectively eliminating the cost of any abstractions we have introduced. STAPL includes a distributed graph container with several algorithms. When STAPL becomes available, we intend to conduct performance comparisons with overlapping algorithms and introduce a Parallel BGL process group using STAPL's communication layer, which will provide a more flexible, adaptive option to users of the Parallel BGL.

6 Conclusion

The Parallel BGL is a generic C++ library of graph algorithms and data structures that inherits the flexible interface of the sequential Boost Graph Library. Algorithms in the Parallel BGL are scalable, matching the scalability of other libraries and theoretical results, and efficient, maintaining a large speedup over one other publicly-available parallel graph library.

The Parallel BGL sports a modest selection of distributed graph algorithms, including breadth-first— and depth-first search, Dijkstra's single-source shortest paths algorithm, connected components, minimum spanning tree, and PageRank [41]. We are currently extending the library with additional algorithm implementations, in particular maximum-flow algorithms and graph partitioning.

We are also investigating extensions to the sequential (and Parallel) BGL to support fine-grained parallelism. We are particularly interested in studying the abstractions required to support high-performance parallelism on shared-memory machines and their relation to the process groups we have devised for distributed memory. With a unified model, we hope to seamlessly support clusters of SMPs within the Parallel BGL.

7 Availability

The current release of the Parallel BGL, which supports primarily distributed-memory computation, is available at `http://www.osl.iu.edu/research/pbgl` under a BSD-style license. It integrates with the (sequential) BGL, available as part of the Boost libraries [11].

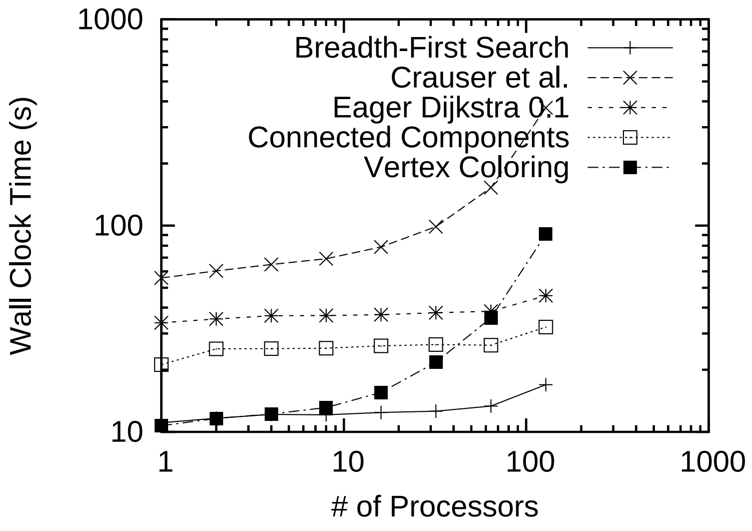


Figure 11: Performance of various algorithms in the Parallel BGL on small-world graphs with $546,875$ vertices per processor and an average degree of 15.

8 Acknowledgements

This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment. Thanks to the National Science Foundation for supporting this research under a grant numbered EIA-0202048. The authors also thank Brian Barrett and Chris Mueller for their help with performance analysis and Nick Edmonds, who implemented the connected components algorithm for the Parallel BGL.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[2] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, and Robert van de Geijn. PLAPACK: Parallel linear algebra package. In *SIAM Parallel Processing Conference*, 1997.

[3] 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.

[4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, August 2001.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, editors. *LAPACK User's Guide*. SIAM, 3rd edition, 199.

[6] Matt Austern. (draft) technical report on standard library extensions. Technical Report N1711=04-0151, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2004.

[7] Matthew H. Austern. *Generic Programming and the STL*. Professional computing series. Addison-Wesley, 1999.

[8] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509–512, October 1999.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: a linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997)*, 1997.

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

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

[12] Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[13] Libor Buš and Pavel Tvrdík. A parallel algorithm for connected components on distributed memory machines. In *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 280–287. Springer-Verlag, 2001.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] Paul Erdos and Alfred Renyi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[20] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262. ACM Press, 1999.

[21] GNU compiler collection. `http://www.gnu.org/software/gcc/`, 2004.

[22] 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.

[23] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

[24] 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.

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

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

[27] Douglas Gregor and Andrew Lumsdaine. The execution instance overloading pattern. In *Workshop on Patterns in High-Performance Computing*, 2005.

[28] Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, 2005.

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

[30] 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.

[31] D. E. Knuth. *Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, 1994.

[32] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[33] 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.

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

[35] K. Mehlhorn and St. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

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

[37] 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.

[38] David R. Musser, Gillmer J. Derge, and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001.

[39] 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.

[40] Nathan Myers. A new and useful technique: "traits". *C++ Report*, 7(5):32–35, June 1995.

[41] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, November 1998.

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

[43] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

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

[45] Alexander A. Stepanov. Generic programming. *Lecture Notes in Computer Science*, 1181, 1996.

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

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

[48] Leslie G. Valiant. General purpose parallel architectures. In *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 943–973. MIT Press, 1990.

[49] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[50] Duncan Watts and Steven Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[51] Scott White, Joshua O'Madadhain, Danyel Fisher, and Yan-Biao Boey. Java Universal Network/Graph framework. `http://jung.sourceforge.net/`, 2004.