Single-Source Shortest Paths with the Parallel Boost Graph Library

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

Abstract

The Parallel Boost Graph Library (Parallel BGL) is a library of graph algorithms and data structures for distributed-memory computation on large graphs. Developed with the Generic Programming paradigm, the Parallel BGL is highly customizable, supporting various graph data structures, arbitrary vertex and edge properties, and different communication media. In this paper, we describe the implementation of two parallel variants of Dijkstra's single-source shortest paths algorithm in the Parallel BGL. We also provide an experimental evaluation of these implementations using synthetic and real-world benchmark graphs from the 9th DIMACS Implementation Challenge.

1 Introduction

Large-scale graph problems arise in many application areas. As with other large-scale computations, large-scale graph computations can potentially benefit from the use of high-performance parallel computing systems. Parallel graph algorithms have been well-studied in the literature [20, 26] and selected algorithms have been implemented for shared memory [15, 17, 19, 27, 33], distributed memory [10, 25], and highly multithreaded architectures [6, 7]. While such implementations are important to demonstrate proof of concept for parallel graph algorithms, they tend to be of limited use in practice because such implementations are not typically reusable outside of their test environments. As a result, new uses of a given parallel graph algorithm will almost surely have to be implemented from scratch.

In mainstream software development, software libraries provide a common infrastructure that amortizes the costs of implementing widely-used algorithms and data structures. In computational sciences, software libraries can be extremely valuable to an entire research community. Libraries provide a language that improves dissemination of research results and simplifies comparison of alternatives. In addition, a single widely-used (and thus widely-studied) implementation is more likely to be reliable, correct, and efficient. While there are several high-quality sequential graph libraries available, such as LEDA [34], Stanford GraphBase [29], and JUNG [45] there are relatively few attempts at parallel graph libraries. Of the parallel graph libraries that have been reported in the literature [3, 13, 24], none provide the flexibility needed in a general-purpose library.

In addition to flexibility, performance is an important concern for software libraries. Libraries such as BLAS [30] in the scientific computing community exist specifically to provide high levels of performance to applications using them. However, tensions may arise between the needs for flexibility in a library and the needs for performance. Recently, the generic programming paradigm has emerged as an approach for library development that simultaneously meets the needs of flexibility and performance.

The sequential Boost Graph Library (BGL) [39, 40], (formerly the Generic Graph Component Library [32]), 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 [2] and ANSI/ISO C++ standard committee [5]. Following the principles of generic programming [37, 41] and written in a style similar to the C++ Standard Template Library (STL) [36, 42], the algorithms provided by the Parallel BGL are parameterized by the data types on which they operate. Arbitrary graph data types can be used with BGL algorithms; in particular,

Breadth-first search	Dijkstra's shortest paths	Floyd-Warshall all-pairs shortest paths
Depth-first search	Bellman-Ford shortest paths	Johnson's all-pairs shortest paths
King ordering	Push-Relabel max flow	Kruskal's minimum spanning tree
Transpose	Sequential vertex coloring	Dynamic connected components
Topological sort	Prim's minimum spanning tree	Strongly connected components
Sloan ordering	Biconnected components	Incremental connected components
Gursoy-Atun layout	Kamada-Kawai spring layout	Fruchterman-Reingold force directed layout
Transitive closure	Edmunds-Karp max flow	Brandes betweenness centrality
Connected components	Minimum degree ordering	Reverse Cuthill-Mckee ordering
Articulation points	Smallest last vertex ordering	

Table 1: Algorithms currently implemented in the sequential BGL.

Dijkstra single-source shortest path	Breadth-first search	Boruvka Minimum spanning tree
Dehne-Götz Minimum spanning tree	Depth-first search	Connected components
Crauser et al. single-source shortest path	Dinic Max-flow	Biconnected components
Strongly-connected components	PageRank	Graph coloring
Fructerman-Rheingold layout		

Table 2: Algorithms currently implemented in the Parallel BGL.

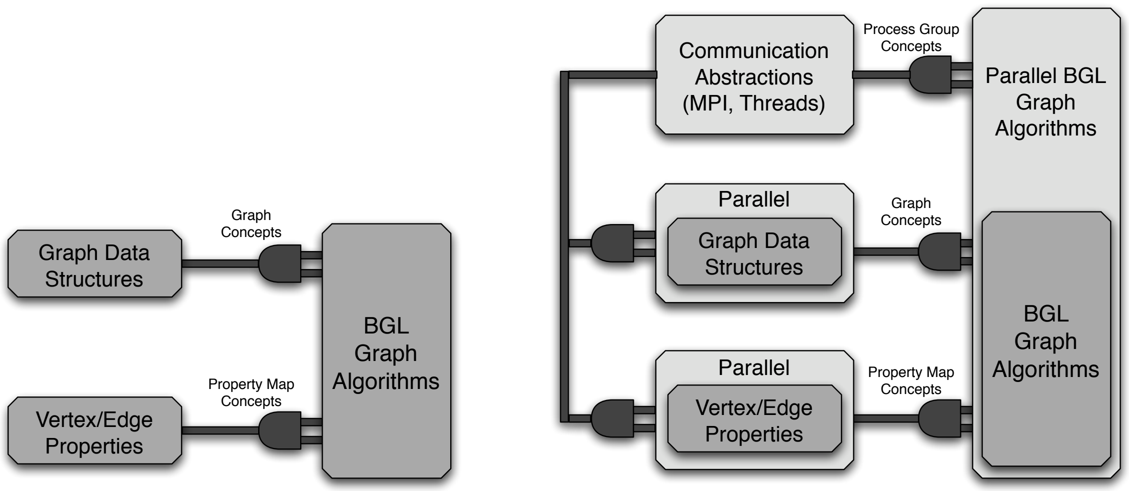
independently-developed third-party graph types can be used without the need to modify the BGL algorithms themselves. The BGL does provide its own data types, which are parameterized by the underlying storage types. This parameterization allows extensive customization of the BGL, from storing user-defined data types with the vertices and edges of a graph to completely replacing the Parallel BGL graph types with application-specific data structures, without incurring additional overhead. Table 1 lists some of the algorithms that are currently implemented in the BGL.

Following our philosophy of software libraries and reuse, we have recently developed the Parallel Boost Graph Library (Parallel BGL) [21] on top of the sequential BGL. The Parallel BGL provides data structures and algorithms for parallel computation on graphs. The Parallel BGL retains much of the interface of the (sequential) BGL upon which it is built, greatly simplifying the task of porting programs from the sequential BGL to the Parallel BGL. Because it is built on top of the sequential BGL, the Parallel BGL also retains the performance and flexibility of the underlying BGL. Table 2 lists some of the algorithms that are currently implemented within the Parallel BGL.

2 The (Parallel) Boost Graph Library

The Parallel BGL is a generic graph library written for high performance and maximum reusability, and is itself built upon the generic Boost Graph Library. The core of the BGL—sequential or parallel—is a set of generic graph algorithms, that are polymorphic with respect to the underlying graph types. BGL graph algorithms can be applied to any graph data type, even to types that are not included with the library, provided that type supplies all the functionality required by the algorithm. Additionally, BGL graph algorithms are often customizable in other ways, through visitor objects and abstract representations of vertex and edge properties (e.g., edge weights). The genericity of the BGL is such that it can (for example) operate on a LEDA [34] graph just as efficiently as it can operate on its own graph types, without requiring the user to perform any data conversion.

The Parallel BGL employs a unique, modular architecture built on the sequential BGL. Figure 1 illustrates the components in the sequential and Parallel BGL and their interactions. There are three primary kinds of interfaces in the Parallel BGL: graphs, which describe the structure of a graph that may either be stored in (distributed) memory or generated on-the-fly; property maps, which associate additional information with the vertices and edges of a graph; and process groups, which facilitate communication among distributed



(a) Boost Graph Library

(b) Parallel Boost Graph Library

Figure 1: Architectures of the sequential and parallel Boost Graph Libraries, illustrating how various abstractions (communication medium, graph data structures, graph properties) "plug into" the generic graph algorithms via concepts.

data structures and algorithms. Two of these interfaces, graphs and property maps, are inherited from—and are therefore compatible with—the sequential Boost Graph Library. These are illustrated by the dark shaded blocks in Figure 1.

The Parallel BGL components (represented by lightly-shaded blocks in Figure 1) typically wrap their sequential BGL counterparts, building distributed-memory parallel processing functionality on top of efficient sequential code. Layering a parallel library on top of a sequential library has many benefits: one need not reimplement core data structures or algorithms; potential users may have some prior knowledge of the interfaces that can be transferred from the sequential to the parallel library; and improvements to the sequential library will immediately improve the parallel library. However, the performance of the parallel library then becomes dependent on the sequential library, making it extremely important that the sequential library be both efficient and customizable. We have found that generic libraries such as the (sequential) BGL can be layered in this fashion without performance degradation, and they even allow reuse of algorithm implementations in the parallel context. The implementation of breadth-first search, Dijkstra's shortest paths, and Fruchterman-Reingold force-directed layout in the Parallel BGL, for instance, are merely invocations of the generic implementations of the sequential BGL using appropriate distributed data structures [22].

Components in the (Parallel) BGL are loosely connected via *concepts*, which provide abstraction without unnecessary performance penalties. A *concept* is essentially an interface description. Generic algorithms are written using concepts, and any data type that meets the requirements of the concept can be used with the generic algorithm. Concepts differ from abstract base classes or "interfaces" in object-oriented languages in several ways, two of which are important in the context of the BGL. First, concepts are purely compile-time entities, and unlike virtual functions their use incurs no run-time overhead. Second, concepts permit *retroactive modeling*, which means that one can provide the appropriate concept interface for a data type *without changing the data type*; this allows external data structures to be used with generic libraries such as the (Parallel) BGL.

2.1 Generic Algorithms

The generic algorithms in the (Parallel) BGL are implemented using C++ templates, which provide compile-time polymorphism with no run-time overhead. Figure 2 contains the complete implementation of the

```
template<class IncidenceGraph, class Buffer, class BFSVisitor, class ColorMap>
void
breadth_first_visit   (const IncidenceGraph& g,
                        typename graph_traits<VertexListGraph>::vertex_descriptor s,
                        Buffer& Q, BFSVisitor vis, ColorMap color)
{
  put(color, s, Color::gray());                          vis.discover_vertex (s, g);
  Q.push(s);
  while (! Q.empty()) {
    Vertex u = Q.top(); Q.pop();                         vis.examine_vertex(u, g);
    for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
      Vertex v = target(*ei, g);                         vis.examine_edge(*ei, g);
      ColorValue v_color = get(color, v);
      if (v_color == Color::white()) {                   vis.tree_edge(*ei, g);
        put(color, v, Color::gray());                    vis.discover_vertex (v, g);
        Q.push(v);
      } else {                                           vis.non_tree_edge(*ei, g);
        if (v_color == Color::gray())                    vis.gray_target (*ei, g);
        else                                             vis.black_target (*ei, g);
      }
    } // end for
    put(color, u, Color::black());                       vis.finish_vertex (u, g);
  } // end while
}
```

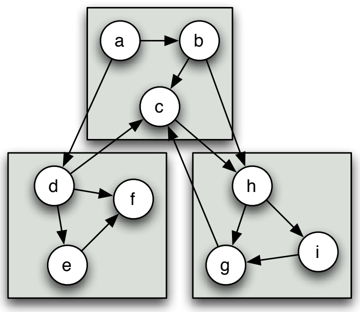
Figure 2: Generic implementation of the (sequential) breadth-first search algorithm in the BGL. The algorithm resides in the left column and the associated event points are written in the right column.

sequential breadth-first search algorithm in the BGL. The four template parameters for this algorithm correspond to the graph type itself (IncidenceGraph), the queue that will be used to store vertices (Buffer), the visitor that will be notified for various events during the breadth-first search (BFSVisitor), and the *property map* that will be used to keep track of which vertices have been seen (ColorMap).

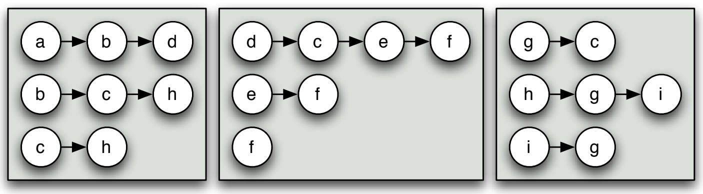
The most important feature of breadth_first_visit() is that it can operate on an arbitrary graph type, as long as the type meets certain (minimal) requirements. In particular, to be used as a graph with breadth_first_visit(), the functions out_edges(), source(), and target() must exist for the given type. These requirements are part of the Incidence Graph concept, which is documented elsewhere in greater detail [39, 40]. For types that provide the required functionality, but do not have these required functions, a simple adaptation layer can easily map from the provided functionality to the required function names. This kind of adaptation is provided in BGL and allows LEDA graphs (for example) to be used directly with BGL algorithms.

The ColorMap template parameter is also interesting because it separates the notion of the "color" of a vertex from the storage of the vertex itself. In BGL terminology, ColorMap is a *property map*, because it provides a particular property for each vertex of the graph. When initiating a breadth-first search, all vertices will be white. When a vertex is seen, it is colored gray. Once all of its outgoing edges have been visited, it is colored black. This information can be stored either inside the graph or in an external data structure, such as an array or hash table.

The remaining two template parameters, Buffer and BFSVisitor, allow further customization of the behavior of the breadth-first search algorithm. We will revisit these parameters when we discuss the implementation of (parallel) Dijkstra's algorithm in the Parallel BGL in Section 3.



(a) Distributed graph



(b) Distributed adjacency list representation

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

2.2 Graph Data Structures

In addition to generic algorithms, the (Parallel) BGL provides several configurable graph data structures. The sequential BGL provides an adjacency list representation, a more compact and efficient (but less versatile) compressed sparse row representation, and an adjacency matrix representation, any of which can be used for the vast majority of the generic algorithms in the BGL. In addition, the BGL provides an adaptation layer for various third-party graph types.

The Parallel BGL provides distributed counterparts to the adjacency list and compressed sparse row graphs of the BGL. Graphs in the Parallel BGL are distributed using a row-wise decomposition: each processor stores a disjoint set of vertices and all edges outgoing from those vertices. Figure 3 (a) shows a small graph consisting of nine vertices, distributed across three processors (indicated by the shaded rectangles). Figure 3 (b) illustrates how this graph can be represented using a distributed adjacency list in the Parallel BGL. Each processor contains several vertices. Attached to those vertices is a list of edges. For instance, vertex a on the leftmost processor has two outgoing edges, (a, b) and (a, d), so b and d are stored in its list. We note that we use the term “distributed” in a general sense. The distributed types in Parallel BGL can be used in shared-memory environments, in which case parallelism can be effected via MPI or threads.

The Parallel BGL algorithms can operate on any distributed graph type that meets the distributed graph requirements, allowing the user to choose the best data structure for her task. The distributed compressed sparse row graph uses the same distribution scheme as the distributed adjacency list shown in Figure 3 (b), but instead of maintaining separate lists for the out-edges of each vertex, the out-edge lists are packed into a single, contiguous array. The distributed compressed sparse row graph therefore requires far less memory than the distributed adjacency list and exhibits better locality, resulting in better performance. However, the distributed compressed sparse row graph requires significantly more effort to use. For example, because insertion into the middle of the edge list is $O(|E|)$, the edges must be ordered by source vertex before they are inserted. We typically use distributed compressed sparse row for benchmarking (see Section 4.4 for a performance comparison between these two graph data structures).

3 Implementing Single-Source Shortest Paths

We implemented two single-source shortest paths algorithms in the Parallel BGL, both based on Dijkstra's sequential algorithm. Dijkstra's algorithm computes shortest paths by incrementally growing a tree of shortest paths for a weighted graph $G = (V, E)$ from the source vertex s out to the most distant vertices. The primary data structure used in Dijkstra's algorithm is a priority queue of vertices that have been seen but not yet processed and ordered based on their distance $d(u)$ from the source vertex s. Prior to execution of Dijkstra's algorithm, $d(u) = \infty$ for all $u \neq s$, $d(s) = 0$, and the priority queue contains the vertex s. At each step in the computation, the algorithm removes the vertex u with the smallest $d(u)$ from the priority queue, then *relaxes* each outgoing edge (u, v). The *relax* step determines whether there is a better path from s to v via u, i.e., if $d(u) + w(u, v) < d(v)$ (where $w(u, v)$ is the non-negative weight of edge (u, v)). When a better path is found, $d(v)$ is updated with the value of $d(u) + w(u, v)$ and it is either inserted into the priority queue (if v had not previously been seen) or its position in the queue is updated. Dijkstra's algorithm terminates when the priority queue is empty.

Dijkstra's algorithm can be readily adapted for distributed memory. The graph itself is distributed using a row-wise decomposition, with each processor owning both a subset of the vertices as well as all edges outgoing from those vertices (as in the adjacency list of Figure 3). Likewise, the priority queue is distributed, with each processor's priority queue storing only those vertices owned by that processor. The processors will only remove vertices from their local priority queue, so each processor only relaxes the edges outgoing from vertices that it owns. When an edge is relaxed, the owner of the source vertex sends a message containing the target vertex, its new distance, and (optionally) the name of the source vertex to the target vertex's owner. To ensure that only the vertex u with the smallest $d(u)$ (globally) is selected, the computation is divided into supersteps: at the beginning of each superstep, the processors coordinate to determine the global minimum distance $\mu = min\{d(u) : u \text{ is queued}\}$. The processors then select a vertex u with $d(u) = \mu$, and the owner of vertex u removes it from the local priority queue and relaxes its outgoing edges. All processors then receive the messages produced by u's owner as edges are relaxed, update their local priority queues with new vertices and new distances, and the superstep completes. Successive supersteps process the remaining vertices in the shortest paths tree, one vertex per superstep. The algorithm terminates when all local priority queues are empty.

There are two obvious opportunities for parallelizing the naïvely distributed Dijkstra's algorithm. The first opportunity is to relax all of the edges outgoing from the active vertex u in parallel, effectively parallelizing the inner loop of Dijkstra's algorithm. However, the speedup that we can attain by parallelizing this loop is limited by the number of outgoing edges from a given vertex. With a row-wise distribution of the graph data structure, only a single processor has direct access to the outgoing edges of u. Therefore, parallelizing in this manner requires replication or distribution of the outgoing edges for the active vertex u, Even within a shared-memory system, the effect of parallelizing only this inner loop is limited. In many real-world graphs, the average out-degree is a relatively small constant, so parallelizing the relaxation of outgoing edges is not likely to yield good scalability except in the case of extremely dense graphs. For this reason, the current implementation of Parallel BGL therefore does not include parallel relaxation of edges. We are studying the inclusion of this parallel edge relaxation in future versions of the Parallel BGL.

The second opportunity for parallelizing Dijkstra's algorithm is to remove several vertices from the priority queue simultaneously, relaxing their edges in parallel. At the beginning of each superstep, the distributed-memory formulation of Dijkstra's algorithm determines the global minimum distance, μ, and selects a single vertex u to relax. Instead, we could allow every vertex u with $d(u) = \mu$ to be removed from the priority queue (and its outgoing edges relaxed) within the superstep. The scalability of the algorithm is then tied to the number of vertices that can be removed from the distributed priority queue within a single superstep and how well those vertices are distributed among the processors. Ideally, each superstep would remove a large number of vertices, evenly distributed among the processors. Unfortunately, real-world graphs rarely have a large number of vertices with the same distance from the source vertex, so the degree of parallelism we can extract from this direct parallelization is limited.

To expose more parallelism in Dijkstra's algorithm we need to remove more vertices from the priority queue in each superstep. This is accomplished by allowing the removal of vertices whose distances exceed μ. When removing a vertex u with $d(u) > \mu$ and relaxing its outgoing edges, it is possible that another processor will find a better route from the source s to u. When a better route is found, u will need to be *re*-inserted into the priority queue so that its edges will be relaxed again, but with a smaller value of $d(u)$. Thus, there is a trade-off between exposing more parallelism (by removing more vertices in each superstep) and avoiding unnecessary work (by limiting how many re-insertions will occur). We have implemented two variations on Dijkstra's algorithm that use different strategies to decide which vertices u with $d(u) > \mu$ should be considered.

```
struct dijkstra_bfs_visitor {
  template<typename Edge, typename Graph>
  void tree_edge(Edge e, Graph& g) {
    if (distance(source(e, g)) + weight(e) < distance(target(e, g)))
        distance(target(e, g)) = distance(source(e, g)) + weight(e);
  }

  template<typename Edge, typename Graph>
  void gray_target(Edge e, Graph& g) {
    if (distance(source(e, g)) + weight(e) < distance(target(e, g)))
      Q.update(target(e, g), distance(source(e, g)) + weight(e));
  }
};
```

Figure 4: Breadth-first search visitor that relaxes each outgoing edge and updates the queue appropriately. This visitor is used by both the sequential and parallel formulations of Dijkstra's algorithm in the (Parallel) BGL.

3.1 Implementation Strategy

Dijkstra's algorithm can be viewed as a modified breadth-first search. A breadth-first search is typically implemented using a first-in first-out (FIFO) queue. Breadth-first search initially places the start vertex s into the queue. At each step, it extracts a vertex from the head of the queue, visits its outgoing edges, and places all new target vertices into the tail of the queue.

Dijkstra's algorithm changes breadth-first search in two ways. First, the FIFO queue is replaced with the priority queue. Second, when visiting the outgoing edges for the active vertex, Dijkstra's algorithm relaxes those edges and updates the ordering in the priority queue. Within the (sequential) Boost Graph Library, Dijkstra's algorithm is implemented as a call to breadth-first search that replaces the FIFO queue with a relaxed heap and provides a visitor that relaxes edges. A simplified version of the visitor is shown in Figure 4. It uses two events to update the queue: tree_edge() is invoked when breadth-first search traverses an edge whose target has not yet been seen, hence the edge is part of the breadth-first spanning tree, and gray_target(), which is invoked when the target of an edge has been seen but not processed. The visitor functions for both events "relax" edges, although only the latter needs to update the priority queue directly. Using this visitor, Dijkstra's algorithm is implemented as a simple call to breadth_first_visit():

```
dijkstra_shortest_paths
  (Graph &g, Vertex source)
{
    relaxed_heap <Vertex> Q;            // Priority queue
    dijkstra_bfs_visitor    bfs_vis (Q); // Visitor that updates the priority queue
    breadth_first_visit   (graph, source, Q, bfs_vis );
}
```

The Parallel BGL contains a distributed-memory parallel breadth-first search implementation, upon which we have built the parallel Dijkstra variants. Both variants use the visitor shown in Figure 4, but they provide different distributed priority queue implementations, each using a different heuristic to determine which vertices should be removed in a superstep. Implementing other distributed priority queue heuristics for Dijkstra's algorithm is relatively simple with the Parallel BGL: one need only implement a queue that models the new heuristics and then call breadth_first_visit() with an instance of the new queue and the Dijkstra visitor from Figure 4, as shown above.

3.2 Eager Dijkstra's Algorithm

The "eager" Dijkstra's algorithm uses a simple heuristic to determine which vertices should be removed in a given superstep. The eager algorithm uses a constant lookahead factor λ, and in each superstep the processors remove every vertex u such that $d(u) \leq \mu + \lambda$, ordered by increasing values of $d(u)$.

When $\lambda = 0$, the eager algorithm is equivalent to the naïve parallelization of Dijkstra's algorithm, and exposes very little parallelism. Larger values of λ can expose more parallelism, but might result in a work-inefficient algorithm if too many vertices need to be re-inserted into the priority queue. When $\lambda = min\{weight(e) | e \in E\}$, we can expose additional parallelism without introducing any re-insertions. The optimal value for λ depends on the graph density, shape, and weight distribution, among other factors. We provide an experimental evaluation of the effect of λ on performance in Section 4.2.

The eager Dijkstra distributed queue from the Parallel BGL is responsible for implementing both the eager lookahead behavior of the algorithm and for managing synchronization among the processors. In addition to push() and pop(), it implements empty() and update() operations. Whenever the push() or update() operation is invoked, a message is sent to the owning process. These messages are only processed at the end of each superstep, which occurs inside the empty() method. empty() returns false so long as the local queue contains at least one vertex u such that $d(u) \leq \mu + \lambda$. When no such vertex exists, the processor synchronizes with all of the other processors, receiving "push" messages and finally recomputing the global minimum value, μ. Note that empty() only returns false when *all* priority queues on all processors are empty, signaling termination of the algorithm. The design and implementation process used to arrive at this formulation of a distributed queue, and its use with the sequential breadth_first_visit() implementation to effect a parallel algorithm, is further described in [22].

3.3 Crauser et al.'s Algorithm

The parallel Dijkstra variant due to Crauser et al. [15] uses more precise heuristics to increase the number of vertices removed in each superstep without causing any re-insertions. The algorithm uses two separate criteria, the OUT-criterion and the IN-criterion, which can be combined to determine which vertices should be removed in a given superstep. Unlike the eager algorithm, there are no parameters that need to be tuned.

The OUT-criterion computes a threshold L based on the weights of the outgoing edges in the graph. L is given the value $min\{d(u) + weight(u, w) : u$ is queued and $(u, w) \in E\}$. Any vertex v with $d(v) < L$ can safely be removed from the queue, because L bounds the smallest distance value $d(v)$ that can be achieved by relaxing the outgoing edges of any queued vertex.

The IN-criterion computes a threshold based on the incoming edges. If $d(v) - min\{weight(u, v) : (u, v) \in E\} \leq \mu$ (where μ is the global minimum) for a queued vertex v, then v can safely be removed from the queue, because there is no vertex in the queue with an outgoing edge to v that could be relaxed.

The OUT- and IN-criteria can be used in conjunction, so that each superstep removes all vertices that meet either criterion. On random graphs with uniform edge weights, each superstep will remove on average $\mathcal{O}(n^{2/3})$ vertices with high probability [15].

In the Parallel BGL, we have implemented Crauser et al.'s algorithm by creating a new distributed priority queue that applies the OUT- and IN-criteria. This distributed priority queue is very similar to the eager Dijkstra queue. However, the Crauser et al. priority queue contains three relaxed heaps for each processor, ordered by $d(v)$, $d(u) + weight(u, w) : u$ is queued and $(u, w) \in E$ (for the OUT-criterion), and $d(v) - min\{weight(u, v) : (u, v) \in E\}$ (for the IN-criterion). The three relaxed heaps are maintained simultaneously, so that both the IN- and OUT-criteria can be used together. The implementation of Crauser et al.'s algorithm is a single call to breadth_first_visit(), using the new distributed priority queue and the Dijkstra visitor from Figure 4.

3.4 Using Dijkstra's Algorithm

The implementations of Dijkstra's algorithm in the (Parallel) BGL are provided by the function template dijkstra_shortest_paths(), which can operate on both distributed and non-distributed graphs. The algorithm is polymorphic based on the graph type, and can be invoked for any suitable graph from source vertex source and with the specified edge weights:

```
dijkstra_shortest_paths(graph, source, weight_map(edge_weights));
```

The actual implementation of Dijkstra's algorithm selected at compile time depends on what kind of graph is provided in the call. For instance, graph could be a non-distributed adjacency list, in which case the sequential Dijkstra's algorithm will be used:

```
adjacency_list<vecS, vecS, directedS> graph; // non−distributed adjacency list
dijkstra_shortest_paths(graph, source, weight_map(edge_weights)); // sequential Dijkstra's
```

The same sequential Dijkstra's algorithm can instead be used with a (non-distributed) compressed sparse row graph, providing more compact storage and potentially improving algorithm performance:

```
compressed_sparse_row_graph<directedS> graph; // non−distributed CSR graph
dijkstra_shortest_paths(graph, source, weight_map(edge_weights)); // sequential Dijkstra's
```

On the other hand, if graph were a distributed graph, dijkstra_shortest_paths() would instead apply Crauser et al.'s algorithm for distributed-memory parallel shortest paths. The graph in this case is an instance of adjacency_list that uses a distributedS selector, indicating that the vertices should be distributed across the processors. The distributedS selector is parameterized by the process group type, which indicates how parallel communication will be performed. In this case, we have used the bsp_process_group implemented over MPI. The fact that the graph is distributed is encoded within the type of the graph itself, allowing the Parallel BGL to perform a compile-time dispatch to select a distributed algorithm. The call to dijkstra_shortest_paths(), and the majority of the code leading up to the call, remains unchanged when one moves from the non-distributed graph types of the (sequential) BGL to the distributed graph types of the Parallel BGL.

```
adjacency_list<vecS, distributedS<vecS, mpi::bsp_process_group>, directedS> graph; // distributed...
dijkstra_shortest_paths(graph, source, weight_map(edge_weights)); // Crauser et al.'s for distributed memory
```

To use the eager Dijkstra algorithm in lieu of Crauser et al.'s algorithm for a distributed graph, the user need only supply a lookahead value λ. Note that in the following example, the period separating the weight_map parameter from the lookahead parameter is not an error; rather, it is a form of named parameters used within both Boost Graph Libraries. Here, we illustrate the application of the eager Dijkstra algorithm to a distributed graph stored in compressed sparse row format:

```
compressed_sparse_row_graph<directedS, void, void, no_property, distributedS<mpi::bsp_process_group> >
  graph; // distributed CSR graph
dijkstra_shortest_paths(graph, source, weight_map(edge_weights).lookahead(15)); // Eager Dijkstra's
```

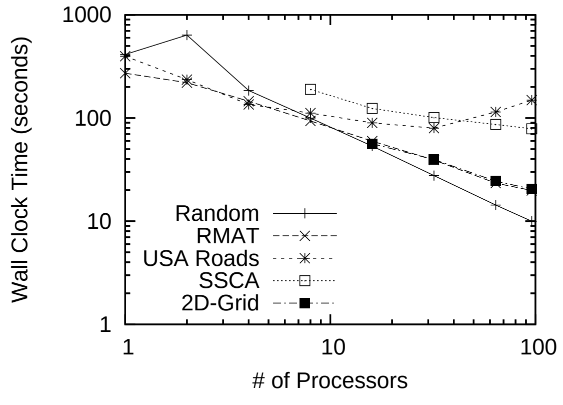
4 Evaluation

For this paper we evaluated the performance and scalability of our single-source shortest paths implementations using various synthetic and real-world graphs; the sequential performance of the BGL has been demonstrated previously [31, 32]. All performance evaluations were performed on the Indiana University Computer Science Department's research cluster Odin. Odin consists of 128 compute nodes connected via Infiniband. Each node contains 4GB of dual-channel PC3200 (400 MHz) DDR-DRAM with two 2.0GHz AMD Opteron 246 processors (1MB Cache) running Red Hat Enterprise Linux WS Release 4. For our tests we have left one processor idle on each node. The Parallel BGL tests were compiled using a pre-release version of Boost 1.34.0 [11] (containing the sequential BGL) and the latest development version of the Parallel BGL [21]. All programs were compiled with version 9.0 of the Intel C++ compiler with optimization flags −O3 −xW −tpp7 −ipo −i_dynamic −fno−alias. All MPI tests use version 1.1 of Open MPI [18] with the mvapi module.

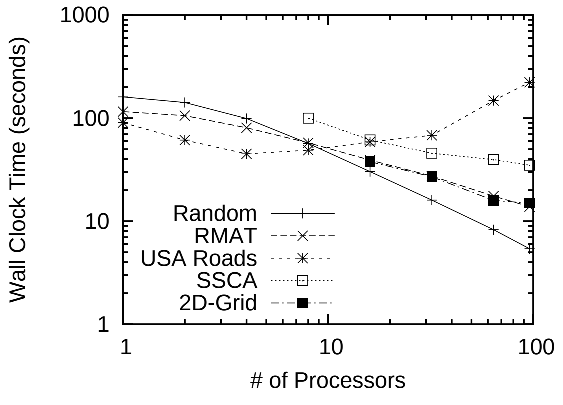
We performed our experiments with real-world and synthetic data from the 9th DIMACS Implementation Challenge [1]. We used several different kinds of synthetic graphs generated using GTgraph [9], each of which exhibits different graph properties. Additionally, we use real-world graph data for the network of roads in the United States. The graphs we have used in this evaluation are:

Random Graphs as produced with the GTgraph random generator, using random model 1 (n,m) graphs. These graphs tend to have very little structure.

RMAT Graphs as produced by the GTgraph implementation of the RMAT [12] power-law graph algorithm.



(a) Crauser et al.'s Algorithm



(b) Eager Dijkstra Algorithm

Figure 5: Strong scalability for the two variants of parallel Dijkstra's algorithm, using fixed-size graphs with $\sim 24M$ vertices and $\sim 58M$ edges.

SSCA Graphs as generated by the GTgraph implementation of the HPCS SSCA # 2 benchmark [8]. This algorithm begins by producing cliques of size uniformly distributed between 1 and n (we set n to 8 for weak scaling tests and 5 for comparison against the USA Roads data) and then adds inter-clique edges with probability p (we set p to 0.5 for weak scaling tests and 0.25 for comparison against the USA Roads data). These graphs also tend to have a large number of multiple edges.

USA Roads Complete U.S. Road network from the UA 2000 Census TIGER/Line data [43]. The graph is very sparse; it has $\sim$ 24 million vertices and $\sim$ 58 million edges.

European Roads Road networks of 17 European countries from the PTV Europe data [38]. The graph is very sparse; it has $\sim$ 19 million vertices and $\sim$ 23 million edges.

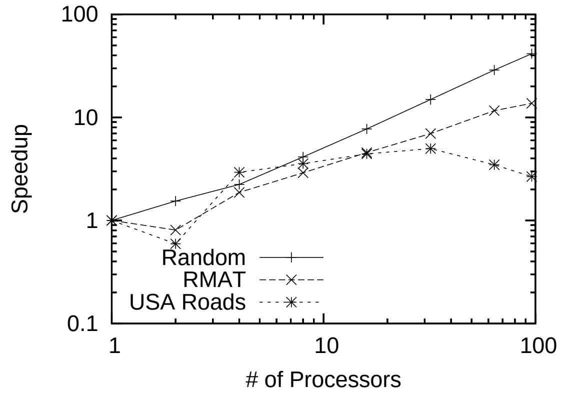
2D Grid Square two dimensional grids were included as a simple and well-structured test case.

Erdös-Renyi Random graphs generated using the Parallel BGL random graph generator.

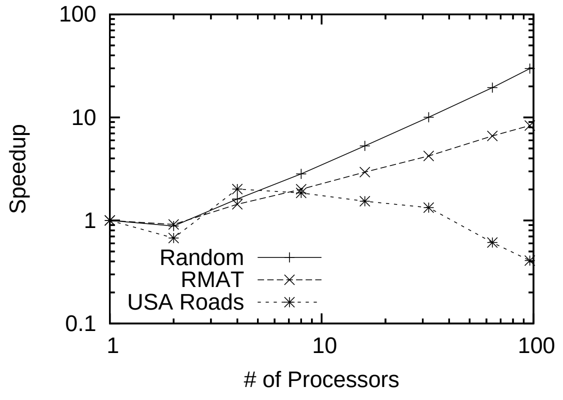
4.1 Strong Scaling

To understand how well the parallel implementations of Dijkstra's algorithm in the Parallel BGL scale as more computational resources are provided, we evaluated the performance of each algorithm on fixed-size graphs. We generated synthetic graphs that are comparable in size to the USA road network, with $\sim 24M$ vertices and $\sim 58M$ edges. Both the random and RMAT graphs were created by specifying the number of vertices and edges parameters, with all other parameters set to defaults. The SSCA graph was generated by specifying the number of vertices, setting the maximum clique size to five, setting the probability of inter-clique edges to 0.25 and setting the maximum edge weight to 100. Note that with the SSCA graph, we were unable to generate graphs as sparse as the USA road network; to do so would require an unrealistically small maximum clique size. Thus, the SSCA graph contains about $148M$ edges. For the Eager Dijkstra algorithm, we have selected a lookahead value $\lambda = 8$ based on experimental evidence gathered for random graphs (Figure 8).

Figure 5 illustrates the strong scalability of the two parallel Dijkstra implementations. The random data appear to scale linearly with both algorithms, though they exhibit minor paging with two processors using the Crauser et al. algorithm. This is the result of a relatively uniform distribution of work due to the uniform nature of the random graph. The RMAT data also scale very linearly, though they exhibit less speedup than the random data. We speculate this is likely due to a less balanced work distribution caused by large variances in the degree of the vertices. The USA road data begin to scale inversely at 8 processors using the eager Dijkstra algorithm for reasons that are examined in section 4.2. The poor scalability of the USA

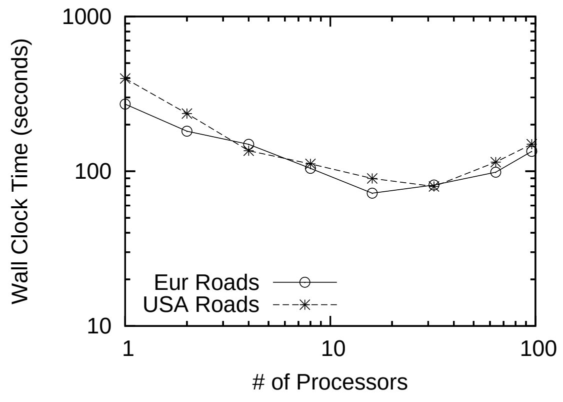


(a) Crauser et al.'s Algorithm

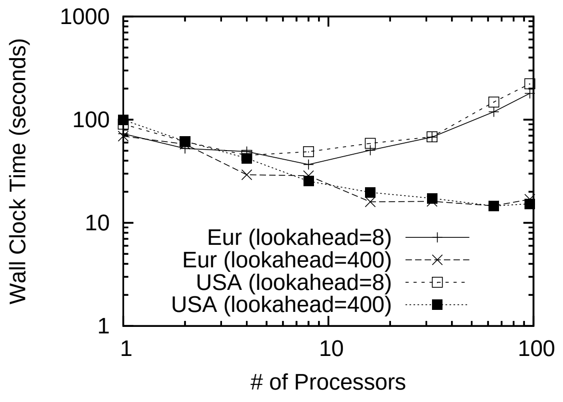


(b) Eager Dijkstra Algorithm

Figure 6: Parallel speedup for the two variants of parallel Dijkstra's algorithm, using fixed-size graphs with $\sim 24M$ vertices and $\sim 58M$ edges.

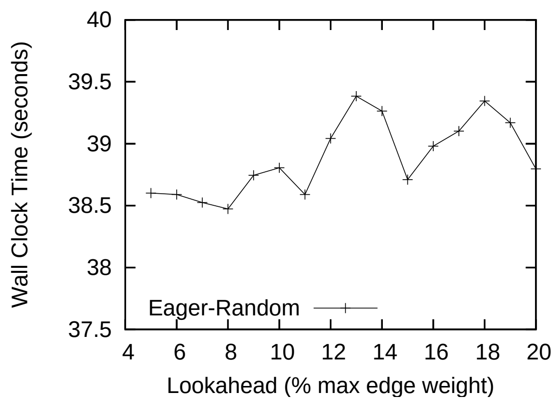


(a) Crauser et al.'s Algorithm

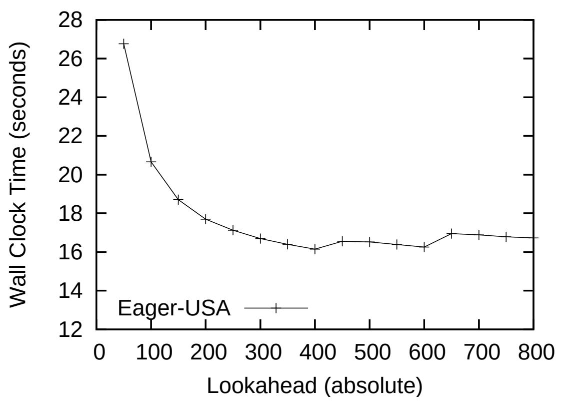


(b) Eager Dijkstra Algorithm

Figure 7: Strong scalability for the two variants of parallel Dijkstra's algorithm, using the USA and European road networks



(a) Random graph



(b) USA road network

Figure 8: Effect of lookahead value on the performance of the eager Dijkstra's algorithm, using fixed-size graphs with $\sim 24M$ vertices and $\sim 58M$ edges.

road data using the Crauser et al. algorithm is most likely due to the conservative nature of the algorithm preventing it from removing a sufficient number of vertices in each superstep. When insufficient numbers of vertices are removed, the result is more communication rounds which increase runtime. Removing insufficient numbers of vertices in a superstep can also lead to load imbalances, further decreasing performance. The SSCA data also scale linearly, though the speedup is not as significant as on the other synthetic data. The SSCA data contain more edges than the other synthetic graph types, but a significant portion of these edges are duplicates in the sense that they have the same source and target. The shortest path algorithms still have to consider the duplicate edges; moreover, the larger number of edges overall leads to a larger memory footprint. Without duplicate edges, the SSCA graph is only slightly denser than the random and RMAT graphs, yet it exhibits the same scaling behavior. This illustrates that the poor scaling of the SSCA graphs is indeed a product of their structure, not their density. Finally, the two dimensional grid data also scales linearly as expected.

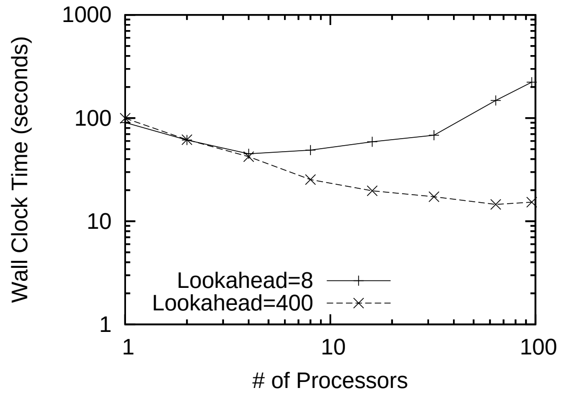
In order to determine if the performance observed was typical of structured data sets such as road networks, both variants of parallel Dijkstra's algorithm were also run on the European road network data. Figure 7 shows similar scalability results to the USA road network data. This supports our theory that the structure of the road network data limits available parallelism.

It should be noted that all of these graphs are relatively small compared to the size of problems the Parallel BGL is capable of solving. The Parallel BGL does introduce some communication overhead in order to manage the distributed data structures, therefore for small problem sizes the sequential BGL may be a more appropriate choice. However for problem sizes too large to fit in core on a single machine, the Parallel BGL is a fast, efficient, and scalable alternative. For problems that do fit in core on a single machine the Parallel BGL may still be able to provide a faster solution using small numbers of processors.

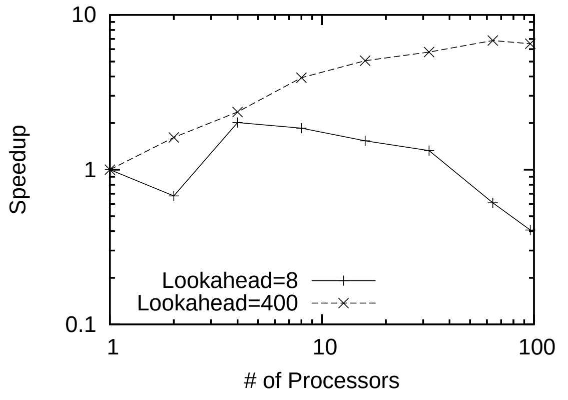
4.2 Eager Dijkstra Lookahead Factor

To determine an appropriate lookahead value to use in our scalability tests we evaluated a range of options on a random graph generated using the GTgraph [9] generator. This graph is comparable in size to the USA road network. We chose a random graph in order to reduce any bias the particular structures of the other graphs may have had on the lookahead value. Previous tests using the Parallel BGL [23] have indicated that optimal lookahead values for random graphs tend to be around 10% of the maximum edge weight in the graph so we examined values around 10%.

Figure 8 shows a minimum at a lookahead value of 8, thus this value was chosen for our scalability results. Examining the strong scalability results in Figure 5 obtained using this lookahead value indicates that the USA road network data scale very poorly. We speculated that this was likely the result of a poor choice of lookahead value for this particular graph, so we tried a variety of alternate lookahead values. Figure 8



(a) Wall clock time



(b) Parallel speedup

Figure 9: Results of running the eager Dijkstra's algorithm with two different lookahead values on the USA road network

shows that the optimal lookahead value for the USA road network was 400, much larger than our original approximation of 8.

Figure 9 illustrates the performance difference that the choice of a lookahead value can have on the eager Dijkstra's algorithm. Optimal lookahead values vary not only with graph size and structure, but also with graph shape, density, and edge weight distribution. We theorize that the poor scaling behavior of the USA road network was due to sparse regions containing many edges with large weights. Traversing high-weight edges in these sparse regions may require several algorithm iterations if the lookahead value is small. Each iteration is relatively expensive due to the all-to-all communication that needs to occur between each BSP superstep and thus increasing the number of iterations causes severe performance degradation. Conversely if the lookahead value is large enough to cause these edges to be explored in a single iteration, then many iterations may be saved and runtime significantly reduced.

Figure 9 shows that using an appropriate lookahead value yields much better scalability on the USA road network. The parallel speedup begins to taper off around 16 processors because the data set is too small to provide adequate local work to overcome the communication overhead. Given a larger data set the shortest paths algorithms in the Parallel BGL should scale well up to hundreds of processors.

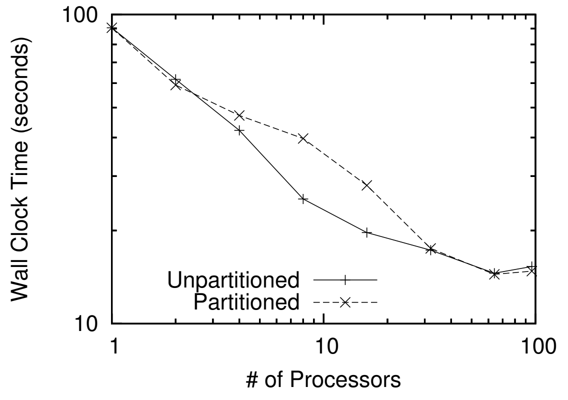
4.3 Graph Partitioning

To evaluate the effects of data distribution on algorithm performance, we applied a data partition to the USA road network. We computed this partition using the k-way METIS [28] partitioning program, but memory pressure prevented us from using edge weights or vertex coordinates in our partitioning. We ran both the Crauser et. al. algorithm and Eager Dijkstra algorithm on the partitioned graphs and compared the results to the unpartitioned version. Figure 10 provides the comparison between the partitioned and unpartitioned USA road network. Somewhat surprisingly, the partition was not beneficial, and in some cases was detrimental. We are currently investigating the cause.

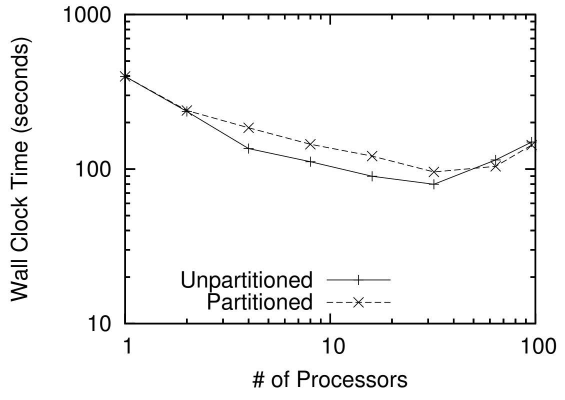
4.4 Graph Data Structures

The Parallel BGL offers several graph data structures including an adjacency list and compressed sparse row (CSR) graph. Changing the data structures used by an algorithm can mean the difference between fitting the algorithm's working set in core and paging. Additionally, more compact data structures often gain a performance advantage from cache reuse. Because more data elements can be kept in cache the likelihood of finding a data element there is higher. Compact data structures such as CSR also have drawbacks, in this case $O(|E|)$ cost for edge insertion, which may require the use of more verbose data structures in some cases.

Figure 11 shows that the CSR graph representation out-performs the adjacency list representation; this

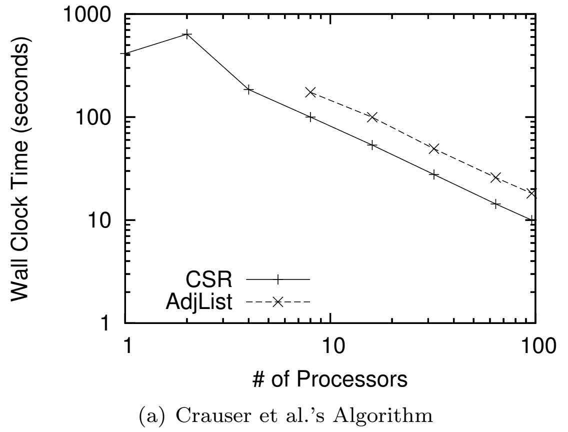


(a) USA road network Eager Dijkstra algorithm

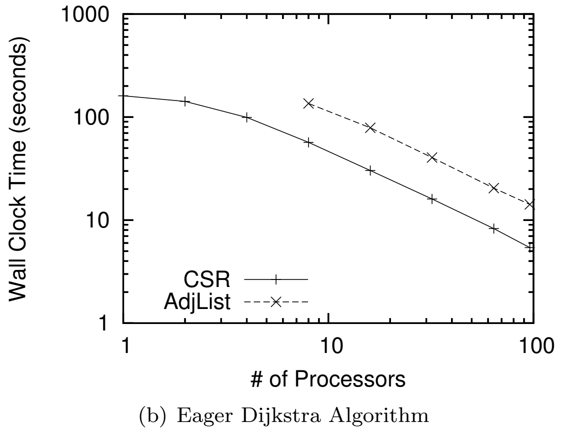


(b) USA road network Crauser et. al. algorithm

Figure 10: Parallel Dijkstra run times on METIS partitioned and unpartitioned USA road network

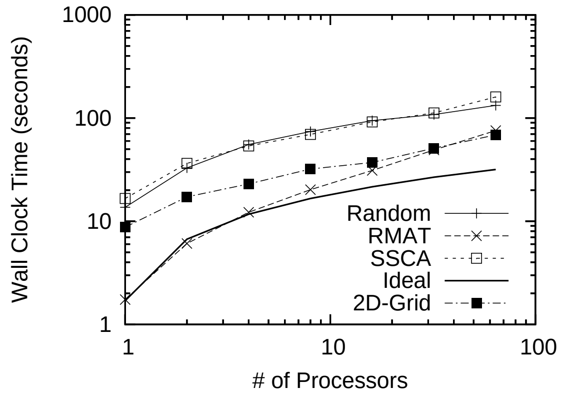


(a) Crauser et al.'s Algorithm

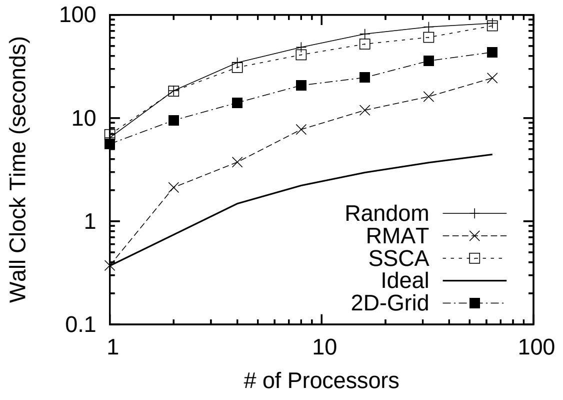


(b) Eager Dijkstra Algorithm

Figure 11: Effect of graph data type on the performance of the two variants of parallel Dijkstra's algorithm.



(a) Crauser et al.'s Algorithm



(b) Eager Dijkstra Algorithm

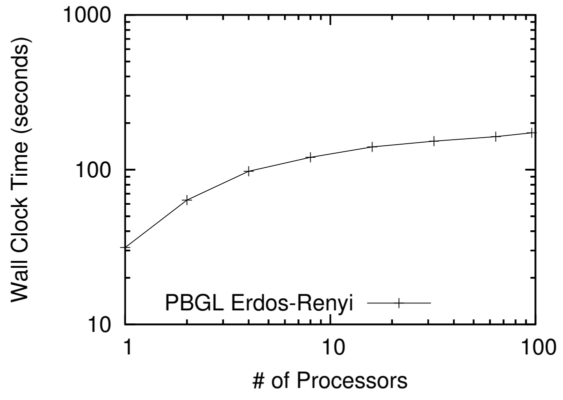
Figure 12: Weak scalability for the two variants of parallel Dijkstra's algorithm, using graphs with an average of $\sim 1M$ vertices and $\sim 10M$ edges per processor (2D-Grid graphs have $\sim 4M$ edges per processor).

can be attributed to its more compact size and efficient access methods. The data for the Adjacency List graph type are missing for fewer processors because of the higher memory footprint of that graph type. This figure also shows that there is a constant performance overhead for the less-compact Adjacency List which may be explained by cache effects and indirection. The CSR graph does exhibit some out of core behavior on two processors for the Crauser et al. algorithm. The eager Dijkstra algorithm does not exhibit this behavior as it uses slightly less memory than the Crauser et al. algorithm and thus was able to fit entirely into core and avoid paging. These results demonstrate the importance of choosing appropriate data structures when implementing an algorithm. Fortunately, the concepts in the Parallel BGL simplify making these choices by explicitly specifying the requirements that data structures must model.

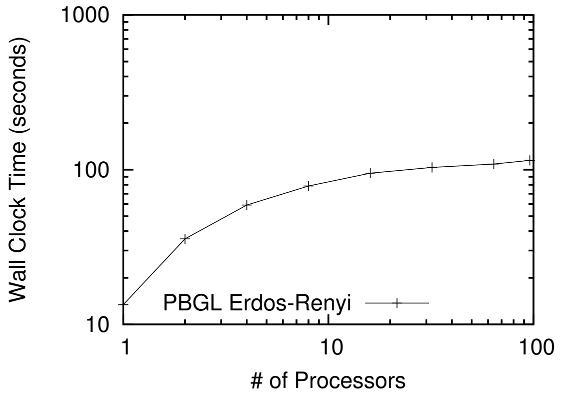
4.5 Weak Scaling

To understand how the parallel implementations of Dijkstra's algorithm scale as the problem size scales, we evaluated the performance of each algorithm on graphs where $|V| \propto |E| \propto p$ (where p is the number of processors). We generated random and RMAT graphs with $\sim 1M$ vertices and $\sim 10M$ edges per processor and SSCA graphs with $\sim 1M$ vertices per processor and as close to $\sim 10M$ edges per processor as possible. Figure 12 illustrates the run times on the GTgraph graphs. We also wanted to generate weak scaling results for the largest possible graph we could fit in core. Reading graphs from disk and generating them using the sequential generator was very expensive so we used the Parallel BGL Erdös-Renyi generator to generate graphs similar to the GTgraph random graphs in core. These graphs had $\sim 2.5M$ vertices and $\sim 12.5M$ edges per processor which results in a maximum graph size of $\sim 240M$ vertices and $\sim 1.2B$ edges on 96 processors. Figure 13 shows the run times for the weak scaling tests on these graphs produced with the Parallel BGL Erdös-Renyi generator. All weak scaling results use a lookahead value $\lambda = 8$.

These experiments show that the runtime increases even though the amount of data per processor remains constant. This is because the amount of work performed by the Crauser et al. algorithm is $O(|V|\log|V| + |E|)$ [15]. As we vary $|V|$ linearly with the number of processors the amount of work increases faster than the number of processors. This yields more work per processor which gives rise to the sub-linear scaling exhibited in Figure 12. In fact, the amount of work per processor is equal to $\frac{|V|}{p} \times \log \frac{|V|}{p}$ where p is the number of processors. In our weak scaling experiment $|V| \propto p$ so the work per processor is proportional to $\log(|V|)$. The eager Dijkstra's algorithm has a similar $\log(|V|)$ factor in the amount of work performed due to the priority queue operations and thus it is reasonable to presume that it will scale sub-linearly as well. Some curve-fitting to the weak scaling data showed that a logarithmic curve does fit the data better than any other function (radical, linear, etc.). This suggests that our weak scaling performance closely approximates the shape of the theoretical maximum though the constant factors may differ.



(a) Crauser et al.'s Algorithm



(b) Eager Dijkstra Algorithm

Figure 13: Weak scalability for the two variants of parallel Dijkstra's algorithm, using BGL-generated graphs with an average of $\sim 2.5M$ vertices and $\sim 12.5M$ edges per processor.

5 Related Work

The CGM*graph* library [13, 14] 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) [16] 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 previously shown that the Parallel BGL's implementation of connected components is at least an order of magnitude faster than CGM*graph*'s implementation [22], which we believe is due to generic programming's dual focus on genericity and efficiency. However, CGM*graph* does not provide any implementations of parallel shortest paths.

The ParGraph library [24] shares many goals with the Parallel BGL. Both libraries are based on the sequential BGL and aim to provide flexible, efficient parallel graph algorithms. 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. ParGraph does not provide any implementations of parallel shortest paths.

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* [44]), effectively eliminating the cost of any abstractions we have introduced. STAPL includes a distributed graph container with several algorithms, but it is unclear whether any parallel shortest paths algorithms have been implemented.

6 Conclusion

The Parallel Boost Graph Library provides flexible distributed graph data structures and generic algorithms. Using the facilities of the Parallel BGL, we implemented two parallel, distributed-memory variants of Dijkstra's algorithm for single-source shortest paths. By building on the pre-existing distributed-memory breadth-first search, the sequential implementation of Dijkstra's algorithm in the (sequential) BGL, and reusing the Parallel BGL's data structures, we were able to implement parallel Dijkstra's algorithm with a relatively small amount of effort and evaluate performance with several different graph types. The results showed that our solutions are computationally efficient and scalable.

Naturally, the scalability of a graph algorithm depends on the structure of the graph on which it operates. Our generic, flexible library scales well on unstructured graphs both in terms of parallel speedup as well as problem size. Problem size scaling is limited by the superlinear complexity of the Crauser and Eager Dijkstra algorithms. Graphs with grid-like structures do not scale as well as unstructured graphs.

We expect to extend the work in this paper in a number of ways. In the short term, we intend to implement additional shortest paths algorithms (e.g., Δ-stepping [35]) and continue to refine the implementations we already have, in order to capture the state of the art in distributed-memory parallel algorithms. In the longer term, we will extend the Parallel BGL to shared-memory parallel processing and take advantage of hybrid models for clusters of SMPs. To facilitate rapid prototyping and development of new sequential and parallel graph algorithms, we are also refining our Python interfaces to sequential and Paralell BGL. Ultimately, our goal is to continue to develop the (Parallel) BGL as a robust platform for parallel graph algorithm and data structure research.

7 Acknowledgements

This work was supported by a grant from the Lilly Endowment and by NSF grants EIA-0131354 and EIA-0202048.

References

[1] 9$^{\text{th}}$ DIMACS implementation challenge - shortest paths. `http://www.dis.uniroma1.it/~challenge9/`, March 2006.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond.* Addison-Wesley, 2004.

[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] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. 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] M. 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.

[6] D. Bader, G. Cong, and J. Feo. A comparison of the performance of list ranking and connected components algorithms on SMP and MTA shared-memory systems. Technical report, October 2004.

[7] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray MTA-2. In *Proceedings of 35th International Conference on Parallel Processing*, August 2006.

[8] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. Technical report, 2005.

[9] D. A. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators. `http://www-static.cc.gatech.edu/~kamesh/GTgraph/`, February 2006.

[10] E. G. Boman, D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, and F. Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *Lecture Notes in Computer Science*, volume 3648, pages 241–251, August 2005.

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

[12] D. Chakrabarti, Y. Zhan, and C. Faloutsos. Rmat: A recursive model for graph mining. In *Proceedings of 4th International Conference on Data Mining*, April 2004.

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

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

[15] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer, 1998.

[16] F. Dehne, A. Fabri, and A. 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.

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

[18] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

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

[20] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, Second Edition.* Addison-Wesley, 2003.

[21] D. Gregor, N. Edmonds, B. Barrett, and A. Lumsdaine. The Parallel Boost Graph Library. `http://www.osl.iu.edu/research/pbgl`, 2005.

[22] D. Gregor and A. 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)*, pages 423–437, October 2005.

[23] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[24] F. Hielscher and P. Gottschling. ParGraph. `http://pargraph.sourceforge.net/`, 2004.

[25] W. Hohberg. How to find biconnected components in distributed networks. *J. Parallel Distrib. Comput.*, 9(4):374–386, 1990.

[26] J. Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

[27] D. B. Johnson and P. T. Metaxas. A parallel algorithm for computing minimum spanning trees. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 363–372, 1992.

[28] Karypis and Kumaer. *METIS*, 1998. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

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

[30] 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.

[31] L.-Q. Lee, J. Siek, and A. Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *ISCOPE'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[32] L.-Q. Lee, J. Siek, and A. Lumsdaine. The Generic Graph Component Library. In *Proceedings of OOPSLA'99*, 1999.

[33] W. C. McLendon III, B. Hendrickson, S. Plimpton, and L. Rauchwerger. Identifying strongly connected components in parallel. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Compututing*, March 2001.

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

[35] U. Meyer and P. 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.

[36] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 2nd edition, 2001.

[37] D. R. Musser and A. A. Stepanov. Generic programming. In P. P. 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.

[38] PTV Europe. European road graphs. `http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php`.

[39] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[40] J. Siek, A. Lumsdaine, and L.-Q. Lee. *Boost Graph Library.* Boost, 2001. `http://www.boost.org/libs/graph/doc/index.html`.

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

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

[43] U.S. Census Bureau. UA census 2000 TIGER/Line files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`.

[44] T. L. Veldhuizen and D. 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.

[45] S. White, J. O'Madadhain, D. Fisher, and Y.-B. Boey. Java Universal Network/Graph framework. `http://jung.sourceforge.net/`, 2004.

Appendices

A Selected Core Test Output

```
f ../inputs/USA−road−d/USA−road−d.NY.gr ../inputs/USA−road−d/USA−road−d.NY.ss
g 264346 733846 1 36946
t 54.976605
v 264346.000000
i 281775.125000
f ../inputs/USA−road−d/USA−road−d.COL.gr ../inputs/USA−road−d/USA−road−d.COL.ss
g 435666 1057066 1 137384
t 89.521994
v 435665.968750
i 414065.000000
f ../inputs/USA−road−d/USA−road−d.FLA.gr ../inputs/USA−road−d/USA−road−d.FLA.ss
g 1070376 2712798 1 214013
t 267.791351
v 1070376.000000
i 1075835.500000
f ../inputs/USA−road−d/USA−road−d.NW.gr ../inputs/USA−road−d/USA−road−d.NW.ss
g 1207945 2840208 1 128569
t 318.589500
v 1207945.000000
i 1179459.250000
f ../inputs/USA−road−d/USA−road−d.NE.gr ../inputs/USA−road−d/USA−road−d.NE.ss
g 1524453 3897636 1 63247
t 421.468533
v 1524453.000000
i 1584641.375000
f ../inputs/USA−road−d/USA−road−d.CAL.gr ../inputs/USA−road−d/USA−road−d.CAL.ss
g 1890815 4657742 1 215354
t 515.651338
v 1890814.875000
i 1901551.625000
f ../inputs/USA−road−d/USA−road−d.LKS.gr ../inputs/USA−road−d/USA−road−d.LKS.ss
g 2758119 6885658 1 138911
t 794.800745
v 2758119.000000
i 2784531.500000
f ../inputs/USA−road−d/USA−road−d.E.gr ../inputs/USA−road−d/USA−road−d.E.ss
g 3598623 8778114 1 200760
t 1133.417436
v 3598623.250000
i 3670446.750000
f ../inputs/USA−road−d/USA−road−d.W.gr ../inputs/USA−road−d/USA−road−d.W.ss
g 6262104 15248146 1 368855
t 2185.667727
v 6262104.000000
i 6187750.000000
f ../inputs/USA−road−d/USA−road−d.CTR.gr ../inputs/USA−road−d/USA−road−d.CTR.ss
g 14081816 34292496 1 214013
t 8338.832300
v 14081816.000000
i 14542507.000000
```