Chapter 1

Parallel Boost Graph Library

Contributed by Trenton W. Ford

1.1 Introduction

The Parallel Boost Graph Library**(PBGL)** is a part of the Boost Graph Library**(BGL)**. The parallel libraries offers packages that focus on the distribution of storage and computation of graphs and graph algorithms. The PBGL uses the MPI scheme Since 1999 the BGL has been in the Boost C++ Libraries commonly referred to as Boost[1, 2]. Since 2008 the PBGL has been part of BGL. Boost is a set of more than 80 separate libraries designed to be fast, relatively lightweight, and as generic as possible. Similarly, these same library implementation paradigms and techniques define the PBGL.

For instructions on how to install and link the Boost and the Parallel Boost Graph Libraries, please visit the Boost Library Documentation for the current release version[1].

1.2 Background

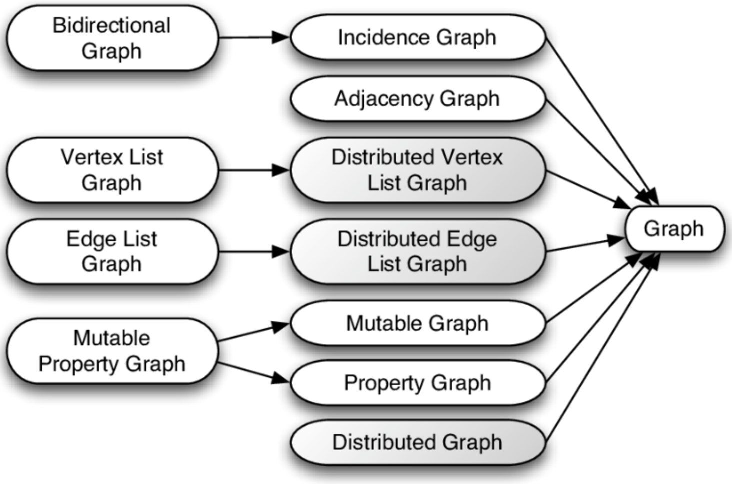
Discuss what was the reasons for the paradigm, what was it derived from, what kind of tool chain it is, what in general is its objective, where can you get code.

Within the PBGL, the central theme of design was genericism. The decision to make a maximally generic graph library was driven the developers' realization that there are many graphs with different properties on top of which many algorithms can operate. These algorithms have associated properties and underlying structures as well. With this understanding it became clear that for any C++ based graph library to have broad applicability, it would require a well-defined graph structure segmentation (underlying data-structure, functional interface, graph object properties, etc.). This structure will be described in detail in Section 1.4.

Prior to BGL beginning development, there was already strong movement towards the creation of maximally generalizable code *(generic programming)* libraries in C++ due to the addition of templating and the Standard Template Library**(STL)** in 1994 and the need for better code reuse within the language. The BGL was developed in response to the growing applicability of graphs as convenient computational representations. The need for a subset of the BGL to be dedicated to the parallelization of the ideas and concepts distilled during the BGL's development became more evident as larger and more complex graphs were beginning to gain popularity and utility.

[1] https://www.boost.org/doc/

Figure 1.1: PBGL specific concepts are shown in gray, while BGL concepts otherwise.



1.3 Expressing Graphs

Within the BGL, graphs can be stored in two main structural types: adjacency lists and adjacency matrices. The PBGL extends these storage methods to handle distributed storage of the data-structures while maintaining the same access and manipulation methods as the BGL exposed. This makes porting code written to work with the BGL straightforward to parallelize using the PBGL. The BGL also includes methods to use other storage schema while still exposing the same standardized and generic BGL interface for developers. This makes the BGL, and by extension, the PBGL quite adept at working in congruence with relational database platforms among other software. To fully understand how graphs are defined within the library, it is first worth understanding how a graph is deconstructed as it relates to the design of the graph generics. These ideas are encapsulated in Section 1.3.1.

1.3.1 Graph Concepts

A graph concept, as it relates to the BGL, is the set of syntactic requirements, function prototypes, as well as semantic requirements that are necessary to make the BGL as generic as possible, while still maintaining the ability to be flexible upon full instantiation of the template. The idea of concepts is a crossover from general generic programming paradigms within C++. For a more through introduction to these concepts a great resource is the book *Generic Programming and the STL*.

A large component of the PBGL design process depends on the BGL upon which it was built. From the BGL, the PBGL adopts it usage of the graph Concepts Taxonomies *(fig 1.1)*, but extends it to capture distributed graph concepts such as graphs built on distributed adjacency lists or distributed adjacency matrices.

The concepts themselves could be more easily considered as taxonomy of graph types where the separations of graph types is driven by the barriers imposed by generic programming and the strongly typed nature of C++. Said another way, the taxonomies served as natural delineations between the requirements and constraints of each graph type.

For a more in-depth look at the rationale behind the graph concepts, please see Douglas and Lumsdaine's original PBGL paper[2].

1.4 Syntax

The Boost libraries have a reputation for being sytactically difficult. This perception is driven mainly by the nature of templating and generic programming in C++. For a developer interested in using Boost and the PBGL it is often useful to clearly define the graph types that the application calls for, and to either find or write a wrapper class that implements the appropriate graph type and exposes an interface that is more convienient. For simple enough graphs using the libraries unwrapped is relatively simple. Sections 1.4.1-1.4.3 contain simple use cases and some example code.

1.4.1 Simple Graph Creation

The most useful introduction to the PBGL is through a simple example or creating a graph. When deciding on the set of graph concepts to utilize, it's usually prudent to first consider the use-case for the graph. For instance, the following graph could pattern a family tree structure where we imagine that the edges are directed and the sparcity of the graph structure makes an adjacency list the optimal underlying data-structure.

Listing 1.1: Create a Simple Family Tree Graph

```
#include <boost/graph/adjacency_list.hpp> // for customizable graphs
// Construct a graph with the vertices container as a vector
enum family {Bob, Tod, Rob, Jeb, Sue, Ann, Mae, Rea, N}
void main()
{
  typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
  // graph class is adjacency_list, vertices and edges will be stored in vectors
  // The graph will be undirected
  Graph g(8); //Create a graph sized for 8 vertices

  add_edge(Bob, Sue, g);
  add_edge(Bob, Mae, g);
  add_edge(Bob, Rob, g);
  add_edge(Sue, Ann, g);
  add_edge(Mae, Tod, g);
  add_edge(Rob, Jeb, g);
  add_edge(Rob, Rae, g);
}
```

The above family tree example contained no parallel components, and relied fully on the serial BGL, but this example serves the purpose of introducing the syntax of the generic library.

1.4.2 Property Maps

The PBGL uses property maps in multiple incarnations to serve as convenient associative methods for graph attributes. Most prominently property maps can be used to relate properties or attributes to *vertex descriptors* and *edge descriptors*. Here is an example of a vertex property and an edge property:

```
struct VertexProperties
{
  std::string name;
}

struct EdgeProperties
{
  double weight;
}
```

If we wanted to map these properties to graph objects, the BGL offers three functions that facilitate the property map associations: ***get(p_map, key), put(p_map, key, value),*** and ***p_map[key]***. Here's an example of using the get, set, and key function:

```
\\ Create a vector map using the provided boost type
boost::vector_property_map<VertexProperties , indexMapType>
  dataMap(num_vertices(graph), indexMap);

\\ Select a random vertex from the graph using an RNG
Vertex v = random_vertex(graph, rng);

\\ Get the vertex properties from vertex v
auto vp = get(dataMap, v);

\\ Change the values in the property
vp.set_current_values(1,1,0);

\\ Associate the newly changed vertex properties with vertex v
put(dataMap, init_infected, vp);
```

1.4.3 Graph Traversal

Moving from one vertex in a graph to another across an edge is called graph traversal. For example, if in the family tree example you wanted to find one of Bob's relatives, you could start from the Bob vertex and traverse one move/step/hop to one of Bob's relatives. Vertices that are connected by an edge are called adjacent vertices. This idea of graph traversal is a foundational activity in graphs and especially graph algorithms. This activity exists in both directed and undirected graphs.

Traversing a graph in the BGL is relatively simple as the activity is facilitated through the use of generic iterator classes. There are five types of iterator generics used for traversal: *vertex iterator, edge iterator, out-edge iterator, in-edge iterator, and adjacency iterators.*

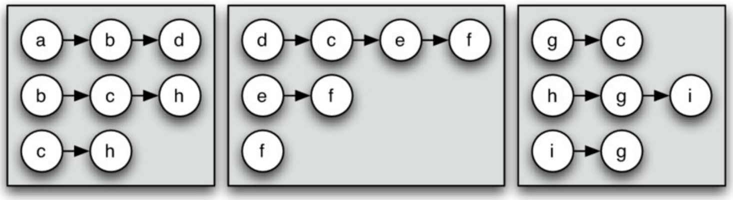
```
Vertex Step(Vertex current_node, Graph G) {

    double random_number = ud(gen);

    // Create two out_edge_iterators of the same type "Graph" as graph G
  graph_traits<Graph::GraphContainer>::out_edge_iterator eit, eend;

  // Select all outgoing edges from current_node
    std::tie(eit, eend) = out_edges( current_node, G);

    for( auto it = eit; it != eend; ++it) {
      //loop through all the edges and store the last seen target vertex
      Vertex target = boost::target(*it, G);
    }
    // return the last seen target vertex
    return target;

}
```

1.5 Key Graph Primitives

Given the generic nature of PBGL, it doesn't contain "primatives" of the sort that one would readily recognize. The lowest level of object that it supports are object types defined within C++ and the STL, and the lowest level objects that can be created within PBGL must have some definitions given in the form of the same datatypes from C++ and STL that it recognizes.

1.6 Execution Model

Figure 1.2: Distributed adjacency list representation



Within PBGL, the primary storage data-structure for graphs is the distributed adjacency list representation. These adjacency lists are partitioned and distributed across multiple computational nodes and the edges and vertices within those partitions are said to "belong to" the computational node upon which its partition of the adjacency list was placed. The same computation node has ownership of the outgoing edges from all of the vertices that it owns as well. Figure 1.2 illustrates the distribution of vertex and edge ownership over three computation nodes.

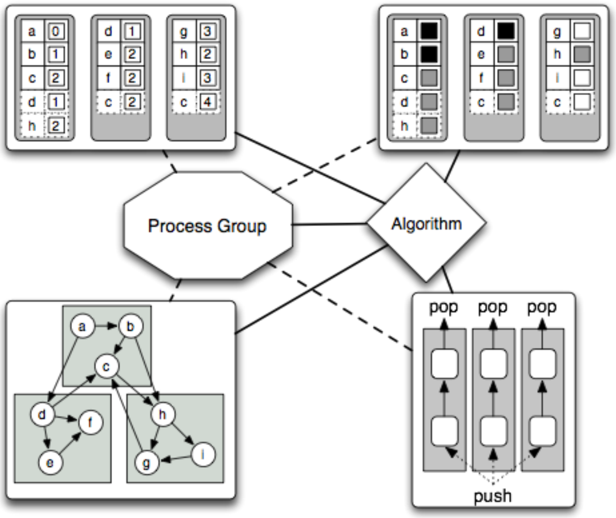
Graph object properties and property maps are also stored on the computational node local to computational node that owns the graph object. From these data-structures, the executional model is derived from the interplay between the Process Group and the parallel algorithm. Figure 1.3 illustrates the relationship between all of the components and how they interconnect.

Within the process group, PBGL masks MPI message passing with standard BGL flavor methods that maintain data consistency on the backend. Direct commands can be passed to the underlying MPI system for added flexibility, but most graph algorithms will find the process group interface more than adequate.

1.7 Examples

A great deal of a developer's time using the PBGL will be spent developing wrappers to reduce the code complexity of using the libraries directly in all of one's code. For basic examples of usage please see the Section 1.4 on Syntax. This section will illustrate an example of PBGL wrapper class that could be used to lower the cognative load of using the PBGL.

Figure 1.3: Full PBGL Architecture



Listing 1.2: Simple PBGL Wrapper

```
#include <boost/config.hpp>
#include <boost/version.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/property_map/property_map.hpp>
#include <boost/static_assert.hpp>

// Enable PBGL interfaces to BGL algorithms
#include <boost/graph/use_mpi.hpp>

// Communication via MPI
#include <boost/graph/distributed/mpi_process_group.hpp>

// Distributed adjacency list
#include <boost/graph/distributed/adjacency_list.hpp>


using namespace boost;
using boost::graph::distributed::mpi_process_group;
using namespace std;

// VertexProperties is the property container that will be passed into the
// templated graph object - should work for serial and parallel versions
struct VertexProperties
{
    std::string name;
};

// EdgeProperties is the property container that will be passed into the
// templated graph object - should work for serial and parallel versions
struct EdgeProperties
{
    int distance = 0;
};

/* definition of basic boost::graph properties */
enum vertex_properties_t { vertex_properties };
enum edge_properties_t { edge_properties };

namespace boost {
    BOOST_INSTALL_PROPERTY(vertex, properties);
    BOOST_INSTALL_PROPERTY(edge, properties);
}

// Graph Class
template < typename VERTEXPROPERTIES, typename EDGEPROPERTIES >
class Graph
{
public:
    /* a distributed adjacency list */
    /* An undirected, weighted graph with distance values stored on the vertices. */
    typedef adjacency_list<
        vecS,
        distributedS<mpi_process_group, vecS>,
        bidirectionalS, // directed graph
        property<vertex_properties_t, VERTEXPROPERTIES>,
        property<edge_properties_t, EDGEPROPERTIES> >
    GraphContainer;

    /* typedefs to begin normalization of Boost syntax */
    typedef typename graph_traits<GraphContainer>::vertex_descriptor Vertex;
    typedef typename graph_traits<GraphContainer>::edge_descriptor Edge;
    typedef std::pair<Edge, Edge> EdgePair;

    typedef typename graph_traits<GraphContainer>::vertex_iterator vertex_iter;
    typedef typename graph_traits<GraphContainer>::edge_iterator edge_iter;
    typedef typename graph_traits<GraphContainer>::adjacency_iterator adjacency_iter;
    typedef typename graph_traits<GraphContainer>::out_edge_iterator out_edge_iter;
    typedef typename graph_traits<GraphContainer>::degree_size_type degree_t;

    typedef std::pair<adjacency_iter, adjacency_iter> adjacency_vertex_range_t;
    typedef std::pair<out_edge_iter, out_edge_iter> out_edge_range_t;
    typedef std::pair<vertex_iter, vertex_iter> vertex_range_t;
    typedef std::pair<edge_iter, edge_iter> edge_range_t;


    /* constructors etc. */
    Graph()
    {}

    Graph(const Graph& g) :
        graph(g.graph)
    {}

    virtual ~Graph()
    {}


    /* structure modification methods */
    void Clear()
    {
       graph.clear();
    }

    Vertex AddVertex(const VERTEXPROPERTIES& prop)
    {
       Vertex v = add_vertex(graph);
       properties(v) = prop;
       return v;
    }

    void RemoveVertex(const Vertex& v)
    {
       clear_vertex(v, graph);
       remove_vertex(v, graph);
    }

    EdgePair AddEdge(const Vertex& v1, const Vertex& v2,
      const EDGEPROPERTIES& prop_12, const EDGEPROPERTIES& prop_21)
    {
       /* TODO: maybe one wants to check if this edge could be inserted */
       Edge addedEdge1 = add_edge(v1, v2, graph).first;
       Edge addedEdge2 = add_edge(v2, v1, graph).first;

       properties(addedEdge1) = prop_12;
       properties(addedEdge2) = prop_21;

       return EdgePair(addedEdge1, addedEdge2);
    }

    Edge AddDirectedEdge(const Vertex& v1,
      const Vertex& v2, const EDGEPROPERTIES& prop)
    {
       return add_edge(v1, v2, graph).first;
    }

    /* selectors and properties */

    adjacency_vertex_range_t getAdjacentVertices(const Vertex& v) const
    {
       return adjacent_vertices(v, graph);
    }

    int getVertexCount() const
    {
       return num_vertices(graph);
    }

    int getVertexDegree(const Vertex& v) const
    {
        return degree(v, graph);
    }

protected:
    GraphContainer graph;
};
```

1.8 Conclusion

Forthcoming.

Bibliography

[1] C++ Boost. Libraries, 2012.

[2] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.