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. 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.

The paradigm is one of maximum flexibility but does so at the cost of code interpretability and with a lack of syntactic sugar that often deters would be users.

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 many graphs with different properties on top of which, many algorithms would 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.).

1.2.1 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.
Figure 1.1: PBGL specific concepts are shown in gray, while BGL concepts otherwise.

A large component of the PBGL design process depends on the BGL upon which it was built. From the BGL, the PBGL adopts its 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.

1.2.2 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 sparsity of the graph structure makes an adjacency list the optimal underlying data-structure.

Listing 1.1: Create a Simple Family Tree Graph

```cpp
#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;
    // 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);
    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.

1.2.3 Property Maps

The PBGL uses property maps in multiple incarnations to serve as convenient containers for graph attributes. Most prominently property maps can be used to define edge descriptors and vertex descriptors. Simple examples of these maps are below.

```cpp
#include <boost/graph/adjacency_list.hpp> // for customizable graphs

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);
}
```

1.2.4 Graph Traversal

Iteration through a graph

Prior to PBGL 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.

1.3 Expressing Graphs

Discuss here how graphs are defined in the paradigm.

1.4 Syntax

Discuss here what an approximate syntax for program statements in the paradigm look like (if a "language"), or if a library what are common calling conventions/arguments.

1.5 Key Graph Primitives

Discuss here what are the key graph primitives supported by the paradigm.
1.6 Execution Model
Describe the execution model of how programs execute, especially parallel.

1.7 Examples
Discuss here any code examples and/or performance reports

1.8 Conclusion
Discuss other topics that may make sense.
Bibliography
