Graph Library: Comparison

Document #:	**D3337r0**
Date:	2024-09-12
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency SG6 Numerics
Revises:	(none)

Reply-to:	Phil Ratzloff (SAS Institute) phil.ratzloff@sas.com Andrew Lumsdaine lumsdaine@gmail.com
Contributors:	Kevin Deweese Muhammad Osama (AMD, Inc) Jesun Firoz Michael Wong (Intel) Jens Maurer Richard Dosselmann (University of Regina) Matthew Galati (Amazon) Guy Davidson (Creative Assembly) Oliver Rosten

1 Getting Started

This paper is one of several interrelated papers for a proposed Graph Library for the Standard C++ Library. The Table 1 describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now separated into the following papers.
P3126	Active	**Overview**, describes the big picture of what we are proposing.
P3127	Active	**Background and Terminology** provides the motivation, theoretical background, and terminology used across the other documents.
P3128	Active	**Algorithms** covers the initial algorithms as well as the ones we'd like to see in the future.
P3129	Active	**Views** has helpful views for traversing a graph.
P3130	Active	**Graph Container Interface** is the core interface used for uniformly accessing graph data structures by views and algorithms. It is also designed to easily adapt to existing graph data structures.
P3131	Active	**Graph Containers** describes a proposed high-performance `compressed_graph` container. It also discusses how to use containers in the standard library to define a graph, and how to adapt existing graph data structures.
P3337	Active	**Comparison to other graph libraries** on performance and usage syntax.

Table 1: Graph Library Papers

Reading them in order will give the best overall picture. If you're limited on time, you can use the following guide to focus on the papers that are most relevant to your needs.

Reading Guide

— If you're **new to the Graph Library**, we recommend starting with the *Overview* (P3126) paper to understand the focus and scope of our proposals. You'll also want to check out it stacks up against other graph libraries in performance and usage syntax in the *Comparison* (P3337) paper.

— If you want to **understand the terminology and theoretical background** that underpins what we're doing, you should read the *Background and Terminology* (P3127) paper.

— If you want to **use the algorithms**, you should read the *Algorithms* (P3128) and *Graph Containers* (P3131) papers. You may also find the *Views* (P3129) and *Graph Container Interface* (P3130) papers helpful.

— If you want to **write new algorithms**, you should read the *Views* (P3129), *Graph Container Interface* (P3130), and *Graph Containers* (P3131) papers. You'll also want to review existing implementations in the reference library for examples of how to write the algorithms.

— If you want to **use your own graph data structures**, you should read the *Graph Container Interface* (P3130) and *Graph Containers* (P3131) papers.

2 Revision History

D3337r0

— New paper comparing the Graph Library to the NWGraph and Boost Graph Libraries on performance and usage syntax.

3 Naming Conventions

Table 2 shows the naming conventions used throughout the Graph Library documents.

Template Parameter	Type Alias	Variable Names	Description
G			Graph
	graph_reference_t<G>	g	Graph reference
GV		val	Graph Value, value or reference
EL		el	Edge list
V	vertex_t<G>		Vertex
	vertex_reference_t<G>	u,v,x,y	Vertex reference. u is the source (or only) vertex. v is the target vertex.
VId	vertex_id_t<G>	uid,vid,seed	Vertex id. uid is the source (or only) vertex id. vid is the target vertex id.
VV	vertex_value_t<G>	val	Vertex Value, value or reference. This can be either the user-defined value on a vertex, or a value returned by a function object (e.g. VVF) that is related to the vertex.
VR	vertex_range_t<G>	ur,vr	Vertex Range
VI	vertex_iterator_t<G>	ui,vi	Vertex Iterator. ui is the source (or only) vertex.
		first,last	vi is the target vertex.
VVF		vvf	Vertex Value Function: vvf(u) → vertex value, or vvf(uid) → vertex value, depending on requirements of the consume algorithm or view.
VProj		vproj	Vertex info projection function: vproj(x) → vertex_info<VId,VV> .
	partition_id_t<G>	pid	Partition id.
		P	Number of partitions.
PVR	partition_vertex_range_t<G>	pur,pvr	Partition vertex range.
E	edge_t<G>		Edge
	edge_reference_t<G>	uv,vw	Edge reference. uv is an edge from vertices u to v . vw is an edge from vertices v to w .
EV	edge_value_t<G>	val	Edge Value, value or reference. This can be either the user-defined value on an edge, or a value returned by a function object (e.g. EVF) that is related to the edge.
ER	vertex_edge_range_t<G>		Edge Range for edges of a vertex
EI	vertex_edge_iterator_t<G>	uvi,vwi	Edge Iterator for an edge of a vertex. uvi is an iterator for an edge from vertices u to v . vwi is an iterator for an edge from vertices v to w .
EVF		evf	Edge Value Function: evf(uv) → edge value, or evf(eid) → edge value, depending on the requirements of the consuming algorithm or view.
EProj		eproj	Edge info projection function: eproj(x) → edge_info<VId,Sourced,EV> .

Table 2: Naming Conventions for Types and Variables

4 Syntax Comparison

We provide a usage syntax comparison of several graph algorithms in Tier 1 of P3128 against the **boost::graph** equivalent. We refer to the reference implementation associated with this proposal as **std::graph**. These algorithms are breadth-first search (BFS, Figure 1), connected components (CC, Figure 2), single sourced shortest paths (SSSP, Figure 3), and triangle counting (TC)(4). We take these algorithms from the GAP Benchmark Suite [1] which we discuss more in Section 5. We also defer to Section 5 any discussion of underlying implementation details.

Unlike **boost::graph**, **std::graph** does not specify edge direction as a graph property. If a graph in **std::graph** implemented by **container::compressed_graph** is undirected, then it will contain edges in both directions. **boost::graph** has a **boost::graph::undirectedS** property which can be used in the **boost::graph::adjacency_matrix** class to specify an undirected graph, but not in the **boost::graph::compressed_sparse_row_graph** class. Thus in Figures 1-4, the graph type always includes **boost::graph::directedS**. Similarly to **std::graph**, undirected graphs must contain the edges in both directions.

Intermediate data structures (i.e. edgelists) will be needed to construct the compressed graph structures. In order to focus on the differences in algorithm syntax, we omit code which populates the graph data structures. In the following sections we address the syntax changes for each of these algorithms.

```cpp
using namespace boost;

using G =
  compressed_sparse_row_graph<directedS,
      no_property, no_property>;
using VId = graph_traits<G>::vertex_descriptor;

G g;
//populate g

vector<VId> parents(num_vertices(g));

auto vis = make_bfs_visitor(
  make_pair(record_predecessors(parents.begin(),
      on_tree_edge())));


breadth_first_search(g, vertex(0, g), visitor(vis
    ));
```

```cpp
using namespace std;
using namespace graph;

using G = container::compressed_graph<void, void,
     void, uint32_t, uint32_t>;
using VId = vertex_id_t<G>;

G g;
// populate g

vector<VId> parents(size(vertices(g));

auto bfs = edges_breadth_first_search_view<G,void
    ,true>(g, 0);


for (auto&& [uid, vid, uv] : bfs) {
  parents[vid] = uid;
}
```

Figure 1: Breadth-First Search Syntax Comparison

4.1 Breadth-First Search

BFS is often described as a graph algorithm, though a BFS traversal by itself does not actually perform any task. In reality, it is a data access pattern which specifies an order vertices and edges should be processed by some higher level algorithm. **boost::graph** provided a very customizable interface to this data access pattern through the use of visitors which allows users to customize function calls during BFS events. For example discover_vertex is called when a vertex is encountered for the first time; examine_vertex is called when a vertex is popped from the queue; examine_edge is called on each edge of a vertex when it is discovered, etc.

This capability is very powerful but often cumbersome if the BFS traversal simply requires vertex and edge access upon visiting. For this reason **std::graph** provides a simple, range-based-for loop BFS traversal called a view. Figure 1 compares the most simple **boost::graph** BFS visitor against the range-based-for loop implementation.

```cpp
using namespace std;
using namespace boost;

using G =
  compressed_sparse_row_graph<directedS,
      no_property, no_property>;

G g;
//populate g

vector<size_t> c(N); //components
size_t num = connected_components(g, &c[0]);
```

```cpp
using namespace std;
using namespace graph;

using G = container::compressed_graph<void, void,
     void, uint32_t, uint32_t>;


G g;
//populate g

vector<size_t> c(size(vertices(g))); //components
size_t num = connected_components(g, c);
```

Figure 2: Connected Components Syntax Comparison

The authors of this proposal acknowledge that some power users still want the full customization provided by visitors, and we plan to add them to this proposal.

4.2 Connected Components

There is very little difference in the connected component interfaces.

4.3 Single Source Shortest Paths

Of the four algorithms discussed here, only SSSP makes use of some edge property, in this case distance. Along with the input edge property, the algorithm also associates with every vertex a distance from the start vertex, and a predecessor vertex to store the shortest path. In Figure 3 we see that **boost::graph** requires property maps to lookup edge and vertex properties. These property maps are tightly coupled with the graph data structures. We propose properties be stored external to the graph. For edge properties we provide a weight lambda function to the algorithm to lookup distance from the **edge_reference_t**.

4.4 Triangle Counting

boost::graph does not contain a global triangle counting similar to the one proposed by **std::graph**. Instead we must iterate through the vertices counting the number of triangles on every vertex, and adjust for overcounting at the end.

```cpp
using namespace std;
using namespace boost;

using G =
  compressed_sparse_row_graph<directedS,
      no_property, property<edge_weight_t, int>>;
using VId = graph_traits<G>::vertex_descriptor;

G g;
//populate g

vector<VId> p(num_vertices(g)); //predecessors
vector<int> d(num_vertices(g)); //distances

property_map< graph_t, edge_weight_t >::type
    weightmap = get(edge_weight, g);


dijkstra_shortest_paths(g, vertex(0, g),
    predecessor_map(make_iterator_property_map( p
    .begin(), get(vertex_index, g))).distance_map
    (make_iterator_property_map(d.begin(), get(
    vertex_index, g))));
```

```cpp
using namespace std;
using namespace graph;

using G = container::compressed_graph<int, void,
    void, uint32_t, uint32_t>;

using VId = vertex_id_t<G>;

G g;
//populate g

vector<VId> p(size(vertices(g))); //predecessors
vector<int> d(size(vertices(g))); //distances
init_shortest_paths(distance, predecessors);
auto weight_fn = [&g](graph::edge_reference_t<
    graph_type> uv) -> int {
  return edge_value(g, uv);
};




dijkstra_shortest_paths(g, 0, d, p, weight_fn);
```

Figure 3: Single Source Shortest Paths Syntax Comparison

```cpp
using namespace boost;

using G =
  compressed_sparse_row_graph<directedS,
      no_property, no_property>;
using VId = graph_traits<G>::vertex_descriptor;

G g;
//populate g

size_t count = 0;
for(size_t i = 0; i < N; i++) {
  VId cur = vertex(i, g);
  count += num_triangles_on_vertex(g, cur);
}
count /= 6;
```

```cpp
using namespace graph;

using G = container::compressed_graph<void, void,
     void, uint32_t, uint32_t>;



G g;
//populate g

size_t count;




count = triangle_count(g);
```

Figure 4: Triangle Counting Syntax Comparison

5 Performance Comparison

5.1 Experimental Setup

To evaluate the performance of this proposed library, we compare its reference implementation (**std::graph**) against **boost::graph** and NWGraph on a subset of the GAP Benchmark Suite[1]. This comparison includes four of the five GAP algorithms that are in the tier 1 algorithm list of this proposal: triangle counting (TC), weak connected components (CC), breadth-first search (BFS), and single-source shortest paths (SSSP). Table 3 summarizes the graphs specified by the GAP benchmark. These graphs were chosen to be large but still fit on shared memory machines and have edge counts in the billions. We compare to BGL because it the commonly used sequential C++ graph library as described above. NWGraph was implemented with many of the ideas of this proposal in mind, and we expect very similar performance between NWGraph and this reference implementation.

Name	Description	#Vertices (M)	#Edges (M)	Degree Distribution	(Un)directed	References
road	USA road network	23.9	57.7	bounded	undirected	[2]
Twitter	Twitter follower links	61.6	1,468.4	power	directed	[3]
web	Web crawl of .sk domain	50.6	1,930.3	power	directed	[4]
kron	Synthetic graph	134.2	2,111.6	power	undirected	[5]
urand	Uniform random graph	134.2	2,147.5	normal	undirected	[6]

Table 3: Summary of GAP Benchmark Graphs

The NWGraph authors published a similar comparison to BGL[7] in which they demonstrated performance improvement of NWGraph over BGL. To simplify experimental setup, we rerun these new experiments using the same machine used in[7], (compute nodes consisting of two Intel® Xeon® Gold 6230 processors, each with 20 physical cores running at 2.1 GHz, and 188GB of memory per processor). NWGraph and **std::graph** were compiled with gcc 13.2 using -Ofast -march=native compilation flags.

Even though NWGraph contains an implementation of Dijkstra, the SSSP results in [7] were based on delta-stepping. For this comparison, **std::graph** and NWgraph both use Dijkstra. The NWGraph implementations also used a version of SSSP which did not compute a predecessor map, only providing the final distances. **std::graph** provides SSSP without predecessors called *dijkstra_shortest_distances* which is similar to the Dijkstra in Figure 3 with the predecessor argument omitted. **boost::graph** can also compute shortest distances only by omitting the predecessor map. We use the shortest distance version for these experiments.

The NWGraph and **std::graph** implementation of CC is based on the Afforest [8] algorithm. While BFS and SSSP implementations are very similar for NWGraph and **std::graph**, the latter contains support for event-based visitors. If this functionality is not required it should be optimized out and not incur a performance penalty, but we seek to verify this experimentally. NWGraph and **std::graph** contain similar implementations of triangle counting which perform a set intersection of the neighbor list of vertices u and v, only if v is a neighbor of u. By first performing a lexicographic sort of the vertex ids of the adjacency structure, the set intersection is limited to neighbors with vertex ids greater than u and v, or equivalently the upper triangular portion of the adjacency matrix. Table 4 summarizes our GAP benchmark results for **std::graph** compared to **boost::graph** and NWGraph.

5.2 Experimental Analysis

BFS results are consistent between the three implementations, except for the kron graph where **std::graph** is 2.4x faster than NWGraph and 3.4x faster than **boost::graph**.

CC results are consistent between NWGraph and **std::graph**, which are both much faster than **boost::graph** on twitter, kron, and urand. This is reasonable as **boost::graph** is using a simple breadth-first search based CC

Algorithm	Library	road	twitter	kron	web	urand
BFS	**boost::graph**	1.09s	12.11s	54.80s	5.52s	73.26s
	NWGraph	0.91s	11.25s	38.86s	2.37s	64.63s
	std::graph	1.39s	8.54s	16.34s	3.52s	62.75s
CC	BGL	1.36s	21.96s	81.18s	6.64s	134.23s
	NWGraph	1.05s	3.77s	10.16s	3.04s	36.59s
	std::graph	0.78s	2.81s	8.37s	2.23s	33.75s
SSSP	BGL	4.03s	47.89s	167.20s	28.29s	OOM
	NWGraph	3.63s	109.37s	344.12s	35.58s	400.23s
	std::graph	4.22s	79.75s	211.37s	33.87s	493.15s
TC	BGL	1.34s	>24H	>24H	>24H	4425.54s
	NWGraph	0.41s	1327.63s	6840.38s	131.47s	387.53s
	std::graph	0.17s	459.08s	2357.95s	50.04s	191.36s

Table 4: GAP Benchmark Performance: Time for GAP benchmark algorithms is shown for **boost::graph**, NWGraph, **std::graph**

algorithm while the other two implementations use the Afforest algorithm. Of the four algorithms, CC shows the closest agreement between NWGraph and **std::graph**.

SSSP results are more mixed, with differing performance on twitter and kron. Interestingly of the algorithms we profile, this is the only one where **boost::graph** is often faster than the other implementations, faster than **std::graph** by 1.7x on twitter and 1.3x on kron, though failing by running out of memory on urand.

TC performance from the naïve **boost::graph** implementation is far slower than the adjacency matrix set intersection used by NWGraph and **std::graph**. Since the same triangle is counted 6 times in **boost::graph**, we expect at least that much of a slowdown, but in fact the slowdown is often much worse. However the TC results are concerning because the **std::graph** performance is around 2x that of NWGraph. We plan to review the implementation details to discover the cause of this discrepancy.

6 Memory Allocation

Unlike existing STL algorithms, the graph algorithms we propose here will often require their own memory allocations. Table 5 records the internal memory allocations required for our implementations of the GAP Benchmark algorithms where relevant. It is important to note that the memory usage is not prescribed by the algorithm interface in P3128, and is ultimately up to the library implementer. Some memory use, such as the queues in BFS and SSSP, will probably be common to most implementations. However, the color map in BFS and the reindex map in CC (used to ensure the resulting component indices are contiguous) could potentially be avoided.

Algorithm	Required Member Data	Max Size
BFS	queue	$O(\lvert V\rvert)$
	color map	V
CC	reindex map	$O(\lvert components\rvert)$
SSSP	priority queue	$O(\lvert E\rvert)$
TC	None	NA

Table 5: Memory Allocations of GAP Benchmark Algorithm Implementations

Acknowledgements

Phil Ratzloff's time was made possible by SAS Institute.

Portions of *Andrew Lumsdaine's* time was supported by NSF Award OAC-1716828 and by the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada.

Muhammad Osama's time was made possible by Advanced Micro Devices, Inc.

The authors thank the members of SG19 and SG14 study groups for their invaluable input.

References

[1] A. Azad, M. M. Aznaveh, S. Beamer, M. P. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz, H. A. Gabb, G. Gill, B. Hegyi, S. Kolodziej, T. M. Low, A. Lumsdaine, T. Manlaibaatar, T. G. Mattson, S. McMillan, R. Peri, K. Pingali, U. Sridhar, G. Szarnyas, Y. Zhang, and Y. Zhang, "Evaluation of graph analytics frameworks using the gap benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 216–227, 2020.

[2] "9th DIMACS implementation challenge - Shortest paths.." `http://www.dis.uniroma1.it/challenge9/`, 2006.

[3] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," *WWW*, 2010.

[4] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," *WWW*, pp. 595–601, 2004.

[5] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," in *Cray User's Group*, CUG, 2010.

[6] P. Erdős and A. Rényi, "On random graphs. I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[7] A. Lumsdaine, L. D'Alessandro, K. Deweese, J. Firoz, T. Liu, S. McMillan, P. Ratzloff, and M. Zalewski, "Nwgraph library code." "https://github.com/pnnl/NWGraph".

[8] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *IPDPS*, pp. 12–21, IEEE, 2018.

[9] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, Dec. 2001.