Graph Library: Graph Container Interface

Document #:	**P3130r0**
Date:	2024-02-05
Project:	Programming Language C++
Audience:	Library Evolution SG19 Machine Learning SG14 Game, Embedded, Low Latency SG6 Numerics
Revises:	P1709r5

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 (Codeplay) Jens Maurer Richard Dosselmann (University of Regina) Matthew Galati (Amazon)

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**, describing the big picture of what we are proposing.
P3127	Active	**Background and Terminology** providing the motivation, theoretical background and terminology used across the other documents.
P3128	Active	**Algorithms** covering 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** describing 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.

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* paper (P3126) to understand focus and scope of our proposals.

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

— If you want to **use the algorithms**, you should read the *Algorithms* paper (P3128) and *Graph Containers* paper (P3131).

— If you want to **write new algorithms**, you should read the *Views* paper (P3129), *Graph Container Interface* paper (P3130) and *Graph Containers* paper (P3131). 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 container**, you should read the *Graph Container Interface* paper (P3130) and *Graph Containers* paper (P3131).

2 Revision History

P3130r0

— Split from P1709r5. Added *Getting Started* section.

— Add default implementation for `target_id(g,uv)` when the graph type matches the pattern `random_access_range<forward_range<integral>>` or `random_access_range<forward_range<tuple<integral,...>>>` ; `vertex_id_t<G>` also defaults to the `integral` type given.

— Revised concept definitions, adding `sourced_targeted_edge` and `target_edge_range` , and replaced summary table with code for clarity. Also assured that all combinations of adjacency list concepts for *basic*, *sourced* and *index* exist.

— Move text for graph data structures created from std containers from Graph Container Interface to Container Implementation paper.

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
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 descriptor projection function: vproj(x) → vertex_descriptor<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 .
EId	edge_id_t<G>	eid,uvid	Edge id, a pair of vertex_ids.
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 descriptor projection function: eproj(x) → edge_descriptor<VId,Sourced,EV> .
PER	partition_edge_range_t<G>		Partition Edge Range for edges of a partition vertex.

Table 2: Naming Conventions for Types and Variables

4 Graph Container Interface

The Graph Container Interface defines the primitive concepts, traits, types and functions used to define and access an adacency graph, no matter its internal design and organization. Thus, it is designed to reflect all forms of adjacency graphs including a vector of lists, CSR-based graph and adjacency matrix, whether they are in the standard or external to the standard.

All algorithms in this proposal require that vertices are stored in random access containers and that `vertex_id_t<G>` is integral, and it is assumed that all future algorithm proposals will also have the same requirements.

The Graph Container Interface is designed to support a wider scope of graph containers than required by the views and algorithms in this proposal. This enables for future growth of the graph data model (e.g. incoming edges on a vertex), or as a framework for graph implementations outside of the standard. For instance, existing implementations may have requirements that cause them to define features with looser constraints, such as sparse vertex_ids, non-integral vertex_ids, or storing vertices in associative bi-directional containers (e.g. std::map or std::unordered_map). Such features require specialized implementations for views and algorithms. The performance for such algorithms will be sub-optimal, but is preferrable to run them on the existing container rather than loading the graph into a high-performance graph container and then running the algorithm on it, where the loading time can far outweigh the time to run the sub-optimal algorithm. To achieve this, care has been taken to make sure that the use of concepts chosen is appropriate for algorithm, view and container.

5 Core Concepts

This section describes the core concepts to describe the adjacency lists used for graphs in the Graph Library. There are a number of qualifiers concept names that are are used.

— **basic** where there is only a vertex id but no vertex reference.

— **index** where the vertex range is random-access and the vertex id is integral.

— **sourced** where an edge has a source id.

5.1 Edge Concepts

The types of edges that can occur in a graph are described with the edges concepts. The `E` edge template parameter allows for different types of edges, such as incoming and outgoing edge types.

```
template <class G, class E>
concept basic_targeted_edge = requires(G&& g, edge_reference_t<G> uv) { target_id(g, uv); };

template <class G, class E>
concept basic_sourced_edge = requires(G&& g, edge_reference_t<G> uv) { source_id(g, uv); };

template <class G, class E>
concept basic_sourced_targeted_edge = basic_targeted_edge<G, E> && //
                                      basic_sourced_edge<G, E> && //
                                      requires(G&& g, edge_reference_t<G> uv) { edge_id(g, uv); };
template <class G, class E>
concept targeted_edge = basic_targeted_edge<G, E> && //
                        requires(G&& g, edge_reference_t<G> uv) { target(g, uv); };

template <class G, class E>
concept sourced_edge = basic_sourced_edge<G, E> && //
                       requires(G&& g, edge_reference_t<G> uv) { source(g, uv); };

template <class G, class E>
concept sourced_targeted_edge = targeted_edge<G, E> && //
                                sourced_edge<G, E> && //
                                requires(G&& g, edge_reference_t<G> uv) { edge_id(g, uv); };
```

There are two edge range concepts.

```
template <class G>
concept basic_targeted_edge_range = requires(G&& g, vertex_id_t<G> uid) {
  { edges(g, uid) } -> ranges::forward_range;
};

template <class G>
concept targeted_edge_range = basic_targeted_edge_range<G> && //
                              requires(G&& g, vertex_reference_t<G> u) {
                                { edges(g, u) } -> ranges::forward_range;
                              };
```

5.2 Vertex Concepts

The `vertex_range` concept is the general definition used for adjacency lists while `index_vertex_range` is used for high performance graphs where vertices typically stored in a `vector` .

```
template <class G> // (exposition only)
concept _common_vertex_range = ranges::sized_range<vertex_range_t<G>> && //
                               requires(G&& g, vertex_iterator_t<G> ui) { vertex_id(g, ui); };

template <class G>
concept vertex_range = _common_vertex_range<vertex_range_t<G>> && //
                       ranges::forward_range<vertex_range_t<G>>;

template <class G>
concept index_vertex_range = _common_vertex_range<vertex_range_t<G>> && //
                             ranges::random_access_range<vertex_range_t<G>> && //
                             integral<vertex_id_t<G>>;
```

5.3 Adjacency List Concepts

The basic adjacency lists add concepts where there is no vertex object, only a vertex id.

```
template <class G>
concept basic_adjacency_list = vertex_range<G> && //
                               basic_targeted_edge_range<G> && //
                               targeted_edge<G, edge_t<G>>;

template <class G>
concept basic_index_adjacency_list = index_vertex_range<G> && //
                                     basic_targeted_edge_range<G> && //
                                     basic_targeted_edge<G, edge_t<G>>;
template <class G>
concept basic_sourced_adjacency_list = vertex_range<G> && //
                                       basic_targeted_edge_range<G> && //
                                       basic_sourced_targeted_edge<G, edge_t<G>>;

template <class G>
concept basic_sourced_index_adjacency_list = index_vertex_range<G> && //
                                             basic_targeted_edge_range<G> && //
                                             basic_sourced_targeted_edge<G, edge_t<G>>;
```

The adjacency list concepts bring together the vertex and edge concepts used for core graph concepts. All algorithms initially proposed for the Graph Library use the `index_adjacency_list` .

```
template <class G>
concept adjacency_list = vertex_range<G> && //
```

```
                        targeted_edge_range<G> && //
                        targeted_edge<G, edge_t<G>>;

template <class G>
concept index_adjacency_list = index_vertex_range<G> && //
                               targeted_edge_range<G> && //
                               targeted_edge<G, edge_t<G>>;
template <class G>
concept sourced_adjacency_list = vertex_range<G> && //
                                 targeted_edge_range<G> && //
                                 sourced_targeted_edge<G, edge_t<G>>;

template <class G>
concept sourced_index_adjacency_list = index_vertex_range<G> && //
                                       targeted_edge_range<G> && //
                                       sourced_targeted_edge<G, edge_t<G>>;
```

6 Traits

Table 3 summarizes the type traits in the Graph Container Interface, allowing views and algorithms to query the graph's characteristics.

Trait	Type	Comment
`has_degree<G>`	concept	Is the `degree(g,u)` function available?
`has_find_vertex<G>`	concept	Are the `find_vertex(g,_)` functions available?
`has_find_vertex_edge<G>`	concept	Are the `find_vertex_edge(g,_)` functions available?
`has_contains_edge<G>`	concept	Is the `contains_edge(g,uid,vid)` function available?
`define_unordered_edge<G,E> : false_type`	struct	Specialize for edge implementation to derive from `true_type` for unordered edges
`is_unordered_edge<G,E>`	struct	`conjunction<define_unordered_edge<E>, is_sourced_edge<G, E>>`
`is_unordered_edge_v<G,E>`	type alias	
`unordered_edge<G,E>`	concept	
`is_ordered_edge<G,E>`	struct	`negation<is_unordered_edge<G,E>>`
`is_ordered_edge_v<G,E>`	type alias	
`ordered_edge<G,E>`	concept	
`define_adjacency_matrix<G> : false_type`	struct	Specialize for graph implementation to derive from `true_type` for edges stored as a square 2-dimensional array
`is_adjacency_matrix<G>`	struct	
`is_adjacency_matrix_v<G>`	type alias	
`adjacency_matrix<G>`	concept	

Table 3: Graph Container Interface Type Traits

7 Types

Table 4 summarizes the type aliases in the Graph Container Interface. These are the types used to define the objects in a graph container, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, compressed_graph and adjacency matrix.

The type aliases are defined by either a function specialization for the underlying graph container, or a refinement of one of those types (e.g. an iterator of a range). Table 5 describes the functions in more detail.

`graph_value(g)` , `vertex_value(g,u)` and `edge_value(g,uv)` can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types.

Type Alias	Definition	Comment
`graph_reference_t<G>`	`add_lvalue_reference<G>`	
`graph_value_t<G>`	`decltype(graph_value(g))`	optional
`vertex_range_t<G>`	`decltype(vertices(g))`	
`vertex_iterator_t<G>`	`iterator_t<vertex_range_t<G>>`	
`vertex_t<G>`	`range_value_t<vertex_range_t<G>>`	
`vertex_reference_t<G>`	`range_reference_t<vertex_range_t<G>>`	
`vertex_id_t<G>`	`decltype(vertex_id(g))`	
`vertex_value_t<G>`	`decltype(vertex_value(g))`	optional
`vertex_edge_range_t<G>`	`decltype(edges(g,u))`	
`vertex_edge_iterator_t<G>`	`iterator_t<vertex_edge_range_t<G>>`	
`edge_t<G>`	`range_value_t<vertex_edge_range_t<G>>`	
`edge_reference_t<G>`	`range_reference_t<vertex_edge_range_t<G>>`	
`edge_value_t<G>`	`decltype(edge_value(g))`	optional
The following is only available when the optional `source_id(g,uv)` is defined for the edge		
`edge_id_t<G>`	`edge_descriptor<vertex_id_t<G>,true,void,void>`	
`partition_id_t<G>`	`decltype(partition_id(g,u))`	optional
`partition_vertex_range_t<G>`	`vertices(g,pid)`	optional
`partition_edge_range_t<G>`	`edges(g,u,pid)`	optional

Table 4: Graph Container Interface Type Aliases

There is no contiguous requirement for `vertex_id` from one partition to the next, though in practice they will often be assigned contiguously. Gaps in `vertex_id` s between partitions should be allowed.

8 Classes and Structs

The `graph_error` exception class is available, inherited from `runtime_error` . While any function may use it, it is only anticipated to be used by the `load` functions at this time. No additional functionality is added beyond that provided by `runtime_error` .

9 Functions

Table 5 summarizes the functions in the Graph Container Interface. These are the primitive functions used to access an adacency graph, no matter its internal design and organization. Thus, it is designed to be able to reflect all forms of adjacency graphs including a vector of lists, CSR-based graph and adjacency matrix.

When the graph matches the pattern `random_access_range<forward_range<integral>>` or `random_access_range<forward_range<tuple<integral,...>>>` , the default implementation for `target_id(g,uv)` will return the `integral` . Additionally, if the caller does not override `vertex_id(g,ui)` , the `integral` value will define the `vertex_id_t<G>` type.

Functions that have n/a for their Default Implementation must be defined by the author of a Graph Container implementation.

Value functions (`graph_value(g)` , `vertex_value(g,u)` and `edge_value(g,uv)`) can be optionally implemented, depending on whether the graph container supports values on the graph, vertex and edge types. They return a

Function	Return Type	Complexity	Default Implementation
`graph_value(g)`	`graph_value_t<G>`	constant	n/a, optional
`partition_count(g)`	`vertex_id_t<G>`	constant	1
`vertices(g)`	`vertex_range_t<G>`	constant	`g` if `random_access_range<G>` , n/a otherwise
`num_vertices(g)`	`integral`	constant	`size(vertices(g))`
`find_vertex(g,uid)`	`vertex_iterator_t<G>`	constant	`begin(vertices(g))+ uid` if `random_access_range<vertex_range_t<G>>`
`vertex_id(g,ui)`	`vetex_id_t<G>`	constant	`(size_t)(ui - begin(vertices(g)))` Override to define a different `vertex_id_t<G>` type (e.g. int32_t).
`vertex_value(g,u)`	`vertex_value_t<G>`	constant	n/a, optional
`vertex_value(g,uid)`	`vertex_value_t<G>`	constant	`vertex_value(g,*find_vertex(g,uid))` , optional
`degree(g,u)`	`integral`	constant	`size(edges(g,u))` if `sized_range<vertex_edge_range_t<G>>`
`degree(g,uid)`	`integral`	constant	`size(edges(g,uid))` if `sized_range<vertex_edge_range_t<G>>`
`partition_id(g,u)`	`partition_id_t<G>`	constant	0
`partition_id(g,uid)`	`partition_id_t<G>`	constant	`partition_id(g,*find_vertex(g,uid))`
`vertices(g,pid)`	`partition_vertex_range_t<G>`	constant	`vertices(g)`
`num_vertices(g,pid)`	`integral`	constant	`size(vertices(g))`
`edges(g,u)`	`vertex_edge_range_t<G>`	constant	`u` if `forward range<vertex_t<G>>` , n/a otherwise
`edges(g,uid)`	`vertex_edge_range_t<G>`	constant	`edges(g,*find_vertex(g,uid))`
`target_id(g,uv)`	`vertex_id_t<G>`	constant	(see below)
`target(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ target_id(g, uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`
`edge_value(g,uv)`	`edge_value_t<G>`	constant	`uv` if `forward_range<vertex_t<G>>` , n/a otherwise, optional
`find_vertex_edge(g,u,vid)`	`vertex_edge_t<G>`	linear	`find(edges(g,u), [](uv)target_id(g,uv)== vid;})`
`find_vertex_edge(g,uid,vid)`	`vertex_edge_t<G>`	linear	`find_vertex_edge(g,*find_vertex(g,uid),vid )`
`contains_edge(g,uid,vid)`	`bool`	constant	`uid < size(vertices(g))&& vid < size( vertices(g))` if `is_adjacency_matrix_v<G>` .
		linear	`find_vertex_edge(g,uid)!= end(edges(g,uid) )` otherwise.
`edges(g,u,pid)`	`partition_edge_range_t<G>`	linear	`edges(g,u)`
`edges(g,uid,pid)`	`partition_edge_range_t<G>`	linear	`edges(g,uid)`
The following are only available when the optional `source_id(g,uv)` is defined for the edge			
`source_id(g,uv)`	`vertex_id_t<G>`	constant	n/a, optional
`source(g,uv)`	`vertex_t<G>`	constant	`*(begin(vertices(g))+ source_id(g,uv))` if `random_access_range<vertex_range_t<G>> && integral<target_id(g,uv)>`
`edge_id(g,uv)`	`edge_id_t<G>`	constant	`edge_descriptor<vertex_id_t<G>,true,void, void>{source_id(g,uv),target_id(g,uv)}`

Table 5: Graph Container Interface Functions

single value and can be scaler, struct, class, union, or tuple. These are abstract types used by the GVF, VVF and EVF function objects to retrieve values used by algorithms. As such it's valid to return the "enclosing" owning class (graph, vertex or edge), or some other embedded value in those objects.

`find_vertex(g,uid)` is constant complexity because all algorithms in this proposal require that `vertex_range_t<G>` is a random access range.

If the concept requirements for the default implementation aren't met by the graph container the function will need to be overridden.

9.1 Determining the Vertex Id Type

To determine the type for `vertex_id_t<G>` the following steps are taken, in order, to determine its type.

1. Use the type returned by `vertex_id(g,ui)` when overridden for a graph.
2. When the graph matches the pattern `random_access_range<forward_range<integral>>` or `random_access_range<forward_range<tuple<integral,...>>>` , use the `integral` type specified.
3. Use `size_t` in all other cases.

`vertex_id_t<G>` is defined by the type returned by `vertex_id(g)` and it defaults to the difference_type of the underlying container used for vertices (e.g int64_t for 64-bit systems). This is sufficient for all situations. However, there are often space and performance advantages if a smaller type is used, such as int32_t or even int16_t. It is recommended to consider overriding this function for optimal results, assuring that it is also large enough for the number of possible vertices and edges in the application. It will also need to be overridden if the implementation doesn't expose the vertices as a range.

10 Unipartite, Bipartite and Multipartite Graph Representation

`partition_count(g)` returns the number of partitions, or partiteness, of the graph. It has a range of 1 to n, where 1 identifies a unipartite graph, 2 is a bipartite graph, and a value of 2 or more can be considered a multipartite graph.

If a graph data structure doesn't support partitions then it is unipartite with one partition and partite functions will reflect that. For instance, `partition_count(g)` returns a value of 1, and `vertices(g,0)` (vertices in the first partition) will return a range that includes all vertices in the graph.

A partition identifies a type of a vertex, where the vertex value types are assumed to be uniform in each partition. This creates a dilemma because the existing `vertex_value(g,u)` returns a single type based template parameter for the vertex value type. Supporting multiple types can be addressed in different ways using C++ features. The key to remember is that the actual value used by algorithms is done by calling a function object that retrieves the value to be used. That function is specific to the graph data structure, using the partition to determine how to get the appropriate value.

— `std::variant` : The lambda returns the appropriate variant value based on the partition.

— Base class pointer: The lambda can call a member function to return the value based on the partition.

— `void*` : The lambda can cast the pointer to a concrete type based on the partition, and then return the appropriate value.

`edges(g,uid,pid)` and `edges(g,ui,pid)` filter the edges where the target is in the partition `pid` passed. This isn't needed for bipartite graphs.

11 Loading Graph Data

The `load` functions are used to load vertex and edge data into a graph. They may throw a `graph_error` exception.

All graph data structures need to implement `load_graph` , `load_vertices` and `load_edges` . Whether `load_vertices` or `load_edges` can be called multiple times, or after `load_graph` is called, is dependent on the underlying graph data structure. `load_partition` only needs to be implemented if a graph supports partitions.

Projections are used to convert values in the input range to the expected copyable type. In the following `load_vertices` prototype, `vproj(ranges::range_value_t<VRng>&)` → `vertex_descriptor<vertex_id_t<G>, vertex_value_t<G>>` . If there is no vertex value stored in the graph then `vertex_value_t<G>` will be `void` and the resulting `vertex_descriptor` will have a single id member. If `vproj(ranges::range_value_t<VRng>&)` is the same as `vertex_descriptor<vertex_id_t<G>, vertex_value_t<G>>` then `VProj = identity` can be used.

```
template <adjacency_list G, ranges::forward_range VRng, class VProj = identity>
requires copyable_vertex<invoke_result<VProj, ranges::range_value_t<VRng>>,
                         vertex_id_t<G>, vertex_value_t<G>>
constexpr void load_vertices(G&, const VRng& vrng, VProj vproj);
```

The same pattern is applied using `ERng` and `EProj` for edges.

For graphs with vertex values, `load_vertices` should be called before `load_edges` .

Whether `load_vertices` or `load_edges` can be called multiple times is graph-dependent.

For graphs with partitititions, `load_partition` must be called to load vertices for each partition `pid` . `pid` values must be contiguous and their vertices should be loaded contiguously. `empty(vrng)` may be empty if there are no vertices in the partition.

Function	Return Type	Complexity	Default Implementation
`load_graph(g, erng, vrng, eproj=identity(), vproj=identity())`	void	V + E	n/a
`load_vertices(g, vrng, vproj=identity())`	void	V	n/a
`load_partition(g, pid, vrng, vproj=identity())`	void	V(p)	`load_vertices` is called if partitions are not supported; there will be a single partition.
`load_edges(g, erng, eproj=identity(), vertex_count=0)`	void	E	n/a

Table 6: Graph Load Functions

12 Edgelists

An edgelist is a range of `edge_descriptor` values, used by some algorithms. Examples include the `edgelist` adjacency list view, or possibly a transform of an existing range that projects `edge_descriptor` values. There is no concrete data structure in this proposal like there is for an adjacency list.

13 Using Existing Graph Data Structures

Reasonable defaults have been defined for the GCI functions to minimize the amount of work needed to adapt an existing graph data structure to be used by the views and algorithms.

There are two cases supported. The first is for the use of standard containers to define the graph and the other is for a broader set of more complicated implementations.

This is described in more detail in the paper for Graph Library Container 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.