INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

C++ Graph Concepts for Partitioned Global Address Space

Stefan Effenberger

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

C++ Graph Concepts for Partitioned Global Address Space

Stefan Effenberger

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Tobias Fuchs
Abgabetermin:	ADD DATE

I hereby declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

Munich, ADD DATE

. . . . . .

(Signature)

Abstract

ADD ABSTRACT

Contents

1. Introduction

Many scientific projects are largely enabled by simulation. Because such simulations often require huge computational capabilities, single compute nodes with a shared-memory architecture cannot provide enough computation power and storage for numerous cases. For this reason, in High Performance Computing (HPC), work is distributed among multiple interconnected nodes to facilitate the solving of large problems in a timely manner. Since processors cannot directly access the memory of other nodes, the traditional programming model for such systems requires programmers to explicitly distribute data between nodes via message passing. This imposes high demands on the programming skills of scientists who might not have a background in computer science.

Therefore, with the Partitioned Global Address Space (PGAS) model, a new approach is proposed: The memory space of individual nodes in a system is unified within a global address space so that each node can directly access the memory of all other nodes. Programmers are still required to keep data access between nodes to a minimum because data transfer over an interconnect is costly. To further reduce the demands on the programmer, distributed data structures that handle data distribution and load balancing are needed.

Furthermore, data-intensive tasks have been gaining a continually growing interest in the scientific community. Traditionally, applications in HPC follow a computation-centric approach by solving numerical algorithms in the fastest possible way. As "Big Data" is becoming increasingly important in scientific projects, a shift towards more data-oriented applications can be observed in recent HPC projects [ZZZ+14]. This trend requires distributed data structures that allow for the storage of large amounts of irregular data and cater to the needs of ever-changing dynamic data.

1.1. Problem statement

Data can be represented in numerous ways. The most generic form of data representation is enabled by *graphs*. A graph $G(V, E)$ is a pair with a set of vertices V and a set of edges E that connect the vertices. This allows for the representation of data and its relationships in regular as well as irregular patterns.

On distributed machines, graph data structures can be implemented using a variety of different characteristics. This has led to many different implementations - usually a new implementation for each algorithm - which are hardly compatible with each other. To overcome this situation, generic programming abstractions to facilitate reuse of existing code and to lower the demands on programmers are needed.

As of today, no generic graph abstractions implementing the PGAS model exist. This work therefore aims to provide a graph abstraction for C++ containers that allows for the implementation of arbitrary graph algorithms following the PGAS model on distributed memory machines.

1.2. Scope and objectives

In this work, C++ concepts for graph containers following the PGAS model is presented. The graph concepts are meant to provide a generic framework for the programming of arbitrary graph algorithms in the context of distributed machines and especially the Partitioned Global Address Space model. This means that it meets the following requirements:

- Native support for one-sided communication
- Support for the programming of synchronous graph algorithms
- Support for the programming of asynchronous graph algorithms
- Portability across platforms and portable efficiency

Furthermore, this work provides concepts for the dynamic allocation of graph data across multiple machines with a focus on optimized data locality.

A reference implementation is then used to verify the usability, correctness and universality of the given concepts.

2. Background

This chapter covers some fundamental background knowledge needed for a better understanding of the following chapters of this thesis. Only explanations directly relevant to the topics of this thesis are provided.

Since the result of this work are C++ concepts, some important language expressions and concepts are firstly discussed, along with a description of the Standard Template Library [SL95] on which concepts this work is built upon. The reader is then introduced to the domain of High Performance Computing which is the main application area of this work. A brief overview of the Partitioned Global Address Space programming model is then followed by a description of the DASH Library [FFK16b] that serves as a framework for the reference implementation of this work.

2.1. Graph representations

This section provides a brief overview of different graph representations used in computer sciences [CLRS09]. Basically, there are two different representations with different but overlapping application areas.

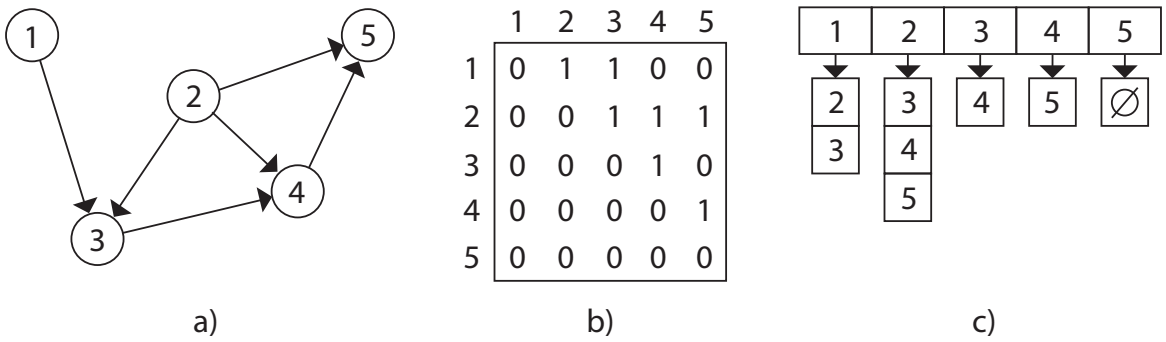

Figure 2.1.: A directed graph (a) that is represented as an adjacency matrix (b) and as an adjacency list (c)

Figure 2.1 shows a directed graph that is represented in two ways: *adjacency matrix* and *adjacency list*. Both representations are also suitable for *undirected* and *bidirectional* graphs, but not for *hypergraphs*.

2.1.1. Adjacency matrix

A graph $G(V, E)$ with a set of vertices V and a set of edges E can be represented by a $|V| \times |V|$ matrix $A = a_{ij}$ using a binary edge coding: If an edge (i, j) exists in the set of edges E, a_{ij} is set to 1. In all other cases, a_{ij} is set to 0.

A matrix data structure occupies a memory region for $|V|^2$ elements regardless of the amount of edges in the graph. For large, *sparse graphs*, this results in an unnecessarily high amount of required memory. A normal matrix therefore is only suitable for *dense graphs*. However, matrices containing a huge amount of elements with the same content (zeros in this case) can be compressed. For example, the *Compressed Sparse Row (CSR)* format [Saa03] allows to reduce the memory consumption of sparse matrices significantly. As can be seen in Figure 2.1 b), directed graphs only occupy the fields at the right side of the matrix' main diagonal. For undirected graphs, fields are symmetric along the diagonal. This means, for any directed, undirected or bidirectional graph, it is sufficient to only store the matrix values of one side of the diagonal thus further reducing the memory consumption.

2-dimensional matrices are stored in a linear, 1-dimensional order in memory. For this reason, adding new rows and columns to the matrix (i.e. adding new vertices to the graph) would require a complete re-allocation of the memory. Matrices are therefore not a suitable data structure for dynamic graphs for which the number of vertices is not known upon construction.

2.1.2. Adjacency list

An adjacency list is a set of vertices (preferably stored as an array) each pointing to an *edge list* of vertices adjacent to them. The data structure is highly dynamic: New vertices can be added to the vertex set and new edges to the corresponding edge lists in $O(1)$.

Attributes can be stored directly in an adjacency list by adding stored variables to vertices in the vertex set and edges in the edge lists. For adjacency matrices, these values have to be stored in external data structures.

Searching for a particular edge in an adjacency list is costly because the complete set of edge lists has to be searched whereas in an adjacency matrix, the information can be obtained straight from the corresponding memory location.

2.2. C++ concepts

The reference implementation of this work builds upon the DASH C++ Template Library (see section 2.5). For this reason, it is written in C++11 [ISO12]. This section illustrates some basic knowledge about important C++ concepts used in the implementation description of chapter 5.

2.2.1. Language concepts

This section describes features and concepts of the C++ language the reference implementation of this thesis is based on. While C++ offers many different features for different programming styles, only a certain subset is used due to performance reasons.

Value and reference semantics

In object oriented programming, objects can either have value or reference semantics. Objects with value semantics are treated like values: An assignment operation copies the object. This way, the identity of the object is not in focus because the copied object has another

identity. Operations on this copy do not affect the original object which means only the value of the object is in focus.

Objects with reference semantics on the other hand are referred to by references or pointers. Their identity becomes important: Multiple references can point to the same data.

Many object-oriented programming languages such as Java only offer reference semantics for non-primitive types. C++ on the other hand allows the programmer to define whether an object adheres to value semantics or reference semantics.

For an object to be able participating in value semantics, some operations like copy construction and assignment have to be implemented in a certain way. C++ compilers provide default implementations of the required operations, but depending on the object's implementation, further measures might have to be taken by programmers. For example, objects that allocate memory dynamically on the *freestore* have to explicitly copy the referenced data.

Listing 2.1 shows an `increment` function with both value and reference semantics. `increment_vs` takes the copy of a `VertexIndex` object as parameter, increments its offset and returns a copy of the object with incremented offset. The offset of the passed `VertexIndex` stays the same as only the offset of its copy has been changed. In contrast, `increment_rs` changes the offset of the passed object and also returns a reference to the same object.

Listing 2.1: Value and reference semantics

```
1 VertexIndex increment_vs(VertexIndex v) { // value semantics
2   return ++(v.offset);
3 }
4
5 VertexIndex & increment_rs(VertexIndex & v) { // reference semantics
6   return ++(v.offset);
7 }
```

Value semantics seem to result in a lot of copying that might hit performance. C++ compilers implement an optimization technique called *copy elision* that omits copy construction in functions returning objects with value semantics by returning the same memory location of the temporarily created object. *Copy elision* is part of the C++ standard and thus compilers are required to enforce it. For this reason, C++ value semantics in many cases have no performance drawbacks in comparison to reference semantics. Therefore, reference semantics is mainly used when identity of an object is important or when copying of an object is expensive.

Reference semantics is also important for runtime polymorphism, because objects with value semantics might be sliced (i.e. only the part of the base class is copied).

Operator overloading

In C++, almost all existing operators can be overloaded for any operand types. Any class can therefore be handled with native operators in a completely customized way. Only four operators like the member access operator cannot be overloaded and it is not possible to create new operators that do not exist in the language itself.

Listing 2.2: Operator overloading

```
class Iterator {
  public:
    Iterator & operator++() {
      ++position;
      return *this;
    }
  private:
    int position = 0;
};

Iterator it;
++it;
```

Listing 2.2 shows a class `Iterator` with an overloaded version of the pre-increment operator. The operator can be directly applied to an object of `Iterator` incrementing the `position` member.

Static vs. runtime polymorphism

C++ offers two kinds of polymorphism: *static* and *runtime*. The difference lies in the way types are bound. *Static polymorphism* can be completely resolved at compile time, while types in *runtime polymorphism* have to be resolved during the runtime of a program.

In runtime polymorphism, methods of a derived class are called with a pointer of the base class by using *virtual functions* of the base class. A call to such a virtual function requires resolving which concrete derived class the pointer of the base class refers to during runtime. This is achieved by storing a *vtable* for each base class in memory and linking to this *vtable* with a pointer from all related objects. The runtime can then lookup the address of the desired class's method. Listing 2.3 shows how a method implemented in a derived class is executed calling the same method on a pointer of the base class.

Listing 2.3: Runtime polymorphism

```
class Base {
  virtual void do_something() {
    // do something
  }
};

class Derived : Base {
  virtual void do_something() {
    // do something else
  }
};

Base * b;
Derived d;
b = &d;
b.doSomething(); // Derived::do_something() is called
```

Because the *vtable* lookup on every call is expensive, C++ allows programmers to design polymorphic types with static polymorphism that can be completely resolved at compile time leading to better performance during runtime. This can be achieved with simple method overloading and with templates.

In C++11, templates can be defined for classes and functions. They allow the programmer to define a family of either. Listing 2.4 shows a class `Base` that accepts a template parameter which can be of any type. A call to `do_something` is delegated to an object of the type of the template parameter. Type resolving is done during compilation of the program so that a compile error would occur if no `do_something` method were available in `TypeB`.

Listing 2.4: Static polymorphism with class templates

```
struct TypeA {
  void do_something() {
    // do something
  }
}

struct TypeB {
  void do_something() {
    // do something else
  }
}

template<typename Type>
class Base {
  public:
    void do_something() {
      type.do_something();
    }
  private:
    Type type;
};

Base<TypeB> b;
b.do_something(); // TypeB::do_something() is called
```

Templates actually result in code generation: For every instantiation of a class template, a new type is created which results in a larger binary file.

2.2.2. Standard Template Library

The *Standard Template Library (STL)* [SL95] is a C++ software library. Most parts of it have been integrated into the C++ Standard and are now part of the *C++ Standard Library*. The STL has been designed as a library for efficient generic programming with value semantics. Its containers are template classes that enable static polymorphism.

The STL contains a variety of containers and algorithms. Algorithms are decoupled from containers with the help of iterators so that any algorithm can work with any container that is STL-compatible. Because DASH - the library this work's reference implementation is part

of (see section 2.5) - follows the concepts of the STL, STL algorithms can be executed with any DASH container, including the graph container of this work's reference implementation.

This section focuses on STL concepts in the C++11 standard.

Concepts

In C++11, a *concept* is a named set of requirements for a type. The C++ standard contains concepts for all components of the C++ standard library and this work provides a concept similar to these. At the time of this writing a C++ extension that allows for the formal specification and compile-time evaluation of concepts directly in code is in development [ISO15]. Because this extension is still work in progress and not part of C++11, it is not used in this thesis. Thus, the graph concepts of this work require the programmer to manually ensure that all requirements of the respective concept are met in their implementation.

A C++ concept can be derived from other, existing concepts. For example, the `SequenceContainer` concept of the C++ standard implements the `Container` concept and additionally accounts for linear arrangement of the contained elements. *Types* and *Methods/Operations* are defined along with their semantics. For some expressions, *computational complexity* is additionally defined and programmers have to keep their implementations inside of these constraints.

The graph concepts of this work try to assimilate the concepts of the standard library as far as possible.

Iterators

Iterators are the connection between *containers* and *algorithms*. Every container offers iterators with a standard interface to allow for the iteration of contained elements. Algorithms only have to comply to iterator interfaces and are agnostic of the actual interface of the container. Thus, the same algorithm can run with a variety of different containers without it being re-implemented for every existing container.

Listing 2.5 shows the computation of a sum of vector `v` using the `accumulate` algorithm of the C++ standard library. `v.begin()` returns an iterator to the beginning of `v` and `v.end()` an iterator past the last element of `v`. The `accumulate` algorithm then iterates over the elements of `v` without knowing the details of its underlying container.

Listing 2.5: Vector sum using standard algorithm

```
1 std::vector<int> v = { 1 , 2, 3 };
2 int sum = std::accumulate(v.begin(), v.end(), 0);
```

Because containers have different memory layouts and algorithms have different requirements on iterators, the C++11 standard defines four iterator concepts that are hierarchically organized, i.e. iterators of a higher category implement all operations of the iterators in lower categories. Table 2.1 shows these concepts starting from the lowest category:

Table 2.1.: STL iterator categories

Category	operations
InputIterator	increment, read
ForwardIterator	multi-pass increment
BidirectionalIterator	decrement
RandomAccessIterator	random access

`InputIterators` can be incremented to iterate over elements one by one. There is no guarantee that a value dereferenced from an `InputIterator` is still valid after it has been incremented again. This guarantee is only given by `ForwardIterator`s which makes them a requirement for algorithms that have to iterate over elements more than once. `BidirectionalIterator`s can be decremented and thus are able to iterate in opposing direction. `RandomAccessIterator`s can access any element in a given range in constant time.

Any iterator can also implement the `OutputIterator` concept that enables writing operations on elements.

Containers

Standard C++ containers implement common data structures like arrays and stacks. The container library consists of *sequence containers* that store elements in a specific order and *associative containers* that allow searching for particular elements. Additionally, *adaptors* like `std::queue` are used to restrict the interfaces of existing containers.

All concepts of concrete implementations meet the requirements of the standard library's `Container` concept that defines element access via `ForwardIterator`s. Containers can additionally meet the requirements of further concepts which might override certain requirements like the iterator type. The `ReversibleContainer` concept for example allows containers to either use `BidirectionalIterator`s or `RandomAccessIterator`s. A concrete container concept like `std::vector` that meets the requirements of `ReversibleContainer` can then define the iterator type as needed (`RandomAccessIterator` in this case).

2.3. High Performance Computing

High Performance Computing (HPC) is a broad term describing advances for the fastest possible computation of a given problem. Gustafson's Law [Gus88] suggests that a compute system can linearly grow with the problem size: A problem of two times its original size can be computed on a system with twice as many processors in the same time (best case scenario). This means that very large problems can be computed in an acceptable timeframe if there is a sufficiently large compute system available. Depending on the problem size, two different system architectures are used in HPC:

Shared Memory A shared memory system consists of a single node with multiple processors connected to the same random access memory. Memory access for the different processors can be uniform, but many systems implement a non-uniform memory access (NUMA) design where a part of the memory is assigned to each of the processors. A processor in a NUMA system can access its assigned memory faster than the memory of the other processors.

Because processors can access all data at all times, communication between processors has a low cost which simplifies programming on these systems in comparison to distributed memory systems. Achieving high performance on NUMA systems is more problematic because the programmer has to take data locality into account [Lam13].

Distributed Memory Multi-processor systems in which each processor has access to its own memory space are called distributed memory systems. These systems usually consist of several shared memory nodes with the processors of one node not being able to directly access memory of other nodes. While single shared memory systems can only be scaled to a certain extent, the scalability of distributed systems is much higher [PTM96].

The nodes are connected with a network interconnect for communication between the processors. Due to the latency of the interconnect being significantly higher than the latency of a memory bus in a shared memory system, communication is much more costly. This imposes higher demands on the programmers' skills in comparison to shared memory systems: A proper *domain decomposition* has to be performed that partitions the data in a way which allows for minimization of communication between the participating machines.

The largest problems in science are computed on "supercomputers" like the *SuperMUC* at the *Leibniz Supercomputing Centre* in Munich. These distributed memory machines consist of hundreds or even thousands of homogeneous nodes that are connected with a specialized interconnect. To this date, *message passing* is the prevalent programming model for such systems.

2.4. Partitioned Global Address Space

Shared Memory and *Message Passing* are the dominant models in HPC as of this writing. As pointed out in section 2.3 however, the usage of Message Passing requires high skills in computer architecture and programming. To ease this problem, the Partitioned Global Address Space (PGAS) model has been proposed. It unifies some of the benefits of both of these models by creating a global address space over the initially local-only address spaces of distributed machines.

Figure 2.2 a) presents the architecture of a shared-memory machine: Multiple processors share a common address space. The processors are attached to the same memory over a bus. In some systems, memory might be local to some processors which means the rest of the processors has a higher latency when trying to access the non-local memory. Still, every processor can access every part of the address space. Communication takes place *implicitly* by writing and reading shared variables. Because data written by one processor can be accessed by another processor in a fast manner, little care has to be taken regarding the decomposition of data. For this reason however, shared memory programs are typically not scalable on distributed machines [SAB+10].

Figure 2.2 b) shows that a distributed memory machine basically consists of several shared memory machines linked to each other via an interconnect. Since processors cannot directly access data stored in the memory of other machines, *explicit communication* is needed in order to synchronize the processors. This is typically done by two-sided communication:

The *sending* of a message has to be accepted at the remote machine with a a corresponding *receive* call.

Machines conduct their computations simultaneously and either synchronize in discrete time intervals or exchange data asynchronously. Either way, sending data over an interconnect imposes high latency and low throughput in comparison to the data access over a memory bus in shared memory systems. For this reason, programmers have to carefully decompose data in order to distribute the work load uniformly and minimize communication overhead.

Figure 2.2 c) illustrates the concept of Partitioned Global Address Space: The local portions of memory are unified under a global address space which allows processors to directly access data on remote machines. Data access is performed using one-sided communication: No *receive* call on the remote machine is needed.

Since data transferal over an interconnect is still costly, programmers have to take the same care for data locality as with the traditional message passing approach. To allow for this, the locality of a datum is directly exposed to the programmer.

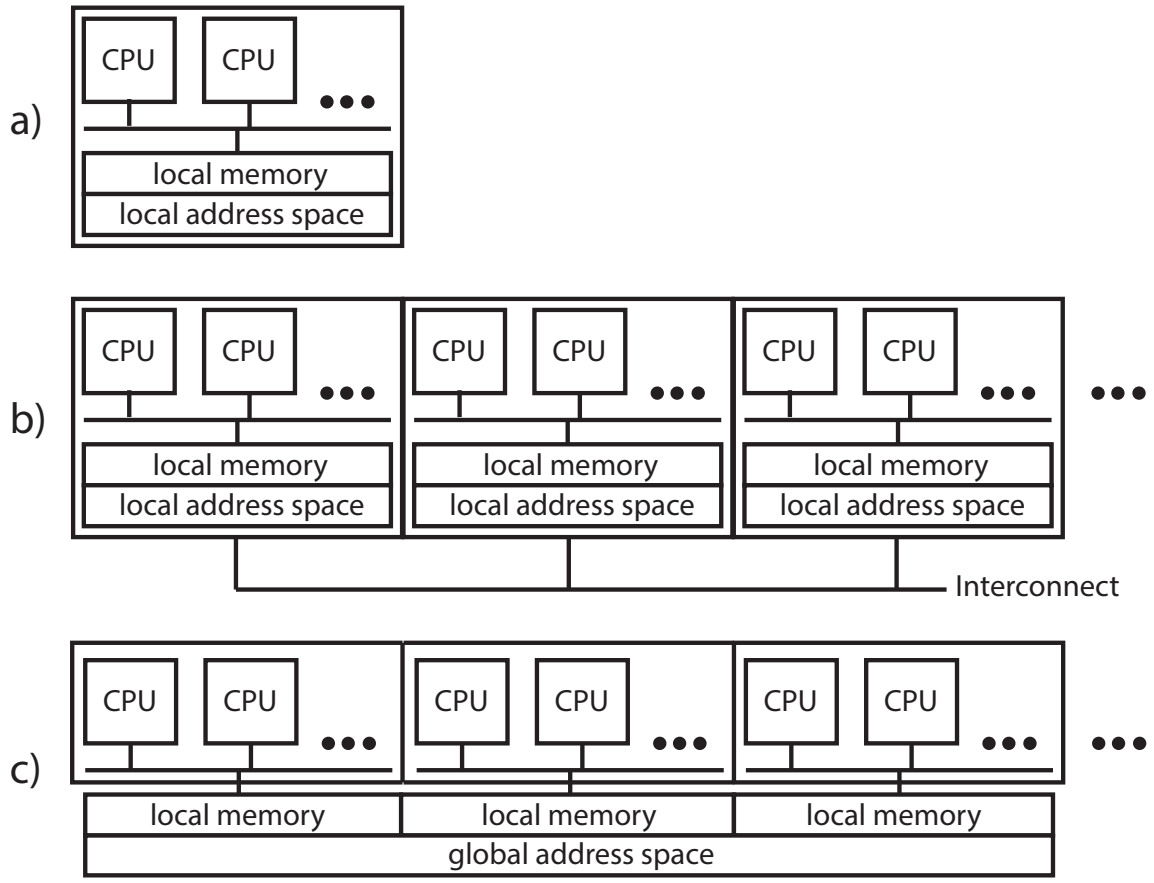

Figure 2.2.: View on Shared Memory (a), Distributed Memory (b) and Partitioned Global Address Space (c)

Existing PGAS approaches are mainly comprised of dedicated programming languages

such as Unified Parallel C (UPC) [C+05], Co-Array Fortran [NR98] or Chapel [CCZ07] that allow for compiler optimizations in respect to distributed machines but lack portability and reach. In contrast to this, efforts exist to create libraries for existing programming languages used by many HPC systems.

2.5. DASH C++ Template Library

DASH [FFK16b] is a compiler-free PGAS approach: It consists of a simple C++ library that can be compiled with any C++ compiler and thus can be used out-of-the-box on most HPC systems. The library is part of the Priority Programme "Software for Exascale Computing" (SPPEXA)[1] which supports research on computing systems achieving 10^{18} floating point operations per second and above. While PGAS languages require existing programs to be completely rewritten from scratch, DASH allows the applications to be incrementally ported and thus facilitates wider adoption of the PGAS model in the HPC community.

DASH operates on top of the *DASH Runtime* (DART) which is a PGAS memory allocation and communication abstraction written in C. DART enables global memory allocation, pointers to remote memory locations and one-sided communication on top of existing libraries like MPI [For12] or GASPI [GS13]. With DART-MPI [ZMI+14], a fully functional DART abstraction on top of MPI-3 is used in DASH releases at the time of this writing.

In DASH, processing elements are referred to as *units*. Units can be any processing element such as threads or processes. DASH programs are implemented using the Single Program Multiple Data (SPMD) model: The data is partitioned onto the participating units and each unit executes the same code on its part of the data. Furthermore, units form *teams* that can be created at runtime. Because HPC hardware topologies become more complex over time (e.g. [KDSA08]), DASH supports hierarchical team creation to allow for a more fine-grained exploitation of data locality compared to the typical local-remote distinction of the PGAS model.

Data is referred to in terms of global pointers and references. A `GlobPtr<T>` object holds information about the unit and local memory location of the referenced datum. It can be dereferenced to a `GlobRef<T>` object which behaves like a C++ reference and can be converted to an object of type `T`. This type conversion triggers a one-sided `get` operation transferring the data from its remote source to the caller. Similarly, data can be written into the referenced memory location of a `GlobRef<T>` object.

DASH provides a set of containers for distributed data storage. Aside from the static data structures Array and Matrix, dynamic data structures are available. Since the graph concepts of this work belong into the latter category, details of it are discussed in the following.

2.5.1. Dynamic memory allocation

Dynamic allocation in DASH is encapsulated in the `GlobHeapMem` concept. `GlobHeapMem` offers two basic operations to dynamically allocate memory during runtime: `grow` and `shrink`. These operations increase or decrease the local size of the memory allocated on the respective unit. Changes in memory space are not reflected in global address space until the operation `commit` is called which publishes the changes across all units.

[1] `http://www.sppexa.de`

A dynamic container in DASH pre-allocates some memory during its initialization. When the memory is completely used, further additions of elements result in `GlobHeapMem.grow` operations. A call to the `barrier` operation of the container results in all newly added elements of the container to be publicly available on all units.

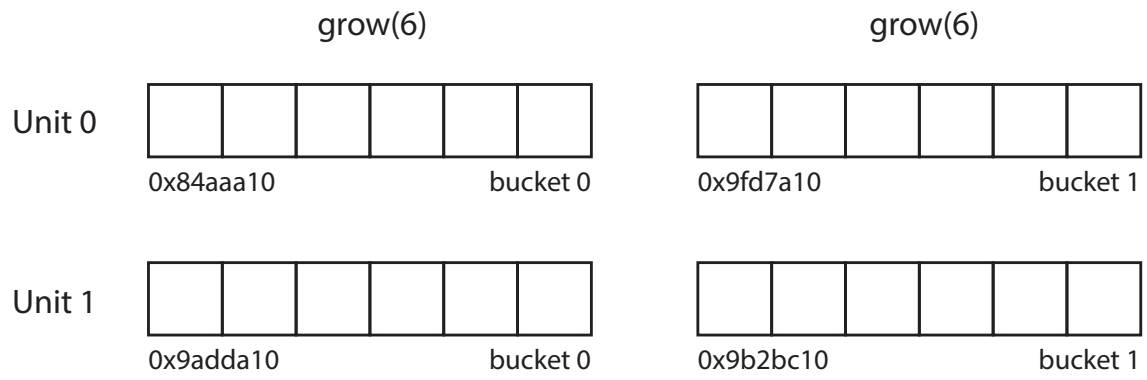

Figure 2.3.: Memory space of two units after two `GlobHeapMem.grow` operations

Multiple `grow` operations result in a scattered memory space: Each call to `grow` creates a new *bucket* - a contiguous memory region in the local freestore. A class implementing the `GlobHeapMem` concept keeps track of each bucket on each unit so that element indices can be translated to concrete memory locations. Figure 2.3 illustrates the memory spaces of two units after two grow operations for six elements each have been called. The buckets are allocated at different memory locations. Data access to an element therefore requires the memory location of the bucket and the offset of the element inside the bucket. For this reason, bucket locations and sizes are exchanged between all units during a `commit` operation and a reference to the object holding the bucket data is needed in every iterator.

3. Related work

3.1. Shared Memory

Graph processing on distributed machines tends to result in low processor utilization due to bad locality [LGHB07]. Shared memory systems on the other hand allow for simpler graph algorithms with better processor utilization [SB13]. Because some shared memory systems with several terabytes of memory exist (e.g. the LRZ teramem[1]), graphs up to a certain size can also be computed on such systems. While many libraries and frameworks for graph processing on shared memory systems exist, this section only focuses on work relevant to the HPC community.

Spatio-Temporal Interaction Networks and Graphs Extensible Representation (STINGER) [BBAB+09] is a graph data structure for shared memory systems. STINGER tries to provide a standardized and extensible data structure that allows portability across different graph processing frameworks. The data structure consists of a set of standards that have to be abided by actual implementations.

At its core, STINGER is an adjacency list with a fixed structure. Vertices are stored in a contiguous memory region and point to blocks of edges. The set of all edges of a single vertex can be distributed between multiple blocks with each block containing a pointer to the next block. Vertices and edges both store a weight value and edges additionally store timestamps. This allows edges to be automatically removed after they have reached a certain age. STINGER also accounts for fault tolerance by defining standards for checkpointing functionality.

The **MultiThreaded Graph Library (MTGL)** [BHKK07] focuses on graph processing using specialized multithreading systems that contain processors supporting a large amount of thread contexts. The MTGL is specifically designed for the Cray MTA architecture which limits its use significantly because a Cray MTA system has to be available. Brian W. Barrett et al. have implemented an MTGL version for commodity hardware [BBMW09] but its performance is significantly lower compared to the original version for serial machines.

The Cray MTA architecture [CFS99] utilizes processors with relatively slow speed and without data caches. Memory access is always blocking but each processor features a large amount of threads so that there are always free threads available for computation in most cases. This results in a simple and fine-grained synchronization model that is not available on commodity hardware, but also creates subtle concurrency and performance issues. These issues are addressed by the MTGL in order to allow straight-forward programming of graph algorithms on these platforms.

Because of the architecture of the Cray MTA, locality is not an issue in the MTGL: Data is partitioned by the runtime system of the MTA. The interface of the MTGL is - like many

[1] `https://www.lrz.de/services/compute/special_systems/teramem/`

graph processing frameworks - based on the Boost Graph Library (BGL) [SLL01]. It is less generic though: Only *directed graphs* are supported and data is always stored in an *adjacency list* structure.

3.2. Distributed Memory

Procession of large graphs that do not fit into the memory of a single machine takes place on distributed memory machines in HPC. Shared memory approaches that hold parts of the graph in persistent memory (e.g. GraphChi [KBG12]) do not have sufficient performance in order to be useful in the field of high performance graph processing. However, because some graphs like scale-free social networks lack locality, graph partitioning on distributed memory machines is a challenge that can result in unpredictable performance due to low processor utilization [BHKK07]. While performance and scalability are key capabilities of the work mentioned in this section, they can not always be guaranteed.

3.2.1. Bulk-synchronous graph processing

In *Parallel Computing*, Valiant's *Bulk Synchronous Parallel (BSP)* model [Val90] describes a way to exploit locality: Data is partitioned across processors in a way that allows each processor to perform independent computations. BSP algorithms run a series of *supersteps*, each consisting of three phases:

1. Local computation on each processor.
2. Global exchange of computed data between processors.
3. Synchronization of all processors.

The performance of BSP algorithms relies heavily on a well-suited decomposition of the data. Because of the explicit barrier synchronization at the end of each superstep, processor time may be wasted on processors that finish their work faster than others. Deadlocks and data races are however prevented by it.

The *Boost Graph Library (BGL)* [SLL01] is a widely used sequential graph processing framework. With the **Parallel Boost Graph Library (PBGL)** [GL05], the BGL has been ported to the environment of distributed machines using the BSP model. It aims at providing a general-purpose library for graph processing on distributed machines. The PBGL data structures adjacency list and adjacency matrix therefore are designed as generic as possible and the library includes a wide variety of different graph algorithms that can be extended with the *Visitor Pattern* [PJ98]. Attributes of graph components are not restricted to simple edge weights: The PBGL allows programmers to define arbitrary attribute data structures for vertices and edges. Supported graph types in the PBGL include directed, undirected and bidirectional graphs.

Pregel [MAB+10] is a framework for large-scale graph processing developed by Google. Like the PBGL, it partitions graphs by vertex and uses the BSP model for computation. Vertices are statically partitioned by a user-defined hash function. Pregel additionally accounts for fault tolerance with checkpointing mechanisms but only supports directed graphs.

Programming in Pregel is restrictive because it only allows the definition of vertex visitors that have access to messages sent by other vertices in the previous superstep.

The static partitioning mechanism of Pregel can be replaced with a dynamic load balancing system called Mizan [KAA+13]. It monitors Pregel graphs during runtime and migrates vertices to other partitions based on its observations.

ScaleGraph [SU15] is a graph processing library written in IBM's X10 [CGS+05] PGAS language. ScaleGraph supports graphs of arbitrary size. This means, it can also process graphs that do not fit completely into the memory of the used compute system. ScaleGraph supports dynamic graphs stored in an adjacency list data structure and static graphs stored in a sparse matrix data structure. For dynamic graphs, only directed edges are supported but ScaleGraph allows the programmer to define arbitrary attributes for vertices and edges. Because of bad performance experiences in the first version of the library [DHS12], the garbage collection of X10 is disabled for big graphs. On top of the ScaleGraph core, Toyotaro Suzumura et al. have added an API based on the model of Pregel (see subsection 3.2.1). Therefore, while the library is implemented using a PGAS language, it does not provide PGAS functionality to the user and rather relies on the BSP model of Pregel.

3.2.2. Asynchronous graph processing

The BSP model requires computation steps to be coarse-grained. This means that computations have to be performed on sufficiently large parts of the data that belong together to minimize costly communication. Some graphs however demand more fine-grained communication [EWHL10] which would require many BSP iterations with short computation steps resulting in low processor utilization. To account for these kind of graphs, algorithms that do not require explicit synchronization steps can be formulated to run asynchronously. Asynchronous execution of algorithms is especially useful in the field of machine learning [LBG+12]. Graph processing libraries have to explicitly support asynchronous communication because it requires atomic access to vertex and edge attributes.

Nicholas Edmonds et al. present a **graph processing library based on Active Messages** in [EWHL10]. It is based on the PBGL and tries to act as a next generation of the library. The library adds support for asynchronous active messages [ECGS92] that replace the two-sided communication used in the PBGL and transactions for access to graph component attributes. At the time of this writing, the library has not been released.

The **STAPL Parallel Graph Library (SGL)** [FAR+12] is a PGAS library that aims at providing an API similar to sequential graph processing libraries. Thus, it tries to abstract data distribution by offering automated partitioning and load-balancing methods that achieve high scalability. Algorithms in the SGL can either be run bulk-synchronously or asynchronously. With an adjacency list data structure at its core, the SGL graph container is dynamic and a distributed directory is used to identify the location of vertices. This allows for the migration of vertices to other processors and redistribution schemes for load-balancing. The SGL is a promising approach in the field of PGAS graph processing libraries but as of this writing, no release or documentation is to be found.

3.2.3. Linear algebra based graph processing

Many graph algorithms can be expressed by a set of linear algebra operations [FR11]. The graph is represented by an adjacency matrix storing edge weights and additional vectors are used as auxiliary data structures.

Because an adjacency matrix can be partitioned into blocks with each block being placed into the memory of a single processor, the graph can be partitioned in a 2-dimensional way. As research has shown, breadth-first-search and related algorithms can be implemented in a way that exploits the 2-D partitioning for higher performance in comparison to a 1-dimensional partitioning (i.e. partitioning based on vertices instead of edges) [BM11].

The **Combinatorial BLAS** [BG11] exploits these findings with a library containing *Basic Linear Algebra Subroutines* (BLAS) specifically targeted at graph processing. It provides a subset of typical BLAS operations like matrix-matrix multiplication that are sufficient for expressing most graph algorithms. Graph data is stored in a sparse matrix data structure. Vertices are therefore not mutable which results in graphs being static. Also, algorithms like Dijkstra's shortest path, that are based on priority queues, tend to have lower performance.

The Combinatorial BLAS is also used as a computational engine for a high-level graph processing framework: the **Knowledge Discovery Toolbox (KDT)** [LAB+12]. The KDT aims at easing use of graph algorithms for domain experts that are not experts in computer science at the same time. To achieve this, it provides an API that allows to run algorithms on graphs with only few lines of code. Algorithm developers can extend the library by creating algorithms with the underlying linear algebra primitives. While KDT can be easily used by data scientists without background in computer sciences, the performance and scalability are still promising: In comparison to the PBGL, speedups from 3 to 12 for breadth-first search can be observed depending on the problem size and core count. Because it uses the Combinatorial BLAS as underlying engine, KDT also inherits its restrictions explained above.

For static graphs, **ScaleGraph** (see subsection 3.2.1) also supports linear algebra primitives that can be used in the same way as with the Combinatorial BLAS.

4. Graph container concepts

4.1. Problem fundamentals

A generic graph container must enable programmers to implement arbitrary graph algorithms with it. For this reason, an analysis of the problem domain is shown in this section. It results in functional requirements that are then used to deduce graph container concepts in later parts of this chapter.

4.1.1. Elementary graph algorithms

Most graph algorithms are based on two elementary graph algorithms: *breadth-first search* and *depth-first search* [CLRS09]. Support for these algorithms and any extensions to them therefore is an important requirement for a generic graph container. This section shortly describes both algorithms as well as selected algorithms extending them and analyses their requirements.

Breadth-first search

Breadth-first search (BFS) finds all vertices reachable from a source vertex in a graph. The graph is traversed in a way that all vertices are found with the minimum number of edges connecting them to the source vertex: Starting from a certain vertex, all adjacent vertices are explored before their adjacent vertices are explored. All adjacent vertices of a given vertex are called *frontier* or *level*. BFS explores the vertices of a frontier and creates a new frontier for the next step that contains the adjacent vertices of the vertices in the current frontier.

Observation 4.1 *For successful graph traversal, adjacency information has to be accessible to any algorithm.*

Because graphs tend to contain cycles, BFS employs a mechanism for already discovered vertices by graph coloring: All vertices start with a white color. As soon as a vertex is discovered for the first time, its color is changed to grey. Its color is changed to black once all of its adjacent vertices have been discovered. This results in the frontier being colored grey and all other processed vertices being colored black so that the traversal can be progressed in a breadth-first manner.

BFS can be used to construct a *breadth-first tree* that contains all paths from the source vertex to its reachable vertices.

Observation 4.2 *For BFS to work, a graph container must be able to store mutable attributes for vertices: color and predecessor. A breadth-first tree might be later extracted from the predecessor attributes but it can be convenient if the graph container allows the tree to be constructed in an external data structure during computation of the algorithm.*

Observation 4.3 *Frontiers need to be managed with a queue or a similar data structure. This data structure might be external or part of the graph container itself. For distributed environments, the vertices of the frontier must be distributed to the nodes holding the respective vertices. The queue data structure therefore has to be distributed.*

Because BFS naturally finds the paths with the least edges between vertices, it is obvious that it can be used for *shortest path computation*. **Dijkstra's shortest path algorithm** [Dij59] for example is an extension to BFS: Each edge is associated with a distance attribute. Progressing in a breadth-first manner, the smallest sum of edge distances is saved in vertices as an additional attribute. If a vertex is discovered from an edge with a smaller distance sum, the attribute's content is replaced with this new sum.

Similar to Dijkstra's shortest path algorithm, a Minimum Spanning Tree algorithm such as **Prim's algorithm** [Pri57] traverses a graph in breadth-first order and creates a tree based on the minimum of edge weight attributes for each frontier.

Observation 4.4 *Graph containers do not only have to support vertex attributes. Edge attributes are also relevant. While Dijsktra's shortest path algorithm only requires a single attribute for edges, other algorithms might require more.*

Depth-first search

Depth-first search (DFS) explores vertices starting from a source vertex just like BFS. It differentiates from BFS in that it processes vertices in another order: Instead of exploring the vertices of a frontier one after another, DFS explores adjacent vertices directly from the most recently discovered vertex. If all adjacent vertices have been completely explored, DFS uses backtracking, i.e. it returns to the vertex from which the current vertex has been explored, and progresses there. Once finished, DFS proceeds to start another iteration from an arbitrary vertex that has not been discovered in the first iteration - contrary to BFS.

DFS uses a graph coloring scheme similar to BFS. Vertices visited for the first time are colored grey and when the backtracking takes effect, the respective vertex is colored black meaning that all edges from this vertex have been explored. DFS does not need a supporting data structure but apart from that has the same requirements on graph containers as BFS.

Some algorithms use DFS as a subroutine. For example, a **topological sort** of a graph runs DFS and then creates a list with sorted vertices based on the findings of the DFS subroutine. A **Strongly Connected Components** algorithm runs DFS on the graph and then on the transposed version of the graph (i.e. the graph with reversed edges).

Observation 4.5 *Some algorithms have to access graph attributes after another algorithm has stored them inside the graph. For this reason, attributes of all vertices and edges have to be persistent and accessible across single algorithms.*

Because BFS and DFS traverse a graph in such a fundamentally different way, it becomes obvious why locality is problematic with distributed graphs: If a graph partitioning results in high locality exploitation for BFS, it will result in low locality exploitation for DFS and vice versa.

Observation 4.6 *Partitioning of distributed graphs is dependent on locality information in the graph data itself and the algorithm used. Therefore, no generally valid partitioning scheme can be employed and locality information about a graph as well as information about the used algorithms is needed in advance for a reasonable partitioning.*

4.1.2. Functional requirements

Based on the observations of subsection 4.1.1, the following functional requirements for a PGAS graph abstraction have been deduced:

Requirement	**Notes**
Vertex element creation	Creation of single vertices inside the graph data structure
Edge element creation	Creation of single directed or undirected edges inside the graph data structure
Vertex element deletion	Removal of single vertices from the graph data structure
Edge element deletion	Removal of single edges from the graph data structure
Global element iteration	Iteration over vertices, edges and adjacent vertices of the whole data structure
Local element iteration	Iteration over vertices, edges and adjacent vertices of parts of the data structure hosted on the same machine
Vertex attribute creation	Permanent storage of attributes belonging to particular vertices along with these vertices
Edge attribute creation	Permanent storage of attributes belonging to particular edges along with these edges
Vertex attribute mutation	Adjustment of attributes belonging to particular vertices
Edge attribute mutation	Adjustment of attributes belonging to particular edges
Global, asynchronous element access	Access to elements on other machines without synchronization enforcement
Global, asynchronous attribute access	Access to attributes belonging to particular elements on other machines without synchronization enforcement
User-specified data distribution	Storage location of elements and attributes specified by the user
Global dynamic memory allocation	Allocation of memory for elements and attributes across multiple machines during runtime
Global dynamic memory de-allocation	Freeing of memory for elements and attributes across multiple machines during runtime

Requirement	Notes
Epoch-based memory space synchronization	Explicit synchronization of newly added/removed memory space between machines in user-defined time-steps
Container meta-data retrieval	Retrieval of container meta-data like **number of vertices** during runtime

4.2. Container concepts

This section describes graph container concepts that have been deduced from the related work analysis of chapter 3 and the graph algorithm analysis of subsection 4.1.1. The concepts follow the C++ standard library's container concept schemes. A complete concept definition can be found in appendix A.

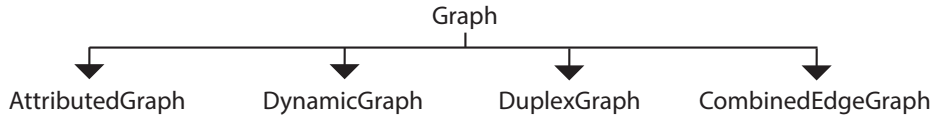

Figure 4.1.: Hierarchy of graph container concepts

Figure 4.1 depicts the different concepts and their relationships to each other. The `Graph` concept provides basic functionality for graph processing while the other concepts refine it with additional functionality.

4.2.1. Graph

The `Graph` concept describes a graph container for static data that is suitable for *directed graphs*. It consists of a constructor that creates a graph based on an existing *edge list*.

The constructor takes two iterators of type `InputIterator` which allows the edge list to be of any STL-compatible container type (e.g. `std::list`). An edge list has to contain elements of type `std::pair` with both contained types being integers describing unique vertices. The constructor then creates the graph by adding vertices based on these integers and connecting them according to the given vertex pairs.

Vertices and edges are represented by their own types. Edge types have to contain data identifying the source and target vertices. Since the `Graph` concept does not comprise an index space, iterators may be used for this task depending on their implementation. The concept is however compatible to index spaces being implemented on top of it.

Vertices and edges are assigned to the local memory of a node if the corresponding edge list has been assigned to the constructor of this node. Nodes can then iterate over elements either locally (over the elements that directly reside in the node's memory) or globally (over all elements in the graph) by calling the respective iterators. Due to the default `Graph` concept describing a static graph, no more elements can be added or removed after the constructor has been called.

The `Graph` concept contains two kinds of iterators:

- Vertex iterators

- Out-edge iterators

This is the minimum needed for adjacency iteration: Each node can iterate over the set of all vertices or its local portion of the vertices and for each vertex, the outgoing edges can be iterated. Global iterators should facilitate a mechanism for direct local dereferencing if the respective element resides on the local machine of the calling node.

Listing 4.1 shows an adjacency iteration over all outgoing edges of local vertices. Since all edges belonging to a particular vertex reside on the same node, only local iterators are available for adjacency iteration of a given vertex identified by a local iterator (`it`).

Listing 4.1: Adjacency iteration with Graph concept

```
1 std::list<std::pair<int, int>> edge_list = { {1, 2}, {1, 3}, {4, 1} };
2 Graph g(4, 3, edge_list.begin(), edge_list.end());
3
4 for(auto v_it = g.vertices().lbegin(); v_it != g.vertices().lend(); ++v_it) {
5   for(auto e_it = g[v_it].out_edges().lbegin(); e_it !=
        g[v_it].out_edges().lend(); ++e_it) {
6     Graph::edge_type e = *e_it;
7   }
8 }
```

4.2.2. DynamicGraph

The `DynamicGraph` concept extends the `Graph` concept with functionality for dynamic addition and deletion of graph elements during runtime. Because elements can be added after construction of the graph, an upfront initialization with an edge list is not required here. For this reason, a new constructor allows the graph to be created without any elements. It initializes the graph with a certain capacity so that each node has a fixed part of memory already assigned to it. Contrary to the constructor of `Graph`, the number of edges is not an absolute number but rather the number of edges per vertex. This means that the reserved capacity for edges equals to $number of vertices * number of edges$. The vertex and edge numbers are supplied for the whole graph, meaning that they have to be multiples of the amount of nodes participating in the memory allocation for the container.

All methods for adding and removing vertices and edges can be called with either local or global iterators pointing to them.

Dynamic data changes of the graph are not reflected in global memory space until the user explicitly issues a `commit` operation. Before that, the changes can only be seen on the same node using local iterators. A `commit` operation can invalidate global iterators.

Creation and deletion of elements happens locally on each node. The resulting changes in local memory and iteration space are not reflected in the respective global spaces until the graph's *barrier synchronization* is called. This *epoch-based* synchronization allows for a trade-off between dynamics and performance.

Listing 4.2 shows an empty graph being constructed and two vertices and an edge between them being added to it.

Listing 4.2: Dynamic creation of graph elements

```
DynamicGraph g(4, 3); // creates a graph with 0 elements but capacity for
                      // 4 vertices and 12 edges

auto v1 = g.add_vertex(); // returns index of created vertex
auto v2 = g.add_vertex();

auto e1 = g.add_edge(v1, v2);

g.commit();             // changes are published
```

4.2.3. AttributedGraph

The `AttributedGraph` concept extends the `Graph` concept with attributed elements. The user can specify static structs (a struct in a contiguous memory region without pointers and references) for vertices and edges respectively. A constructor allows concrete instances of these structs to be added to each element. Vertex and edge types contain copies of the struct instances and make them publicly available. Attributes can be replaced.

Listing 4.3 shows a graph being constructed with two vertices and an edge between them. The vertices and edges store an attribute *id*.

Listing 4.3: Graph construction with attributed elements

```
typedef  std::tuple<
    std::pair<int, v_prop>,
    std::pair<int, v_prop>,
    e_prop
> tuple_t;

struct v_prop {
  int id
};

struct e_prop {
  int id
};

v_prop vp1 { 1 };
v_prop vp1 { 2 };
v_prop ep1 { 1 };
tuple_t e1 = std::make_tuple(
  std::make_pair(1, vp1),
  std::make_pair(2, vp2),
  ep1
);

std::list<tuple_t> edge_list = { e1 };
AttributedGraph<v_prop, e_prop> g(2, 1, edge_list.begin(), edge_list.end());
```

A `DynamicGraph` that also satisfies the requirements of `AttributedGraph` can add attributes on-the-fly, reducing the complexity of the syntax. An example is shown in Listing 4.4.

Listing 4.4: Dynamic creation of attributed graph elements

```
struct v_prop {
  int id
};

struct e_prop {
  int id
};

AttributedDynamicGraph<v_prop, e_prop> g(2, 1);
v_prop vp1 { 1 };
auto v1 = g.add_vertex();
g.set_attribute(v1, vp1);
v_prop vp2 { 2 };
auto v2 = g.add_vertex();
g.set_attribute(v1, vp1);
e_prop ep1 { 1 };
auto e1 = g.add_edge(v1, v2);
g.set_attribute(e1, ep1);
g.commit();
```

4.2.4. DuplexGraph

The `DuplexGraph` concept extends the `Graph` concept with iterators for inbound edges. It is a requirement for *undirected* and *bidirectional graphs* and optional for *directed graphs*.

Listing 4.5 shows a graph with two edges pointing from *vertex 1*. These edges are then iterated with the inbound edge iterator.

Listing 4.5: Inbound edge iteration

```
std::list<std::pair<int, int>> edge_list = { {1, 2}, {1, 3}, {4, 1} };
Graph g(4, 3, edge_list.begin(), edge_list.end());

auto v_it = g.vertices().lbegin();

for(auto e_it = g[v_it].in_edges().begin(); e_it != g[v_it].in_edges().end();
    ++e_it) {
  Graph::edge_type e = *e_it; // returns edge (4, 1)
}
```

4.2.5. CombinedEdgeGraph

The `CombinedGraph` concept extends the `Graph` concept with iterators for a combination of inbound and outbound edges. It is advised that an implementation also satisfies `DuplexGraph`, but this is not a requirement. The specific iteration order for the combination of inbound and outbound edges is not specified in this concept. It is left to the implementation.

Listing 4.6: Combined edge iteration

```
std::list<std::pair<int, int>> edge_list = { {1, 2}, {1, 3}, {4, 1} };
Graph g(4, 3, edge_list.begin(), edge_list.end());

auto v_it = g.vertices().lbegin();

for(auto e_it = g[v_it].edges().begin(); e_it != g[v_it].edges().begin(); ++e_it) {
  Graph::edge_type e = *e_it; // returns edges (1, 2), (1, 3) and (4, 1)
}
```

4.3. Memory spaces

A `Graph` contains two distinct memory spaces for vertices and outbound edges. A `DuplexGraph` additionally contains a memory space for inbound edges. Each memory space comprises its own iteration space (see section 4.4). Memory spaces are maintained on a per-node basis and new memory is allocated locally with an STL-compatible container. The concrete container implementation can be specified by the user but it must allocate *contiguous memory regions* (note: in C++17 [ISO17], this means the container must satisfy the requirements of the `ContiguousContainer` concept. Because this work is based on C++11, no formal requirement can be formulated). By default, `std::vector` should be used as allocator.

An object maintaining a memory space on a certain node also holds information about the memory spaces on all other nodes. Changes in local memory space are hidden from other nodes until a `commit` operation initiates a communication step that exchanges information about the occupied memory.

Remote data can then be accessed with this information using *Remote Direct Memory Access (RDMA)* features of the used communication layer (e.g. MPI-3).

The concrete memory model is up to the implementation. The publicly visible memory space however has to adhere to the following definitions:

Definition 4.1 *Local vertex memory space is logically contiguous. New elements are either added at the end of the memory space or, if available, at the position of elements marked for deletion.*

Definition 4.2 *Local edge memory space consists of multiple logically contiguous edge-list memory spaces that are connected to each other in arbitrary order. New elements are either added at the end of their respective edge-list memory spaces or, if available, at the position of elements marked for deletion in their respective edge-list memory spaces.*

Definition 4.3 *Global memory spaces are a union of local memory spaces ordered by their node ID ascendingly.*

Memory spaces do not have to be completely contiguous in memory. They can consist of several contiguous memory regions. For each of these memory regions, the beginning memory address as well as the length has to be stored. An abstraction can then use this information to present a contiguous memory region to the public.

Figure 4.2 shows exemplary logical memory spaces for vertices and edges respectively. The memory spaces appear contiguous to the outside but parts of them might reside in different memory locations depending on the implementation.

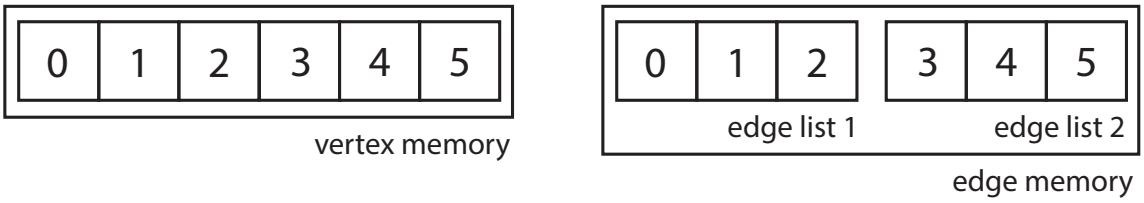

Figure 4.2.: Vertex and edge memory space abstractions

4.4. Iteration spaces

Elements in all graph concepts of this work are identified by iterators. Two different kinds of iterators exist:

Global iterators identify elements in the whole graph. They are needed to iterate over portions of the graph that do not reside locally on a node and to access these elements. Dereferencing a global iterator yields a global reference which in turn can be dereferenced to access the actual data. A *global reference* is therefore responsible for *remote memory access*.

Local iterators identify elements residing locally on a node. A simple pointer to a contiguous memory region can already suffice for this task. However, dynamic containers usually require multiple memory regions to be created during their lifetime. Because local iterators must be able to iterate over the complete local portion of the elements, a more sophisticated abstraction will be needed in case the local memory space consists of more than one contiguous region.

Global and local iterators have to strictly follow the `STL Iterator` concepts in order to be usable by STL algorithms. Additionally, they have to implement mechanisms geared towards PGAS concepts. For example, a global iterator should provide a possibility for finding out if the element it points to resides locally or on a remote node. This work is based upon `DASH Iterator` concepts [FFK16a] that are suitable for any implementation of the presented graph concepts.

Global and local iteration spaces directly map to global and local memory spaces explained in section 4.3. Because of this, iterators are invalidated when changes to the memory spaces

are published.

4.5. Index spaces

The concepts of this work do not comprise index spaces. They can however be implemented on top of them and exist alongside the iteration spaces that are primarily used to identify elements. This section gives a brief overview of possibilities in this regard:

Simple integer index spaces like they are used in STL containers (e.g. `std::vector`) or static DASH containers (e.g. `dash::Array`) are problematic: Because element creation in dynamic containers requires new indices to be generated during runtime, all nodes would have to find consensus so that index numbers are continuous and unique. This would require one of the following mechanisms:

1. Increment a global *maximum index number* on each element creation.
2. Create a local index on each element creation and negotiate global indices for multiple elements during `commit` operations.

Variant 1 requires communication between nodes every time an element is created. This additional overhead is intolerable for high performance applications and therefore not suitable for this concept.

Variant 2 only adds communication during the `commit` operation. If new elements have been created, the `commit` operation executes all-to-all communication. An additional mechanism that negotiates index numbers could therefore be performed in the same communication steps adding only a slight overhead. Unfortunately, using the created elements locally before a `commit` requires the user to handle the elements with a temporary local index that is later substituted by a global index. This leads to additional programming complexity.

A split index can be used to circumvent this problem:

3. Use indices with multiple components (global and local).

Variant 3 allows for indices to be created without global consensus: Indices contain the node ID and the local index on that node. This however comes with another problem: If mechanisms for element migration are employed, the global (node) part of the indices has to be either changed or a global directory structure has to be built in order to redirect requests of a certain index to the correct node.

5. Reference implementation

This section explains a concrete reference implementation of the concepts in chapter 4. The implementation is part of the DASH C++ Template Library and thus written in C++11. It is based on basic C++ concepts illustrated in section 2.2. The reference implementation will be referred to as `dash::Graph` in this chapter.

5.1. Overview

`dash::Graph` is part of the dynamic data containers of the DASH Library. As such, it is interacting with existing components of the library. Figure 5.1 depicts the components and their main interactions.

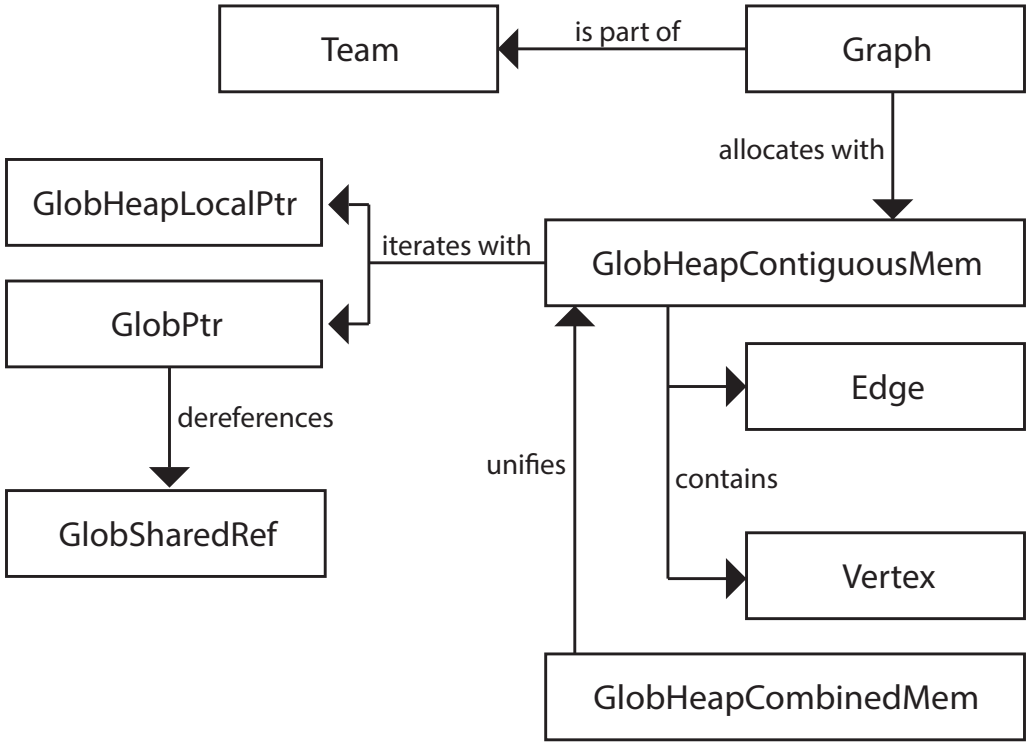

Figure 5.1.: dash::Graph component overview

The `Graph` gets initialized with a reference to an existing `Team`. By default, this is `Team::All()` which includes all units (see section 2.5) DASH has been initialized with. The `Graph` creates three instances of `GlobHeapContiguousMem` for vertices, inbound edges

and outbound edges. These instances are used to globally allocate memory for the respective elements. Since the `Graph` also allows for the iteration of all (inbound and outbound) edges, `GlobHeapCombinedMem` unifies the memory spaces of the two `GlobHeapContiguousMem` instances. Both `GlobHeapContiguousMem` and `GlobHeapCombinedMem` use a specialized template version of `GlobPtr` to iterate over the memory space. Each `GlobPtr` object can then be dereferenced to a `GlobSharedRef` object which enables remote memory access to the referenced element. For locally residing elements, `GlobPtr` dereferences to a local reference pointing to a concrete memory location.

5.1.1. Data structure

The reference implementation's underlying data structure is an *adjacency list*. It consists of a logically contiguous array of vertices and two logically contiguous arrays of edges (inbound and outbound edges respectively). The arrays are connected with each other by a separate data structure that stores information about starting point and length of an edge list belonging to a particular vertex. Figure 5.2 depicts the mapping of vertices to their corresponding edge lists.

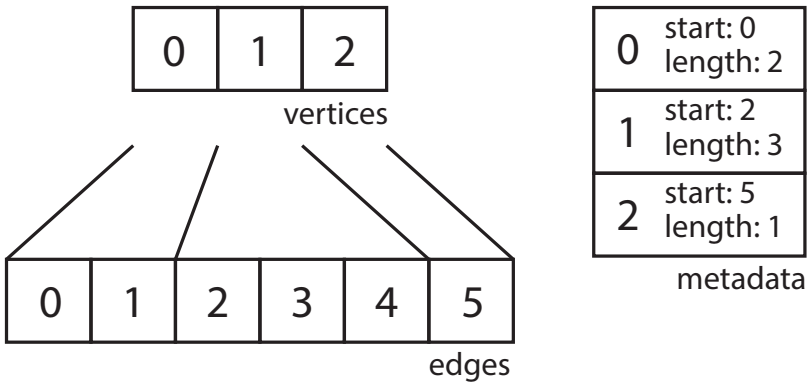

Figure 5.2.: Data structure mapping

Vertices and edges are modeled as individual classes: `Vertex` and `Edge`.Because the meta-information that links the vertex array to the edge arrays is also used by iterators it is not stored inside of the `Vertex` objects but in an independent data structure. Vertices store attribute information only, while edges additionally store identifiers of their source and target vertices. While `dash::Graph` exposes only iterators to the public, source and target vertices are identified by indices internally because iterators can be invalidated during `commit` operations and because the iterators in use have a higher memory consumption.

Attribute information can only consist of variables of static size. These variables are filled with default values upon initialization as per *default initialization* of the C++ standard.

5.1.2. Pointers and references

To allow `GlobPtr` and `GlobSharedRef` act like real pointers and references respectively, operators like the increment and dereference operators are overloaded. This results in usage analogous to native pointers and references:

Listing 5.1: Operator overloading in GlobPtr and GlobSharedRef

```
typedef GlobHeapContigousMem<std::vector<int>> g_mem_type;
GlobPtr<int, g_mem_type> ptr(mem, 0); // position 0 in index space
++ptr;                                // go to position 1 in index space
auto ref = *ptr;                      // dereference ptr to GlobSharedRef object
int val = ref;                        // convert reference to value
```

In this case, `mem` is an instance of `GlobHeapContiguousMem` holding at least two globally available elements.

5.1.3. Graph types

`dash::Graph` currently supports *directed* and *undirected* graph types. A bidirectional graph type is not implemented but is consistent with the graph concept and can therefore be integrated. As described in subsection 4.2.4, directed graphs can be instantiated in two different variants depending on the need for inbound edge iteration. This is due to the fact that this iteration method is not necessary for some algorithms and requires additional communication that lowers the overall performance of the graph, even if not used at all.

For undirected graphs, edges are replicated to the edge lists of both participating vertices. For directed graphs, the same mechanism is used when inbound edge iteration is needed. Edges in directed graphs without inbound edge iteration are not replicated in any way.

5.2. Memory management

`dash::Graph`'s memory space is handled by `GlobHeapContiguousMem`. This class follows the concept described in section 4.3 but adds an additional feature: Fully contiguous global memory regions. While the memory concept only demands single edge lists to be contiguous, `GlobHeapContiguousMem` allocates a contiguous memory space for all globally visible vertices and edges for better locality exploitation. However, this comes at the expense of additional memory re-allocations in each epoch.

5.2.1. Contiguous memory

Figure 5.3 illustrates the basic scheme of the contiguous memory allocation. *Region 1* is a publicly available contiguous memory region. The memory location of *region 1* is known by other units and thus cannot be changed outside of a commit operation. Because of this, *region 2* is allocated at another memory location that might not be contiguous to *region 1*. *Region 2* contains elements that have been added in the current epoch - they are available only locally and cannot be seen by other units.

The `commit` operation starts a new epoch by packing the elements of the two memory regions into another contiguous memory region and notifying other units about the changed location and size of the region. Traversing the elements in this region is now as simple as incrementing a pointer. Local iteration over the elements however requires a hop between the publicly available region and the local-only region because they are not contiguous. `GlobHeapLocalPtr` can iterate over an arbitrary number of buckets with contiguous memory (see section 5.3). In this case, always two buckets (the two mentioned memory regions) are used.

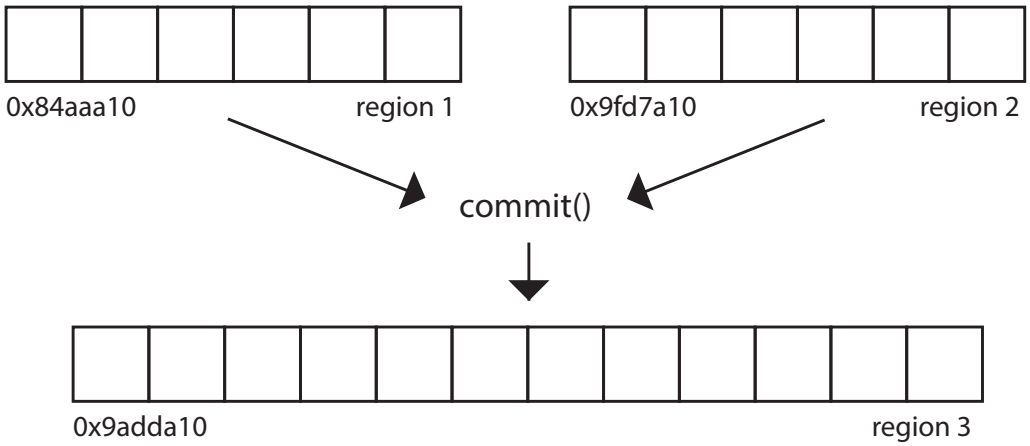

Figure 5.3.: Contiguous memory allocation

5.2.2. Edge list memory

Each edge list has to be maintained with individual second regions for element insertions: The mechanism in subsection 5.2.1 is used for each edge list individually and the `commit` operation packs all edge lists into one contiguous memory region.

5.2.3. Element deletion

Element deletion is not implemented at the time of this writing. This section however describes how a deletion mechanism can be implemented.

Deleting an element in contiguous memory regions requires its memory location to get invalidated instead of removed to avoid shifting of elements and offset invalidation. If many delete operations occur, the memory space will get scattered. For this reason, it is necessary to take measures that reduce the scattering of the memory to a minimum. A *free list* is a simple way to achieve this: Deleting an element results in its memory location being added to the free list. If another element is added, a memory location from the back of the free list is used to store the element. Only if the free list is empty, new memory is allocated.

Because invalidated elements are not part of the memory space anymore, iterators must have access to the free list so that they can skip the respective elements.

5.3. Iteration

The four different iteration spaces of the graph concept (see section 4.4) are handled by the same iterator classes. Since the memory space of `dash::Graph` is epoch-based, iteration space of local iterators can be different to the local part of the iteration space of global iterators.

5.3.1. Local iteration

Local iteration in dynamic containers of DASH is handled by `GlobHeapLocalPtr` that can iterate over multiple non-contiguous memory buckets. `GlobHeapContiguousMem` holds a list of objects containing bucket meta-data including size and a local native pointer to its beginning memory location. The bucket list is passed to a `GlobHeapLocalPtr` along with the position the pointer currently holds in the index space of the buckets. The buckets are equal to the allocated memory regions of `GlobHeapContiguousMem` as described in section 5.2.

Iteration is done by `increment/decrement` operations which result in `GlobHeapLocalPtr` calculating the bucket number the current position belongs to. A `dereference` operation then accesses the local pointer of the bucket object.

5.3.2. Global iteration

For global iteration, a template specialization of `GlobPtr` for `GlobHeapContiguousMem` is used. It is similar to the template specialization of `GlobPtr` for `GlobHeapMem`, which is used for other dynamic DASH containers like `dash::List`.

While the bucket meta-data used by `GlobHeapLocalPtr` contains only information about local buckets, `GlobPtr` requires information about all buckets on all units. Therefore, `GlobHeapContiguousMem` distributes bucket information between all units during a `commit` operation. This information includes bucket sizes as well as DART global pointers to the memory locations of the buckets in global address space.

With the bucket size information and the DART abstraction of global pointers, `GlobPtr` can iterate over global memory space the same way as `GlobHeapLocalPtr` iterates over local memory space. The iteration order is given by the canonical order of units.

5.3.3. Edge iteration

Inbound and outbound edges are handled by separate `GlobHeapContiguousMem` objects. Iteration of either is therefore handled as described in subsection 5.3.1 and subsection 5.3.2. The `CombinedEdgeGraph` concept however requires iteration over all existing edges. For this reason, `GlobHeapCombinedMem` is used to unify the iteration spaces of multiple `GlobHeapContiguousMem` objects.

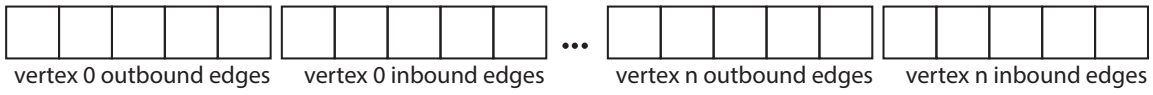

Figure 5.4.: Combined edge iteration space

Figure 5.4 illustrates the order in which the combination of inbound and outbound edges are iterated over.

5.4. Data access

Remote memory data access is strictly handled via `GlobSharedRef` objects. These objects issue DART `get/put` operations to access data on remote machines. If referenced data resides locally, `GlobSharedRef` directly accesses it in the memory.

Because edges are replicated for some graph types (see subsection 5.1.3), writing data to an element with a `GlobSharedRef` object would introduce consistency problems as it is restricted to referencing single memory locations. At the time of this writing, the reference implementation does not support a mechanism to cover this problem. It can however be achieved by another kind of *global reference* that additionally stores locations of replicated edges.

5.5. Additional methods

The following methods have been added to the reference implementation in order to facilitate faster attribute access:

5.5.1. Requirements

- X, the graph type
- x, a value of type X
- li_v, a value of type `X::vertex_size_type`
- li_e, a value of type `X::edge_size_type`

5.5.2. Types

Name	**Type**	**Notes**
vertex_size_type	unsigned integer	

5.5.3. Methods and operators

Expression	**Return**	**Semantics**
$x.vertices().attributes(li_v)$	vertex_attributes_type	reference to the attributes of the vertex identified by li_v
$x.vertices().set_attributes(li_v, va)$		sets the attributes of the vertex identified by li_v
$x.out\_edges().attributes(li_e)$	vertex_attributes_type	reference to the attributes of the vertex identified by li_e
$x.out\_edges().set_attributes(li_e, ea)$		sets the attributes of the vertex identified by li_e
$x.in\_edges().attributes(li_e)$	vertex_attributes_type	reference to the attributes of the vertex identified by li_e

Expression	**Return**	**Semantics**
$xin\_edges().set_attributes(li_e, ea)$		sets the attributes of the vertex identified by li_e

The mentioned methods are not part of the concept.

6. Case studies

This chapter illustrates use cases for the previously described graph concepts using the reference implementation explained in chapter 5. These use cases are then evaluated against existing implementations in chapter 7.

The use cases consist of two important basic graph algorithms: *Connected Components* and *Minimum Spanning Tree*. Both algorithms have been examined in PGAS implementations using the UPC programming language [C+05] and their implementations in this work will roughly follow the specifications explained in [CAS10] which consists of shared-memory algorithms ported to UPC.

6.1. Optimizations

While shared-memory algorithms can be directly ported to distributed machines in PGAS, performance might suffer from irregular data accesses across machines as the latency of the network is much higher than the latency of memory access in shared-memory machines. Cong et al. therefore propose performance optimizations including *communication coalescing* and *cache performance optimization*. While the results show that these optimizations are significant (up to 5 times faster for the designated graph size), this case study only employs the *communication coalescing* optimization because it is the most beneficial for scalability and an implementation of all optimization techniques is beyond the scope of this thesis.

6.1.1. Communication coalescing

Because shared-memory algorithms typically do not employ domain decomposition the way needed for distributed systems, processors issue many access requests to data belonging to other processors. On a shared-memory system, this is not problematic as the access latency is the same or, in case of NUMA domains, only slightly higher. On distributed systems however, these accesses result in remote memory accesses for which the latency is determined by the network and therefore orders of magnitude higher. Thus, directly porting shared-memory algorithms in the PGAS space can result in low performance.

To overcome this problem, data accesses have to happen in batches rather than individually (similar to bulk-synchronous messaging in BSP models). The *communication coalescing* mechanism of [CAS10] proposes to collect all data reads and/or data writes of one synchronization step and distribute them to the respective processors in one message. In algorithm 1, a less generalized version of the algorithm that does not account for local cache-performance optimization is shown.

A *communication coalescing* algorithm for data writes works analogously.

Data: array *indices* of indices to read
Result: array *output*
sort *indices* by processor ID and save permutations;
alltoall(*indices*, *indices_recv*);
for $i \leftarrow 0$ ***to*** *indices_recv.size*() **do**
  data[i] $\leftarrow$ data attribute of the element *indices_recv*[i];
end
alltoall(*data*, *data_recv*);
restore original order with saved permutations on *data_recv*;
return *data_recv*;

Algorithm 1: Communication coalescing algorithm for data reads

6.2. Graph algorithms

This section presents a description of the algorithms used in the case studies as well as a high level description of their implementation.

6.2.1. Connected Components

Connected Components is an elementary graph algorithm based on depth-first-search [CLRS09]. It aims to find components in a graph in which every vertex is connected to at least one other vertex. The algorithm finds one or more sub-graphs with no edges between them. Figure 6.1 depicts a graph comprising two connected components.

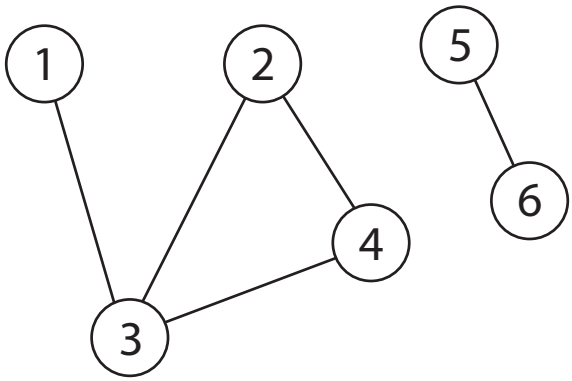

Figure 6.1.: Graph with two connected components

The classical variant of the algorithm performs a depth-first search on a graph G and another depth-first search on the transposed version of the graph G^T. Because locality can hardly be exploited in depth-first search and because two runs per iteration are necessary to compute this algorithm, variations of the algorithm exist that account for better performance [SHS03] [DSB99].

The algorithm used in this case study is based on the shared-memory version found in [CAS10]. Contrary to the classical variant of the algorithm, no depth-first search is performed. All edges of the graph are examined in an arbitrary order leading to much better

locality exploitation: Each processor can examine the edges in the order they are placed in memory.

The algorithm requires vertex attributes for the parent component a vertex belongs to. This parent component attribute is initially set to the vertex itself. The algorithm then proceeds with iteration over the following two steps until no more component changes can be found:

1. For each edge, set the component of the target vertex' current parent to the source vertex' parent if source vertex parent $<$ target vertex parent
2. Perform *pointer-jumping* over all component attributes until the parent components of all vertices point to themselves.

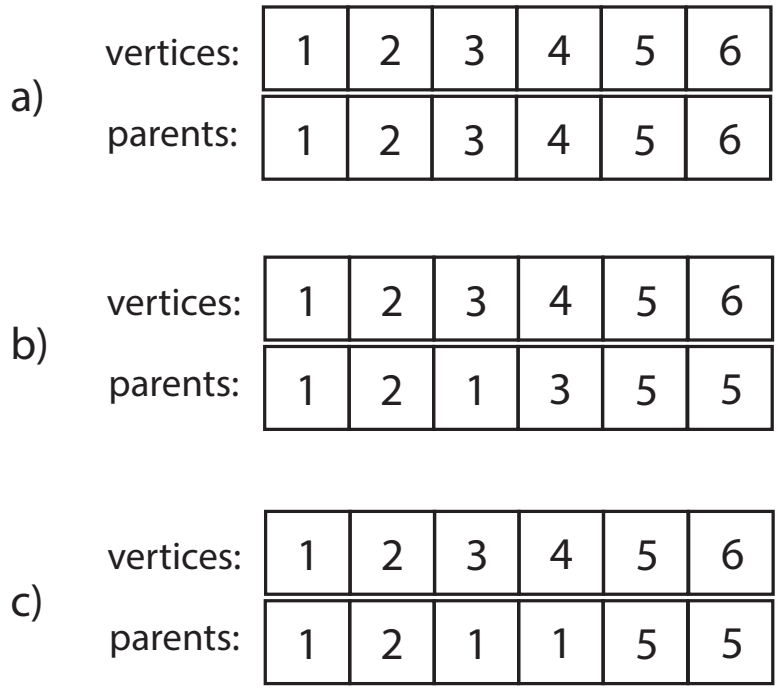

Figure 6.2.: Steps of Connected Components with initial setup (a), first step (b) and second step (c)

Figure 6.2 a) shows the initial setup for the Connected Components algorithm according to the example graph of Figure 6.1: Each vertex has a parent component attribute assigned to it that is initially set to the vertex itself. For each edge in the graph, the component of the target vertex component is set to the component of the source vertex if the target component is greater than the source component in Figure 6.2 b). Pointer-jumping (`parent[vertex]` $\leftarrow$ `parent[parent[vertex]]`) is then performed in Figure 6.2 c) so that each vertex either points to itself or to another vertex that points to itself. It is worth to note that, because edge-lists can be traversed in arbitrary order, the component of *vertex 4* in the first algorithm step can also be *2*. In this case, the pointer-jumping step will not perform any changes. After a second iteration, the component of *vertex 2* will be set to *1* resulting in termination

of the algorithm.

The described algorithm is optimized using communication coalescing explained in subsection 6.1.1. The resulting algorithm can be seen in algorithm 2. The components of the target vertices of all edges local to each processor are gathered in a bulk communication step. This data is used to set the parent component of each target's parent to the source's parent component in another bulk communication step. Pointer jumping is then performed in a synchronized way in order to allow for communication coalescing: The component of each vertex is updated and the pointers are not immediately followed but rather updated in the next iteration. The algorithm stops if there are no changes in the parent components of all vertices in an iteration.

6.2.2. Minimum Spanning Tree

A *Minimum Spanning Tree (MST)* is a tree structure spanning a connected component along paths of minimum edge weight [CLRS09]. The tree spans all vertices in the component and the sum of edge weights of each path is guaranteed to be minimal. The MST therefore is a subset of the graph. Figure 6.3 shows a graph and its corresponding minimum spanning tree.

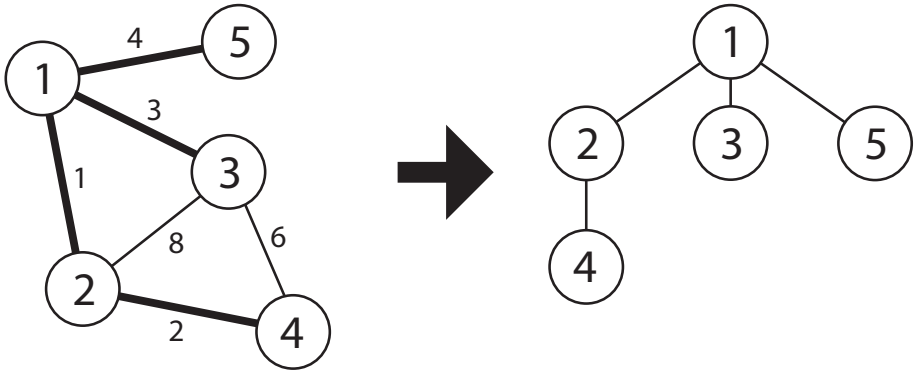

Figure 6.3.: Minimum spanning tree

A graph consisting of multiple connected components contains a *minimum spanning forest* of trees for each component. The algorithm presented in this section computes minimum spanning trees for all components of a graph and therefore results in a minimum spanning forest without the need of classifying the connected components beforehand.

Classical implementations of the algorithm are *Kruskal's algorithm* and *Prim's algorithm* [GH85]. Both algorithms are greedy but have been proven to create trees with minimum edge weight.

This case study however is based on a shared-memory version of *Boruvka's algorithm* [CB06] due to its increased performance in parallel environments [CAS10]. It comprises iteration of three steps:

1. For each vertex, find the outgoing edge with the minimum weight.

2. Find connected components for the induced graph of the found edges.

3. Compact each connected component into a supervertex.

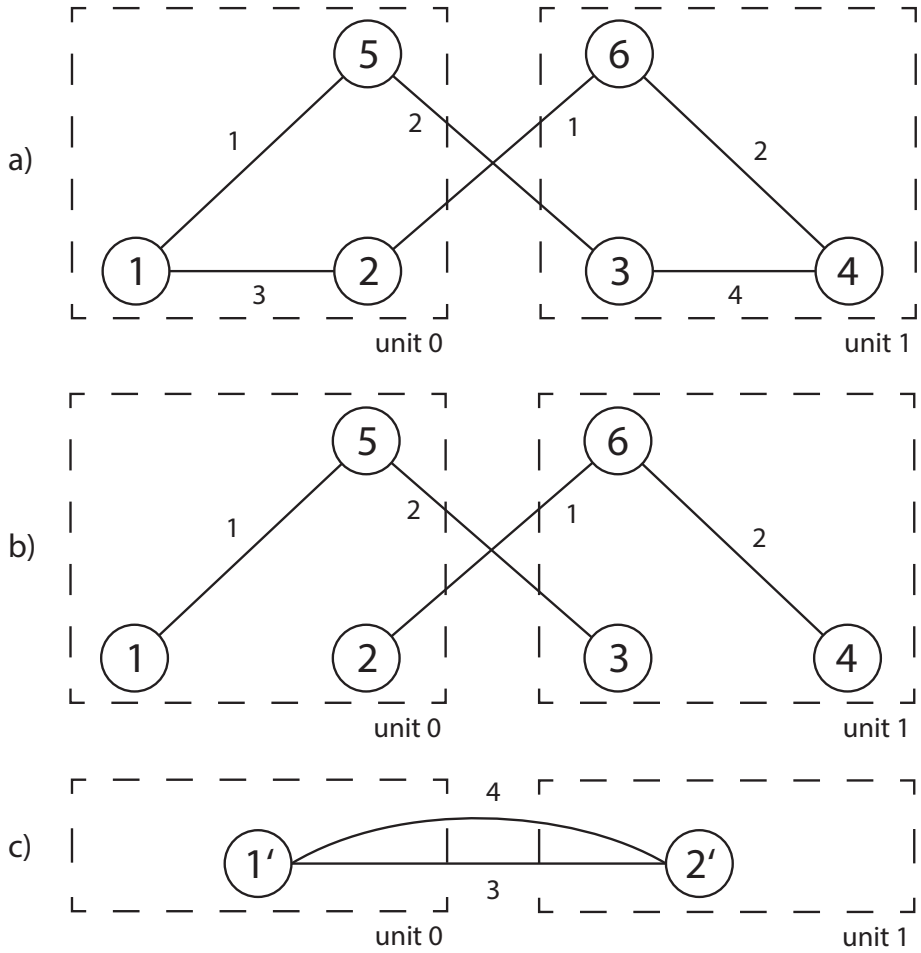

Figure 6.4.: Distributed Minimum Spanning Tree example showing the graph (a), the first iteration (b) and the second iteration (c) [CB06]

Each new iteration is computed on a graph composed of *supervertices* which contain all edges of their respective vertices. Because compacting vertices is costly, vertices are simply labeled with their supervertex index. To find the outgoing minimum weight edge of a supervertex, multiple vertices have to be iterated over.

Since these vertices can reside on different processors, there has to be consensus between multiple processors about the minimum weight edge. This is achieved by computing the minimum weight edge on each processor individually and then exchanging the computed edges along with their weight between all participating processors. Each processor can then determine the edge with the minimum weight and the processor it belongs to updates the edge accordingly.

Figure 6.4 a) depicts an exemplary graph that is distributed among two processors. Figure 6.4 b) shows the first iteration of the algorithm: For each vertex, the edges with minimum weight are chosen. The resulting connected components (1, 3, 5) and (2, 6, 4) are then compacted to the new supervertices 1' and 2' which are shown in Figure 6.4 c). Because processor 1 and processor 2 see different edges between these supervertices, both processors choose a different edge as a candidate for the minimum weight edge between the them. To reach consensus, both processors exchange these edges along with their weight and the processor owning the edge with minimum weight updates the minimum spanning tree.

The MST algorithm of this case study can be seen in algorithm 3. It is similar to algorithm 2 but uses adjacency iteration to determine the minimum weight edge for each (super)vertex.

```
Data: reference G to the graph
while 1 do
    gr ← 0;
    i ← 0;
    for all local out-edges e in G do
        indices[i] ← global iterator position of e.target();
        ++i;
    end
    data ← parent components of indices (communication coalescing step);
    i ← 0;
    for all local out-edges e in G do
        source_parent ← parent component of e.source();
        target_parent ← parent component of e.target();
        if source_parent < target_parent then
            set_data[i] ← (global iterator position of target_parent, source_parent);
            gr ← 1;
        end
        ++i;
    end
    update components with set_data (communication coalescing step);
    allreduce(gr, ADD);
    if gr = 0 then break;
    while 1 do
        pj ← 0;
        i ← 0;
        for all local vertices v in G do
            vertex_parent ← parent component of v;
            indices[i] ← global iterator position of vertex_parent;
            ++i;
        end
        data ← parent components of indices (communication coalescing step);
        i ← 0;
        for all local vertices v in G do
            if component of v ≠ data[i] then
                set parent component of v to data[i];
                pj ← 1;
                ++i;
            end
        end
        allreduce(pj, ADD);
        if pj = 0 then break;
    end
end
```

Algorithm 2: Connected Components with communication coalescing

```
Data: reference G to the graph
while 1 do
    gr ← 0;
    i ← 0;
    for all local vertices v in G do
        for all local out-edges e of v do
            indices[i] ← global iterator position of e.target();
            ++i;
        end
    end
    data ← parent components of indices (communication coalescing step);
    i ← 0;
    for all local vertices v in G do
        for all local out-edges e of v do
            source_parent ← parent component of e.source();
            target_parent ← parent component of e.target();
            if source_parent ≠ target_parent ∧ e = edge with minimum weight then
                set_data[global iterator position of source_parent] ← (target_parent,
                 e);
                set_data[global iterator position of target_parent] ← (source_parent,
                 e);
                gr ← 1;
            end
            ++i;
        end
    end
    update components with set_data (communication coalescing step): if multiple
     edges for the same (supervertex, supervertex) pair exist, update the edge with
     the minimum weight;
    allreduce(gr, ADD);
    if gr = 0 then break;
    while 1 do
        pj ← 0;
        i ← 0;
        for all local vertices v in G do
            vertex_parent ← parent component of v;
            indices[i] ← global iterator position of vertex_parent;
            ++i;
        end
        data ← parent components of indices (communication coalescing step);
        i ← 0;
        for all local vertices v in G do
            if component of v ≠ data[i] then
                set parent component of v to data[i];
                pj ← 1;
                ++i;
            end
        end
        allreduce(pj, ADD);
        if pj = 0 then break;
    end
end
```

Algorithm 3: Minimum Spanning Tree with communication coalescing

7. Evaluation

To examine the performance and functionality of the graph concepts, the reference implementation is evaluated on a live HPC system with various kernels as described in detail in this chapter.

The testing environment consists of the *SuperMUC Phase 2*[2] system at the *Leibniz Supercomputing Centre* in Munich. It comprises *six islands* composed of *512 nodes* each. The nodes are interconnected with an *Infiniband FDR14* network and contain two *Intel Xeon E5-2697 v3* processors (with 28 cores in total) and 64GB of RAM.

All tests are performed on a single island with a varying amount of nodes. Because the load on the network inside the islands might vary depending on other jobs currently running on the system, each test is performed 5 times and the median values are presented.

7.1. Micro-benchmarks

This section presents a performance analysis of specific parts of the graph concepts.

Since element deletion is currently not implemented in the reference implementation, no micro-benchmarking could be performed in this regard.

7.1.1. Element creation

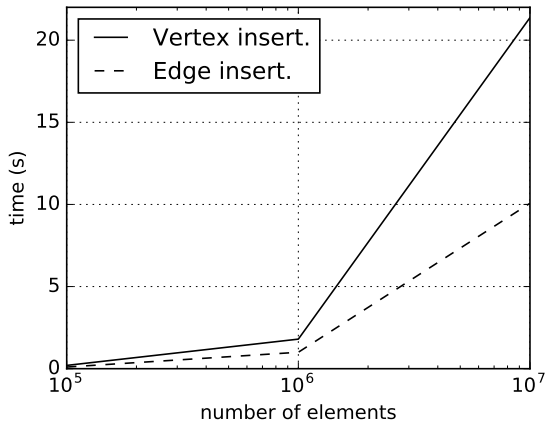

Figure 7.1.: Insertion of vertex and edge elements

Elements are always created on local machines. As no communication is performed during this micro-benchmark, the tests are performed on a single node with a varying amount of

[2] `https://www.lrz.de/services/compute/supermuc/systemdescription/`

elements.

Figure 7.1 shows the time consumed for the insertion of vertices and edges respectively. For each element type, the insertion has been performed with 100k, 1M and 10M elements. The data shows that the time consumed is not strictly linear: For vertices, a tenfold increase of created elements results in a 11.8 times increased runtime.

This happens due to the fact that the elements are stored in memory contiguously. Since the elements are created one after another in this benchmark, the container does not know the amount of elements added beforehand. The underlying data container (see `X::vertex_container_type` and `X::edge_container_type` in appendix A) thus allocates a specific amount of memory and after its capacity is reached, a re-allocation occurs.

7.1.2. Attribute access

Attribute values of elements can be accessed on local as well as remote nodes. This test aims to highlight the differences in access times for read operations with a varying amount of elements and nodes.

Figure 7.2 depicts the runtimes for remote memory access to vertices and edges respectively. The benchmark creates the desired amount of elements and then tries to access the attributes of these elements from a single processor. The data shows that, on a single node, access is almost instant while the access to data residing on remote nodes is costly: The benchmark could only be performed with a small element set because of the long runtime.

WTF? WHY IS THE RUNTIME FOR 448 VERTICES LONGER THAN FOR 896 VERTICES ON 8 NODES?

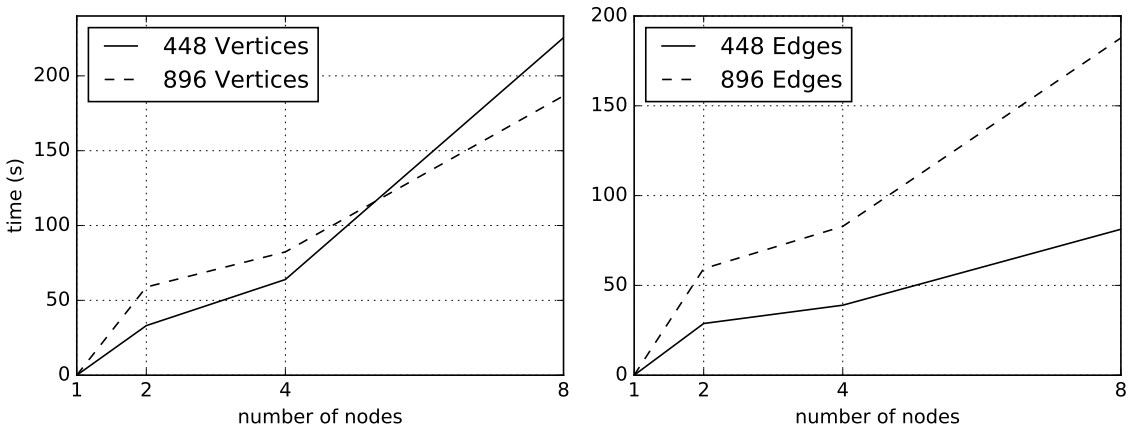

Figure 7.2.: Remote attribute access for vertices and edges

The results clearly demonstrate that irregular access to single elements is costly and should be avoided for high performance applications.

7.1.3. Local iteration

Iteration over local elements does not have any dependencies with remote nodes. For better comparability with the results of the global iteration micro-benchmark, this test is however performed on multiple nodes.

Figure 7.3 shows the results of the iteration over 1M and 10M vertices and edges respectively. To obtain better visible results, each iteration interval is *repeated 10 times*.

A tenfold increase in the amount of elements results in roughly a tenfold increase in the runtime.

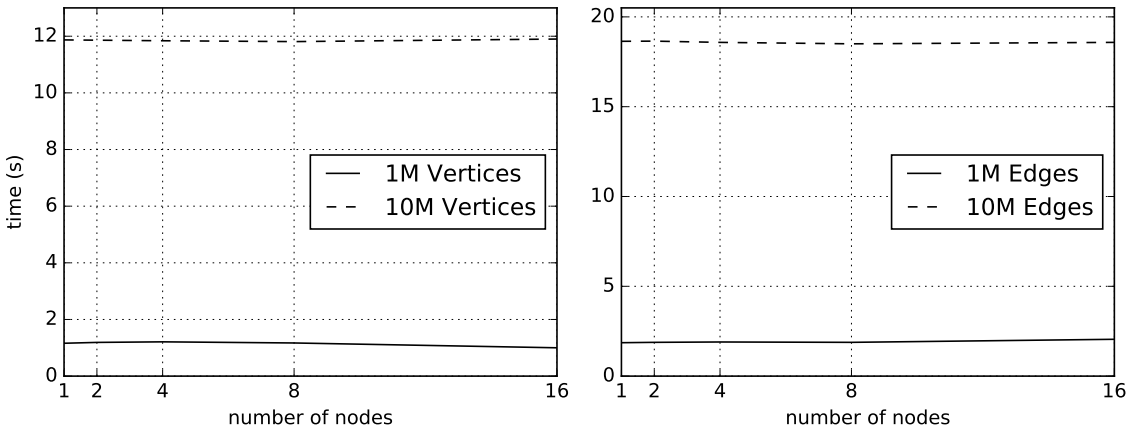

Figure 7.3.: Local iteration of vertices and edges

7.1.4. Global iteration

Iteration over elements stored on multiple nodes requires additional information that can degrade performance in comparison to local iteration. This test aims to highlight the differences between global and local iteration in terms of runtime. It is performed with a varying amount of nodes and a varying amount of elements. The iteration is performed by a single processor.

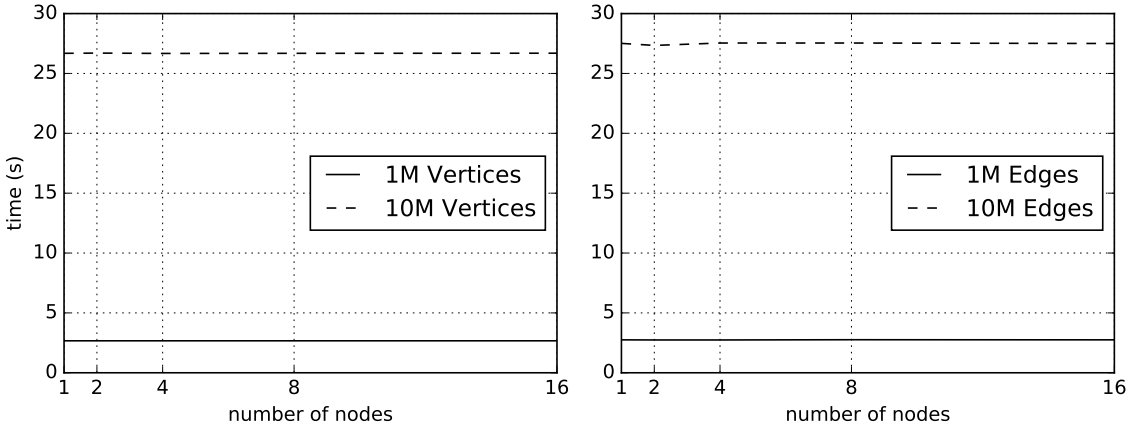

Figure 7.4.: Global iteration of vertices and edges

In Figure 7.4, results of the iterations over 1M and 10M vertices and edges can be seen. For better visible results, each iteration interval is *repeated 10 times*.

The data shows no noticeable runtime variations for different amounts of nodes/processors.

7.1.5. Memory space synchronization

The memory space synchronization is performed between all participating processors. This test examines the runtime of the synchronization process with a varying amount of elements and nodes.

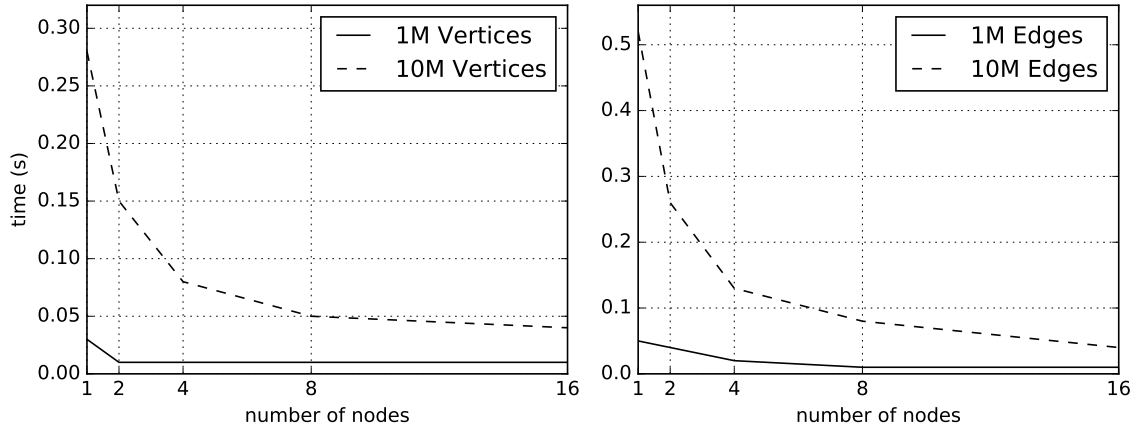

Figure 7.5.: Synchronization of memory space after creation of vertices and edges

As can be seen in Figure 7.5, the runtime of the synchronization process is bound by the amount of elements per processor and not by the amount of elements in the whole graph. With an increasing amount of processors, the 1M and 10M vertices and edges are distributed in smaller chunks reducing the synchronization time.

Interestingly, the synchronization time after the creation of edges takes noticeably longer than the synchronization time after the creation of vertices. Vertices always belong to the processor on which they are created. This is not true for edges: An edge between a vertex residing on *processor 1* and a vertex residing on *processor 2* has to be inserted in both processors' memory. Because it is only created at one location, the synchronization step takes care of the distribution which increases the runtime.

7.2. Case studies

The case studies presented in chapter 6 are evaluated against their UPC counterparts [CAS10] in this section. Because Cong et al. used a different HPC setup for their evaluation, the resulting numbers can not be considered as directly comparable.

7.2.1. Graph setup

This section describes the properties of `dash::Graph` used for the examined case studies.

Property	**Value**
Graph type	Undirected graph

Property	Value
Vertex attributes	- Parent component
Edge attributes	

Table 7.1.: Graph setup for Connected Components

Property	Value
Graph type	Undirected graph
Vertex attributes	- Parent component
Edge attributes	- Weight - Processor ID

Table 7.2.: Graph setup for Minimum Spanning Tree

7.2.2. Input data

To maintain comparability, the input graphs of the case studies discussed in this chapter are generated in roughly the same way as described in [CAS10]. The graphs are generated using a hybrid generation scheme: 50% of the edges are generated randomly, the rest of the edges is generated using an RMAT graph generator [CZF04].

Because Cong et al. do not specify the exact input variables for the RMAT generator in their paper, default values are chosen:

Variable	Value
a	0.25
b	0.25
c	0.25
d	0.25

Table 7.3.: Input variables for RMAT graph generator

7.2.3. Experimental results (Connected Components)

The experiments with the Connected Components algorithm have been conducted with two different approaches:

- **Weak scaling** - An increase in compute power (amount of processors) involves a symmetrical increase in the amount of data that is to be processed.
- **Strong scaling** - The same amount of data is processed with a varying amount of compute power.

Weak scaling (single node)

The processed graph contains 225.000 vertices and 900.000 edges per processor. This amounts to 6.3M vertices and 25.2M edges on a whole node. This is rougly equal to the amount of elements used in the strong scaling experiment on 16 nodes.

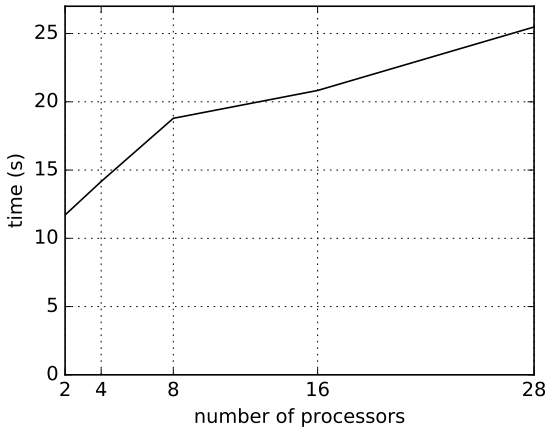

Figure 7.6.: Connected Components: Weak scaling on a single node

Figure 7.6 illustrates the time consumed by this experiment on a varying amount of processors. The data shows that there is already a noticeable algorithmic overhead that leads to an increase in runtime even if there is no communication over an interconnect that could introduce latency. The runtime roughly doubles with an increase of the amount of processors (and also of the dataset) of 14 times.

Weak scaling (multiple nodes)

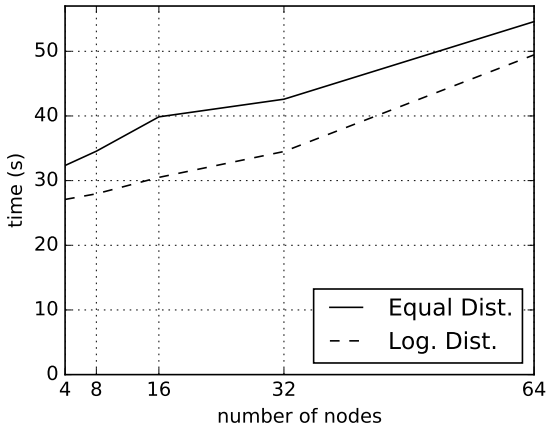

Figure 7.7.: Connected Components: Weak scaling on multiple nodes

Weak scaling on multiple nodes is continued with the same amount of elements per processor as described in the previous section.

Figure 7.7 shows the results of algorithm runs on up to 64 nodes. Interestingly, the plot strongly resembles the plot in Figure 7.6 although parts of the data are now distributed over an interconnect.

Tracing of the application has shown that parts of the algorithm converge into hotspots on the processors holding the vertices with the lowest IDs. This happens mainly in the pointer-jumping step (see subsection 6.2.1). Due to this fact, another algorithm run has been performed with a different setup: Vertices are distributed to their respective processors using a logarithmic function resulting in processors holding the vertices with the lowest ID to store less elements than the rest of the processors. The results are shown with a dashed line in Figure 7.7.

Strong scaling (multiple nodes)

Strong scaling is performed with 100M vertices and 400M edges in all setups. This is equal to the amount of elements used in the algorithm runs of [CAS10].

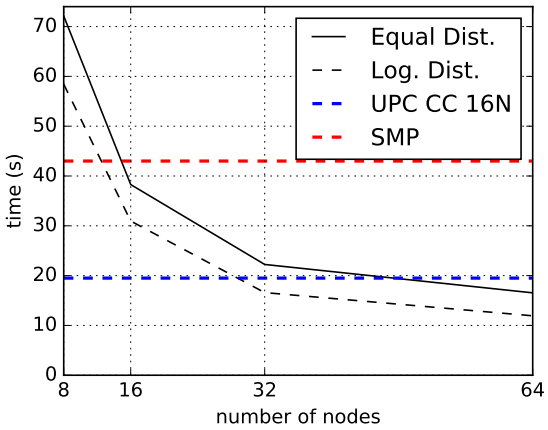

Figure 7.8.: Connected Components: Strong scaling on multiple nodes

In Figure 7.8, the results for an equal distribution of vertices as well as for distribution using a logarithmic function can be seen. Additionally the runtime of the Connected Components algorithm of Cong et al. (UPC CC 16N) is shown with a dashed blue line and the runtime of a shared-memory (SMP) version of the algorithm [CAS10] is drawn with a dashed red line.

The UPC version of Cong et al. has only been run on a cluster of 16 nodes. Therefore, no scalability comparison is available. The SMP version has been run on a single node. Unfortunately, there is also no hardware information available.

7.2.4. Experimental results (Minimum Spanning Tree)

Similar to subsection 7.2.3, experiments with the Minimum Spanning Tree algorithm have been performed with weak scaling and strong scaling on a varying amount of processors/nodes.

Weak scaling (single node)

The processed graph contains 225.000 vertices and 900.000 edges per processor. This amounts to 6.3M vertices and 25.2M edges on a whole node. This is rougly equal to the amount of elements used in the strong scaling experiment on 16 nodes.

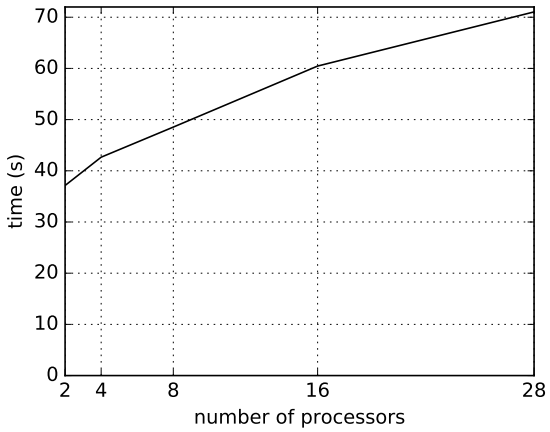

Figure 7.9.: Minimum Spanning Tree: Weak scaling on a single node

Figure 7.9 presents the runtime on a single node. A processor count/dataset increase of 14 times results in a runtime 1.9 times longer.

Weak scaling (multiple nodes)

With 225.000 vertices and 900.000 edges per processor, weak scaling of the algorithm has been performed on uop to 64 nodes.

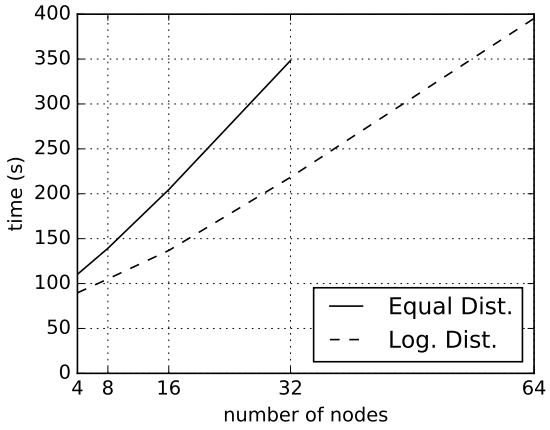

Figure 7.10.: Minimum Spanning Tree: Weak scaling on multiple nodes

The results in Figure 7.10 indicate that the Minimum Spanning Tree algorithm is more communication-intensive in comparison to the Connected Components algorithm. The

hotspot problem described in subsection 7.2.3 becomes even more apparent with this algorithm. As a matter of fact the algorithm could not be finished with an equal vertex distribution on 64 nodes.

The logarithmic vertex distribution shows slightly better scalability, but an increase of the amount of processors/nodes of 16 times still results in a 4 times longer runtime.

Strong scaling (multiple nodes)

Strong scaling is performed with 100M vertices and 400M edges in all setups. This is equal to the amount of elements used in the algorithm runs of [CAS10].

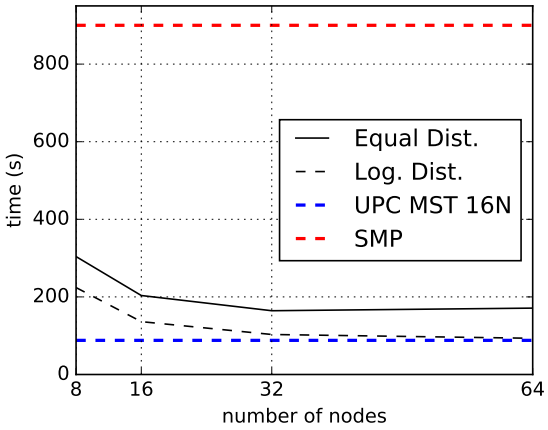

Figure 7.11.: Minimum Spanning Tree: Strong scaling on multiple nodes

In Figure 7.11, the results for an equal distribution of vertices as well as for distribution using a logarithmic function can be seen. Additionally the runtime of the Minimum Spanning Tree algorithm of Cong et al. (UPC MST 16N) is shown with a dashed blue line and the runtime of a shared-memory (SMP) version of the algorithm [CAS10] is drawn with a dashed red line.

The UPC version of Cong et al. has only been run on a cluster of 16 nodes. Therefore, no scalability comparison is available. The SMP version has been run on a single node. Unfortunately, there is also no hardware information available.

7.2.5. Result assessment

7.2.6. Programming assessment

A major aspect of Partitioned Global Address Space is the ability to port shared-memory algorithms to distributed memory with low effort. This approach however can lead to degraded performance:

The Connected Components algorithm has been directly ported to a PGAS version using the graph concepts of this work. Due to many irregular remote accesses on single elements the algorithm could not be finished in a timely manner on a graph with 100M vertices and 400M edges and is thus not suitable for high performance applications.

To achieve good performance results, communication has to be performed in batches even for PGAS applications. For this reason, a communication coalescing mechanism had to be used for the case studies to actually deliver acceptable results. This mechanism however requires a programmer to invest significantly more time into the development of such PGAS algorithms which counteracts the programmability of the concepts of this work.

Judging from the experience gathered from the case studies, the communication coalescing process can be encapsulated by another data structure in order to ease the work for programmers using containers that meet the requirements of the graph concepts.
The desired functionality can be achieved by two different data structures:

- **vertex subset**
- **edge subset**

Both data structures have to meet the following functional requirements:

Requirement	**Notes**
Element referencing	Insertion of references to single elements of the graph (local and global)
Element data creation	Creation of new data for the referenced elements
Bulk retrieval	User-induced retrieval of all referenced element data in one communication step
Bulk data mutation	User-induced mutation of the referenced elements' data using the newly created data in a bulk communication step

The proposed solutions are not part of this work and require additional effort to create sophisticated concepts that span a wide array of use cases.
Considering the uses cases, the proposed data structures could reduce the lines of code of both algorithms by around 70%.

8. Conclusion

8.1. Summary

8.2. Assessment

8.3. Outlook

Appendices

A. Graph container concepts

A.1. Graph

A `Graph` is a container providing minimal functionality for the creation and adjacency iteration of a graph data structure.

A.1.1. Requirements

- X, the graph type
- P, a pair type satisfying the STL `pair` concept
- C, a container type satisfying the STL `SequenceContainer` concept
- m, an object of type `X::mapper`
- x, a value of type X
- it_1 and it_2, `InputIterators`[2] with a valid range $[it_1, it_2]$ referring to elements of type `P<int, int>`
- it_v, a value of type `X::global_vertex_iterator`
- lit_v, a value of type `X::local_vertex_iterator`
- it_e, a value of type `X::global_edge_iterator`
- lit_e, a value of type `X::local_edge_iterator`
- *proc*, a value of type `X::processor_id_type`

A.1.2. Types

Name	**Type**	**Notes**
vertex_type	a model describing vertex information	
edge_type	a model describing edge information	
vertex_proxy_type	a proxy type for the retrieval and modification of vertex data	

[2] see section 2.2.2

Name	Type	Notes
edge_proxy_type	a proxy type encapsulating methods for the retrieval and modification of edge data	
vertex_size_type	unsigned integer	
edge_size_type	unsigned integer	
vertex_container_type	C	user-specified STL-compatible container with contiguous range
edge_container_type	C	user-specified STL-compatible container with contiguous range
global_vertex_iterator	iterator pointing to elements of type `vertex_type`	global iterator type satisfying `InputIterator`
local_vertex_iterator	iterator pointing to elements of type `vertex_type`	local iterator type satisfying `InputIterator`
global_out_edge_iterator	iterator pointing to elements of type `edge_type`	global iterator satisfying `InputIterator`
local_out_edge_iterator	iterator pointing to elements of type `edge_type`	local iterator satisfying `InputIterator`
mapper	a function object mapping vertices to processors	
processor_id_type	a type identifying a processor mapped to the container	

A.1.3. Methods and operators

Expression	Return	Semantics
$X(n_v, it_1, it_2, m)$		constructs the graph holding n_v vertices and std::distance(it_1, it_2) edges given by the iterators it_1 and it_2 of an edge list container
$x.operator[](it_v)$	vertex_proxy_type	vertex identified by it_v
$x.operator[](lit_v)$	vertex_proxy_type	vertex identified by lit_v
$x.operator[](it_e)$	edge_proxy_type	edge identified by it_e
$x.operator[](lit_e)$	edge_proxy_type	edge identified by lit_e
$x.vertices().begin()$	global_vertex_iterator	iterator to the first vertex in global iteration space

Expression	**Return**	**Semantics**
$x.vertices().end()$	global_vertex_iterator	iterator past the last vertex in global iteration space
$x.vertices().lbegin()$	local_vertex_iterator	iterator to the first vertex in local iteration space
$x.vertices().lend()$	local_vertex_iterator	iterator past the last vertex in local iteration space
$x.vertices().size()$	vertex_size_type	amount of globally visible vertices
$x.vertices().size(proc)$	vertex_size_type	amount of globally visible vertices owned by a processor *proc*
$x.vertices().lsize()$	vertex_size_type	amount of locally visible vertices on the respective processor
$x.vertices().empty()$	bool	true if there are more than 0 vertices globally visible
$x.vertices().max\_size()$	vertex_size_type	maximum amount of vertices possible
$x.out\_edges().begin()$	global_out_edge_iterator	iterator to the first outbound edge in global iteration space
$x.out\_edges().end()$	global_out_edge_iterator	iterator past the last outbound edge in global iteration space
$x.out\_edges().lbegin()$	local_out_edge_iterator	iterator to the first outbound edge in local iteration space
$x.out\_edges().lend()$	local_out_edge_iterator	iterator past the last outbound edge in local iteration space
$x[it_v].out\_edges().begin()$	global_out_edge_iterator	iterator pointing to the first element in the edge-list of the vertex pointed to by it_v
$x[it_v].out\_edges().end()$	global_out_edge_iterator	iterator pointing past the last element in the edge-list of the vertex pointed to by it_v
$x[lit_v].out\_edges().lbegin()$	local_out_edge_iterator	iterator pointing to the first element in the edge-list of the vertex pointed to by lit_v
$x[lit_v].out\_edges().lend()$	local_out_edge_iterator	iterator pointing past the last element in the edge-list of the vertex pointed to by lit_v
$x.out\_edges().size()$	vertex_size_type	amount of globally visible outedges

Expression	**Return**	**Semantics**
$x.out\_edges().size(proc)$	edge_size_type	amount of globally visible out-edges owned by a processor *proc*
$x.out\_edges().lsize()$	edge_size_type	amount of locally visible out-edges on the respective processor
$x.out\_edges().empty()$	bool	true if there are more than 0 out-edges globally visible
$x.out\_edges().max\_size()$	edge_size_type	maximum amount of out-edges possible

Conditions

Expression	**Precondition**	**Postcondition**
$X(n_v, it_1, it_2, m)$	n_v = the amount of different integer values in elements of $[it_1, it_2]$	std::distance(x.vertices(). begin(), x.vertices().end()) $== n_v$ std::distance(x.out_edges(). begin(), x.out_edges().end()) $== n_e$
$x.operator[](it_v)$		vertex identified by it_v
$x.operator[](lit_v)$		vertex identified by lit_v
$x.operator[](it_e)$		edge identified by it_e
$x.operator[](lit_e)$		edge identified by lit_e
$x.vertices().begin()$		
$x.vertices().end()$		
$x.vertices().lbegin()$		
$x.vertices().lend()$		
$x.vertices().size()$		
$x.vertices().size(proc)$		processor identified by *proc*
$x.vertices().lsize()$		
$x.vertices().empty()$		
$x.vertices().max\_size()$		
$x.out\_edges().begin()$		
$x.out\_edges().end()$		
$x.out\_edges().lbegin()$		
$x.out\_edges().lend()$		
$x[it_v].out\_edges().begin()$		

Expression	Precondition	Postcondition
$x[it_v].out\_edges().end()$		
$x[lit_v].out\_edges().lbegin()$		
$x[lit_v].out\_edges().lend()$		
$x.out\_edges().size()$		
$x.out\_edges().size(proc)$		processor identified by *proc*
$x.out\_edges().lsize()$		
$x.out\_edges().empty()$		
$x.out\_edges().max\_size()$		

A.2. DynamicGraph

A `DynamicGraph` is a `Graph` that enables dynamic addition and removal of vertices and edges.

A.2.1. Requirements

- X, the graph type
- P, a pair type satisfying the STL `pair` concept
- x, a value of type X
- n_v, a value of type `X::vertex_size_type`
- n_e, a value of type `X::edge_size_type`
- it_v, a value of type `X::global_vertex_iterator`
- lit_v, lit_{v1} and lit_{v2}, values of type `X::local_vertex_iterator`
- it_e, a value of type `X::global_out_edge_iterator`
- lit_e, a value of type `X::local_out_edge_iterator`

A.2.2. Methods and operators

Expression	Return	Semantics
$X(n_v, n_e)$		constructs the graph with reserved memory for n_v vertices and $n_v * n_e$ edges
$x.add\_vertex()$	local_vertex_iterator	adds a vertex
$x.remove\_vertex(it_v)$		removes a vertex pointed to by it_v

Expression	Return	Semantics
$x.remove\_vertex(lit_v)$		removes a vertex pointed to by lit_v
$x.add\_edge(lit_{v1}, lit_{v2})$	P<local_out_edge_iterator, bool>	adds an edge between vertices pointed to by lit_{v1} and lit_{v2} and returns whether the edge has been added
$x.add\_edge(lit_v, it_v)$	P<local_out_edge_iterator, bool>	adds an edge between vertices pointed to by lit_v and it_v and returns whether the edge has been added
$x.remove\_edge(it_e)$		removes an edge identified by it_e
$x.remove\_edge(lit_e)$		removes an edge identified by lit_e
$x.commit()$		synchronizes memory space across all processors

Conditions

Expression	Precondition	Postcondition
$X(n_v, n_e)$		memory allocated for n_v vertices and $n_v * n_e$ edges
$x.add\_vertex()$		returned iterator points to constructed vertex in global iteration space
$x.remove\_vertex(it_v)$	it_v points to a valid vertex in X	
$x.remove\_vertex(lit_v)$	lit_v points to a valid vertex in X	
$x.add\_edge(lit_{v1}, lit_{v2})$	lit_{v1} and lit_{v2} point to valid vertices in X	returned index identifies constructed edge in global index space
$x.add\_edge(lit_v, it_v)$	it_v and lit_v point to valid vertices in X	returned index identifies constructed edge in global index space
$x.remove\_edge(it_e)$	it_e points to a valid edge in X	
$x.remove\_edge(lit_e)$	lit_e points to a valid edge in X	

Expression	Precondition	Postcondition
$x.commit()$		locally added elements published in global address space, global iteratos are invalidated

A.3. AttributedGraph

An `AttributedGraph` is a `Graph` containing arbitrary attributes for vertices and edges.

A.3.1. Requirements

- X, the graph type
- VT, a vertex attribute type
- ET, an edge attribute type
- TP, a tuple type satisfying the STL `tuple` concept
- P, a pair type satisfying the STL `pair` concept
- m, an object of type `X::mapper`
- x, a value of type X
- n_v, a value of type `X::vertex_size_type`
- it_1 and it_2, `InputIterators` with a valid range $[it_1, it_2]$ referring to elements of type `TP<P<int, VT>, P<int, VT>, ET>`
- it_v, a value of type `X::global_vertex_iterator`
- lit_v, a value of type `X::local_vertex_iterator`
- it_e, a value of type `X::global_out_edge_iterator`
- lit_e, a value of type `X::local_out_edge_iterator`
- a_v, a value of type `X::vertex_attributes_type`
- a_e, a value of type `X::edge_attributes_type`

A.3.2. Types

Name	Type	Notes
vertex_attributes_type	VT	user-specified static struct
edge_attributes_type	ET	user-specified static struct

A.3.3. Methods and operators

Expression	**Return**	**Semantics**
$X(n_v, it_1, it_2, m)$		constructs the graph holding n_v vertices and std::distance(it_1, it_2) edges given by the iterators it_1 and it_2 of an edge list container
$x[it_v].attributes()$	*vertex_attributes_type*	returns the attributes of the vertex pointed to by it_v
$x[lit_v].attributes()$	*vertex_attributes_type*	returns the attributes of the vertex pointed to by lit_v
$x[it_e].attributes()$	*edge_attributes_type*	returns the attributes of the edge pointed to by it_e
$x[lit_e].attributes()$	*edge_attributes_type*	returns the attributes of the edge pointed to by lit_e
$x[it_v].set\_attributes(a_v)$		replaces the attributes of the vertex pointed to by it_v with a copy of a_v
$x[lit_v].set\_attributes(a_v)$		replaces the attributes of the vertex pointed to by lit_v with a copy of a_v
$x[it_e].set\_attributes(a_e)$		replaces the attributes of the edge pointed to by it_e with a copy of a_e
$x[lit_e].set\_attributes(a_e)$		replaces the attributes of the edge pointed to by lit_e with a copy of a_e

Conditions

Expression	**Precondition**	**Postcondition**
$X(n_v, it_1, it_2, m)$		memory allocated for n_v vertices and std::distance(it_1, it_2) edges
$x[it_v].attributes()$	it_v points to a valid vertex in X	
$x[lit_v].attributes()$	lit_v points to a valid vertex in X	
$x[it_e].attributes()$	it_e points to a valid edge in X	
$x[lit_e].attributes()$	lit_e points to a valid edge in X	
$x[it_v].set\_attributes(a_v)$	it_v points to a valid vertex in X	
$x[lit_v].set\_attributes(a_v)$	lit_v points to a valid vertex in X	

Expression	Precondition	Postcondition
$x[it_e].set\_attributes(a_e)$	it_e points to a valid edge in X	
$x[lit_e].set\_attributes(a_e)$	lit_e points to a valid edge in X	

A.4. DuplexGraph

A `DuplexGraph` is a `Graph` with iterators for inbound edges for each vertex. A container supporting undirected graphs is necessarily a `DuplexGraph`. For directed graphs, inbound edge iterators are optional.

A.4.1. Requirements

- X, the graph type
- x, a value of type X
- it_v, a value of type `X::global_vertex_iterator`
- lit_v, a value of type `X::local_vertex_iterator`
- *proc*, a value of type `X::processor_id_type`

A.4.2. Types

Name	Type	Notes
global_in_edge_iterator	iterator pointing to elements of type `edge_type`	global iterator satisfying `InputIterator`
local_in_edge_iterator	iterator pointing to elements of type `edge_type`	local iterator satisfying `InputIterator`

A.4.3. Methods and operators

Expression	Return	Semantics
$x.in\_edges().begin()$	global_in_edge_iterator	iterator to the first inbound edge in global iteration space
$x.in\_edges().end()$	global_in_edge_iterator	iterator past the last inbound edge in global iteration space
$x.in\_edges().lbegin()$	local_in_edge_iterator	iterator to the first inbound edge in local iteration space
$x.in\_edges().lend()$	local_in_edge_iterator	iterator past the last inbound edge in local iteration space

Expression	**Return**	**Semantics**
$x[it_v].in\_edges().begin()$	global_in_edge_iterator	iterator to the first inbound edge connected to the vertex pointed to by it_v
$x[it_v].in\_edges().end()$	global_in_edge_iterator	iterator past the last inbound edge connected to to the vertex pointed to by it_v
$x[lit_v].in\_edges().lbegin()$	local_in_edge_iterator	iterator to the first inbound edge connected to the vertex pointed to by lit_v
$x[lit_v].in\_edges().lend()$	local_in_edge_iterator	iterator past the last inbound edge connected to to the vertex pointed to by lit_v
$x.in\_edges().size()$	vertex_size_type	amount of globally visible in-edges
$x.in\_edges().size(proc)$	edge_size_type	amount of globally visible in-edges owned by a processor *proc*
$x.in\_edges().lsize()$	edge_size_type	amount of locally visible in-edges on the respective processor
$x.in\_edges().empty()$	bool	true if there are more than 0 in-edges globally visible
$x.in\_edges().max\_size()$	edge_size_type	maximum amount of in-edges possible

Conditions

Expression	**Precondition**	**Postcondition**
$x.in\_edges().begin()$		
$x.in\_edges().end()$		
$x.in\_edges().lbegin()$		
$x.in\_edges().lend()$		
$x[it_v].in\_edges().begin()$	it_v points to a valid vertex in X	
$x[it_v].in\_edges().end()$	it_v points to a valid vertex in X	
$x[lit_v].in\_edges().lbegin()$	lit_v points to a valid vertex in X	
$x[lit_v].in\_edges().lend()$	lit_v points to a valid vertex in X	
$x.in\_edges().size()$		

Expression	Precondition	Postcondition
$x.in\_edges().size(proc)$		processor identified by *proc*
$x.in\_edges().lsize()$		
$x.in\_edges().empty()$		
$x.in\_edges().max\_size()$		

A.5. CombinedEdgeGraph

A `CombinedEdgeGraph` is a `Graph` with additional iterators for a combination of inbound and outbound edges.

A.5.1. Requirements

- X, the graph type
- x, a value of type X
- it_v, a value of type `X::global_vertex_iterator`
- lit_v, a value of type `X::local_vertex_iterator`
- *proc*, a value of type `X::processor_id_type`

A.5.2. Types

Name	Type	Notes
global_edge_iterator	iterator pointing to elements of type `edge_type`	global iterator satisfying `InputIterator`
local_edge_iterator	iterator pointing to elements of type `edge_type`	local iterator satisfying `InputIterator`

A.5.3. Methods and operators

Expression	Return	Semantics
$x.edges().begin()$	global_edge_iterator	iterator to the first edge (inbound and outbound) in global iteration space
$x.edges().end()$	global_edge_iterator	iterator past the last edge (inbound and outbound) in global iteration space
$x.edges().lbegin()$	local_edge_iterator	iterator to the first edge (inbound and outbound) in local iteration space
$x.edges().lend()$	local_edge_iterator	iterator past the last edge (inbound and outbound) in local iteration space

Expression	**Return**	**Semantics**
$x[it_v].edges().begin()$	global_edge_iterator	iterator to the first edge (inbound and outbound) connected to the vertex pointed to by it_v
$x[it_v].edges().end()$	global_edge_iterator	iterator past the last edge (inbound and outbound) connected to to the vertex pointed to by it_v
$x[lit_v].edges().lbegin()$	local_edge_iterator	iterator to the first edge (inbound and outbound) connected to the vertex pointed to by lit_v
$x[lit_v].edges().lend()$	local_edge_iterator	iterator past the last edge (inbound and outbound) connected to to the vertex pointed to by lit_v
$x.edges().size()$	vertex_size_type	amount of globally visible edges
$x.edges().size(proc)$	edge_size_type	amount of globally visible edges owned by a processor *proc*
$x.edges().lsize()$	edge_size_type	amount of locally visible edges on the respective processor
$x.edges().empty()$	bool	true if there are more than 0 edges globally visible
$x.edges().max\_size()$	edge_size_type	maximum amount of edges possible

Conditions

Expression	**Precondition**	**Postcondition**
$x.edges().begin()$		
$x.edges().end()$		
$x.edges().lbegin()$		
$x.edges().lend()$		
$x[it_v].edges().begin()$	it_v points to a valid vertex in X	
$x[it_v].edges().end()$	it_v points to a valid vertex in X	
$x[lit_v].edges().lbegin()$	lit_v points to a valid vertex in X	
$x[lit_v].edges().lend()$	lit_v points to a valid vertex in X	
$x.edges().size()$		
$x.edges().size(proc)$		processor identified by *proc*
$x.edges().lsize()$		
$x.edges().empty()$		

Expression	**Precondition**	**Postcondition**
$x.edges().max\_size()$		

List of Figures

Bibliography

[BBAB+09] Bader, David A. ; Berry, Jonathan ; Amos-Binks, Adam ; Chavarría-Miranda, Daniel ; Hastings, Charles ; Madduri, Kamesh ; Poulos, Steven C.: Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. In: *Georgia Institute of Technology, Tech. Rep* (2009)

[BBMW09] Barrett, Brian W. ; Berry, Jonathan W. ; Murphy, Richard C. ; Wheeler, Kyle B.: Implementing a portable multi-threaded graph library: The MTGL on Qthreads. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* IEEE, 2009, S. 1–8

[BG11] Buluç, Aydın ; Gilbert, John R.: The Combinatorial BLAS: Design, implementation, and applications. In: *The International Journal of High Performance Computing Applications* 25 (2011), Nr. 4, S. 496–509

[BHKK07] Berry, Jonathan W. ; Hendrickson, Bruce ; Kahan, Simon ; Konecny, Petr: Software and algorithms for graph queries on multithreaded architectures. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* IEEE, 2007, S. 1–14

[BM11] Buluç, Aydin ; Madduri, Kamesh: Parallel breadth-first search on distributed memory systems. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* ACM, 2011, S. 65

[C+05] Consortium, UPC u. a.: UPC language specifications v1. 2. In: *Lawrence Berkeley National Laboratory* (2005)

[CAS10] Cong, Guojing ; Almasi, George ; Saraswat, Vijay: Fast PGAS Implementation of Distributed Graph Algorithms. (2010), 11, S. 1–11

[CB06] Cong, Guojing ; Bader, David A.: Designing irregular parallel algorithms with mutual exclusion and lock-free protocols. In: *Journal of Parallel and Distributed Computing* 66 (2006), Nr. 6, S. 854–866

[CCZ07] Chamberlain, Bradford L. ; Callahan, David ; Zima, Hans P.: Parallel programmability and the chapel language. In: *The International Journal of High Performance Computing Applications* 21 (2007), Nr. 3, S. 291–312

[CFS99] Carter, Larry ; Feo, John ; Snavely, Allan: Performance and Programming Experience on the Tera MTA. In: *PPSC*, 1999

[CGS+05] Charles, Philippe ; Grothoff, Christian ; Saraswat, Vijay ; Donawa, Christopher ; Kielstra, Allan ; Ebcioglu, Kemal ; Von Praun, Christoph ; Sarkar, Vivek: X10: an object-oriented approach to non-uniform cluster computing. In: *Acm Sigplan Notices* Bd. 40 ACM, 2005, S. 519–538

[CLRS09] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. – ISBN 0262033844, 9780262033848

[CZF04] Chakrabarti, Deepayan ; Zhan, Yiping ; Faloutsos, Christos: R-MAT: A recursive model for graph mining. In: *Proceedings of the 2004 SIAM International Conference on Data Mining* SIAM, 2004, S. 442–446

[DHS12] Dayarathna, Miyuru ; Houngkaew, Charuwat ; Suzumura, Toyotaro: Introducing ScaleGraph: an X10 library for billion scale graph analytics. In: *Proceedings of the 2012 ACM SIGPLAN X10 Workshop* ACM, 2012, S. 6

[Dij59] Dijkstra, E. W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik* 1 (1959), Dec, Nr. 1, 269–271. `http://dx.doi.org/10.1007/BF01386390`. – DOI 10.1007/BF01386390. – ISSN 0945–3245

[DSB99] Di Stefano, Luigi ; Bulgarelli, Andrea: A simple and efficient connected components labeling algorithm. In: *Image Analysis and Processing, 1999. Proceedings. International Conference on* IEEE, 1999, S. 322–327

[ECGS92] Eicken, TV ; Culler, David E. ; Goldstein, Seth C. ; Schauser, Klaus E.: Active messages: a mechanism for integrated communication and computation. In: *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on* IEEE, 1992, S. 256–266

[EWHL10] Edmonds, Nicholas ; Willcock, Jeremiah ; Hoefler, T ; Lumsdaine, A: Design of a large-scale hybrid-parallel graph library. In: *International Conference on High Performance Computing, Student Research Symposium, Goa, India*, 2010

[FAR+12] Fidel, Adam ; Amato, Nancy M. ; Rauchwerger, Lawrence u. a.: The stapl parallel graph library. In: *International Workshop on Languages and Compilers for Parallel Computing* Springer, 2012, S. 46–60

[FFK16a] FÃ¼rlinger, Karl ; Fuchs, Tobias ; Kowalewski, Roger: DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. (2016), 12

[FFK16b] Fürlinger, Karl ; Fuchs, Tobias ; Kowalewski, Roger: DASH: a C++ PGAS library for distributed data structures and parallel algorithms. In: *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on* IEEE, 2016, S. 983–990

[For12] Forum, Message Passing I.: *MPI: A Message-Passing Interface Standard Version 3.0*. 09 2012. – Chapter author for Collective Communication, Process Topologies, and One Sided Communications

[FR11] Fineman, Jeremy T. ; Robinson, Eric: Fundamental graph algorithms. In: *Graph Algorithms in the Language of Linear Algebra* 22 (2011), S. 45

[GH85] Graham, Ronald L. ; Hell, Pavol: On the history of the minimum spanning tree problem. In: *Annals of the History of Computing* 7 (1985), Nr. 1, S. 43–57

[GL05] Gregor, Douglas ; Lumsdaine, Andrew: The parallel BGL: A generic library for distributed graph computations. In: *Parallel Object-Oriented Scientific Computing (POOSC)* 2 (2005), S. 1–18

[GS13] Grünewald, Daniel ; Simmendinger, Christian: The GASPI API specification and its implementation GPI 2.0. In: *7th International Conference on PGAS Programming Models* Bd. 243, 2013

[Gus88] Gustafson, John L.: Reevaluating Amdahl's Law. In: *Commun. ACM* 31 (1988), Mai, Nr. 5, 532–533. `http://dx.doi.org/10.1145/42411.42415`. – DOI 10.1145/42411.42415. – ISSN 0001–0782

[ISO12] ISO: *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland : International Organization for Standardization, 2012. – 1338 (est.) S. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372`

[ISO15] ISO: *C++ Extensions for Concepts*. Geneva, Switzerland : International Organization for Standardization, 2015 `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4553.pdf`

[ISO17] ISO: *ISO/IEC 14882:2017 Working Draft, Standard for Programming Language C++*. International Organization for Standardization, 2017

[KAA+13] Khayyat, Zuhair ; Awara, Karim ; Alonazi, Amani ; Jamjoom, Hani ; Williams, Dan ; Kalnis, Panos: Mizan: a system for dynamic load balancing in large-scale graph processing. In: *Proceedings of the 8th ACM European Conference on Computer Systems* ACM, 2013, S. 169–182

[KBG12] Kyrola, Aapo ; Blelloch, Guy E. ; Guestrin, Carlos: Graphchi: Large-scale graph computation on just a pc USENIX, 2012

[KDSA08] Kim, John ; Dally, Wiliam J. ; Scott, Steve ; Abts, Dennis: Technology-driven, highly-scalable dragonfly topology. In: *ACM SIGARCH Computer Architecture News* Bd. 36 IEEE Computer Society, 2008, S. 77–88

[LAB+12] Lugowski, Adam ; Alber, David ; Buluç, Aydm ; Gilbert, John R. ; Reinhardt, Steve ; Teng, Yun ; Waranis, Andrew: A flexible open-source toolbox for scalable complex graph analysis. In: *Proceedings of the 2012 SIAM International Conference on Data Mining* SIAM, 2012, S. 930–941

[Lam13] Lameter, Christoph: NUMA (Non-Uniform Memory Access): An Overview. In: *Queue* 11 (2013), Juli, Nr. 7, 40:40–40:51. `http://dx.doi.org/10.1145/2508834.2513149`. – DOI 10.1145/2508834.2513149. – ISSN 1542–7730

[LBG+12] Low, Yucheng ; Bickson, Danny ; Gonzalez, Joseph ; Guestrin, Carlos ; Kyrola, Aapo ; Hellerstein, Joseph M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. In: *Proceedings of the VLDB Endowment* 5 (2012), Nr. 8, S. 716–727

[LGHB07] Lumsdaine, Andrew ; Gregor, Douglas ; Hendrickson, Bruce ; Berry, Jonathan: Challenges in parallel graph processing. In: *Parallel Processing Letters* 17 (2007), Nr. 01, S. 5–20

[MAB+10] Malewicz, Grzegorz ; Austern, Matthew H. ; Bik, Aart J. ; Dehnert, James C. ; Horn, Ilan ; Leiser, Naty ; Czajkowski, Grzegorz: Pregel: a system for large-scale graph processing. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* ACM, 2010, S. 135–146

[NR98] Numrich, Robert W. ; Reid, John: Co-Array Fortran for parallel programming. In: *ACM Sigplan Fortran Forum* Bd. 17 ACM, 1998, S. 1–31

[PJ98] Palsberg, Jens ; Jay, C B.: The essence of the visitor pattern. In: *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International* IEEE, 1998, S. 9–15

[Pri57] Prim, R. C.: Shortest Connection Networks And Some Generalizations. In: *Bell System Technical Journal* 36 (1957), Nr. 6, 1389–1401. http://dx.doi.org/10.1002/j.1538-7305.1957.tb01515.x. – DOI 10.1002/j.1538–7305.1957.tb01515.x. – ISSN 1538–7305

[PTM96] Protic, J. ; Tomasevic, M. ; Milutinovic, V.: Distributed shared memory: concepts and systems. In: *IEEE Parallel Distributed Technology: Systems Applications* 4 (1996), Summer, Nr. 2, S. 63–71. http://dx.doi.org/10.1109/88.494605. – DOI 10.1109/88.494605. – ISSN 1063–6552

[Saa03] Saad, Youssef: *Iterative Methods for Sparse Linear Systems*. 2003. http://dx.doi.org/10.1137/1.9780898718003.bm. http://dx.doi.org/10.1137/1.9780898718003.bm

[SAB+10] Saraswat, Vijay ; Almasi, George ; Bikshandi, Ganesh ; Cascaval, Calin ; Cunningham, David ; Grove, David ; Kodali, Sreedhar ; Peshansky, Igor ; Tardieu, Olivier: The asynchronous partitioned global address space model. In: *The First Workshop on Advances in Message Passing*, 2010, S. 1–8

[SB13] Shun, Julian ; Blelloch, Guy E.: Ligra: a lightweight graph processing framework for shared memory. In: *ACM Sigplan Notices* Bd. 48 ACM, 2013, S. 135–146

[SHS03] Suzuki, Kenji ; Horiba, Isao ; Sugie, Noboru: Linear-time connected-component labeling based on sequential local operations. In: *Computer Vision and Image Understanding* 89 (2003), Nr. 1, S. 1–23

[SL95] Stepanov, Alexander ; Lee, Meng: *The standard template library*. Bd. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995

[SLL01] Siek, Jeremy G. ; Lee, Lie-Quan ; Lumsdaine, Andrew: *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001

[SU15] Suzumura, Toyotaro ; Ueno, Koji: ScaleGraph: A high-performance library for billion-scale graph analytics. In: *Big Data (Big Data), 2015 IEEE International Conference on* IEEE, 2015, S. 76–84

[Val90] Valiant, Leslie G.: A bridging model for parallel computation. In: *Communications of the ACM* 33 (1990), Nr. 8, S. 103–111

[ZMI+14] Zhou, Huan ; Mhedheb, Yousri ; Idrees, Kamran ; Glass, Colin W. ; Gracia, José ; Fürlinger, Karl: DART-MPI: an MPI-based implementation of a PGAS runtime system. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* ACM, 2014, S. 3

[ZZZ+14] Zhao, D. ; Zhang, Z. ; Zhou, X. ; Li, T. ; Wang, K. ; Kimpe, D. ; Carns, P. ; Ross, R. ; Raicu, I.: FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: *2014 IEEE International Conference on Big Data (Big Data)*, 2014, S. 61–70