SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A System for Distributed Hypergraph Processing

Lukas Denk

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A System for Distributed Hypergraph Processing

Ein System für das Verteilte Verarbeiten von Hypergraphen

Author:	Lukas Denk
Supervisor:	Prof. Dr. rer. nat. Ruben Mayer
Advisor:	Nikolai Merkel
Submission Date:	16.10.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 16.10.2023 Lukas Denk

Abstract

Graphs and graph algorithms are very successful in modeling and analyzing networks with entity-to-entity relationships [Sta07; TL10; BF97]. However, graph edges are always between two entities and thus unable to directly represent group relationships. A hypergraph - a graph generalization where edges can be between an arbitrary number of vertices - is therefore often the better choice to model networks with group relationships [DY08; DB14; ZHS06]. Implementing algorithms for hypergraphs from scratch on is cumbersome. For this reason, several frameworks have been developed [Shu20; Hei+19; Jia+18]. However, they are either distributed and slow or run on a single machine - which restricts the hypergraphs that can be processed.
To solve this dilemma, we introduce *Dyper*, a distributed hypergraph processing system which is distributed *and* fast. It allows to implement hypergraph algorithms in a message based way, where hypervertices and hyperedges send messages to each other. The user only has to define a few local methods on the hypervertices and hyperedges. The rest of the algorithm, like delivering messages, is then taken care of by the framework.
Our experiments on up to 8 2-core nodes showed that Dyper is 4 to 14 times faster than other distributed frameworks.

Contents

1 Introduction

A graph consists of entities, called *vertices*, and relations among entity pairs, called *edges* [Tru93]. Graphs and graph algorithms are very popular to represent and learn about networks from various domains like social media [TL10], transport routes [Guz19] and many others [Sta07; BF97; Bha+12].
However, the restriction of edges to be between two vertices can be an obstacle in domains with group relationships. In such cases, lifting the restriction is often the better choice. Such a graph generalization, where edges are between an arbitrary set of vertices, is called a hypergraph [nist2]. Vertices and edges in a hypergraphs are called *hypervertices* and *hyperedges*, respectively [nist2]. Hypergraph approaches outperform graph approaches in areas like spectral clustering, image segmentation, and many others [DY08; DB14; ZHS06].
Writing a hypergraph algorithm from scratch on is cumbersome. Real world hypergraphs are often huge, sometimes consisting of billions of hypervertices and hyperedges [LK14]. Hypergraph algorithms must therefore run efficiently and sometimes even distributed. Fortunately, many algorithms can be formulated in a message based approach where hypervertices and hyperedges send messages in multiple supersteps [Shu20]. This allows frameworks to take over the message sending and thus enable the implementation of many hypergraph algorithms with just a few extra lines. To the best of our knowledge, there exist only three hypergraph processing frameworks. Unfortunately, they all have their limitations: MESH [Hei+19] and HyperX [Jia+18] both enable a distributed processing but build on top of slow technology. Hygra [Shu20], on the other hand, runs fast but only on a single machine.

Dyper. For this reason, we introduce Dyper[1], a framework that processes hypergraphs efficiently and distributed. It is written in Rust, which has a C++ like performance [PLJ23] and consists of two parts: The first part is a partitioner that takes a hypergraph and partitions it into several partitions. By default, it partitions the set of hypervertices and then replicates each hyperedge to the partitions whose hypervertices have the hyperedge as an in-hyperedge. The partitioner can also partition the set of hyperedges. In this case, it replicates each hypervertex to the partitions on which its

[1]for **D**istributed H**yper**graph Processing

out-hyperedges are. The partitioning is random by default but customizable.
The second part of the program takes a partition and runs a chosen algorithm together with other nodes managing the other partitions. In each round, the program first processes the active hypervertices and then the active hyperedges. The processing of the hypergraph elements is parallelized. An element is active when it wants to send messages in the current superstep. Our program switches between a sparse and dense traversal of the hypergraph elements. The sparse traversal only iterates over the active elements while the dense traversal iterates over all elements and checks each element's activity before initiating it to send messages. The switching between a sparse and dense mode optimizes our program for dense algorithms, where most hypergraph elements are active and for frontier based algorithms, where only a few hypergraph elements are active.
Dyper works with weighed and unweighed hypergraphs. It has four algorithms pre-installed: PageRank, ConnectedComponent, SingleSourceShortestPath and LabelPropagation. Implementing further algorithms is straightforward and only requires to overwrite a few methods and define two small structs.

Evaluation. We also provide an evaluation of Dyper. As expected, the average running time of a superstep decreases significantly with the number of used nodes. Executing, e.g., LabelPropagation on eight instead of one node reduces the average superstep running time to less than a sixth. Moreover, comparing Dyper to other frameworks yielded astonishing results: Running a superstep in PageRank on Dyper is up to fifteen times faster on eight two core machines than what MESH reports for eight twelve core machines for the same hypergraph (~4 vs ~60 seconds) [Hei+19]. This is even more remarkable when taking into account that MESH optimized the partitioning of the hypergraph while we partitioned it randomly. HyperX does not implement PageRank but provides results for the LabelPropagation algorithm. When we tested Dyper, we found that it was about four times faster than HyperX on the same hypergraph while only using half the number of cores (~3.6 vs ~14.0 seconds) [Jia+18]. Hygra, on the other hand, beats Dyper on the same single two core machine for the same hypergraph by roughly two and six to one for the ConnectedComponent and PageRank algorithm (~4.6 vs ~11.0 seconds and ~3.1 vs ~17.9 seconds, respectively). This is probably due to the fact that Hygra is optimized for a single machine [Shu20].

Outline. The remaining parts of this thesis is structured as follows: In Chapter 2, we give some background to message based hypergraph algorithms and the Rust programming language. We then discuss related work in Chapter 3 before we continue with Chapter 4, in which we outline the implementation of the framework and the four algorithms. In Chapter 5, we evaluate the performance of our framework before we finally give a conclusion in Chapter 6.

2 Background

In this chapter, we provide the background information necessary to understand how a hypergraph is implemented in a message based way and how this enables us to build a framework for processing hypergraphs in a distributed system. We start with explaining distributed message based graph algorithms in Section 2.1 before making a carryover to distributed message based hypergraph algorithms in Section 2.2. In Section 2.3, we shortly introduce Rust, the programming language we used for our program.

2.1 Distributed Message Based Graph Algorithms

Many real-world graphs are too big for a single machine. Hence, they must be processed on a distributed system. However, implementing a custom infrastructure for every graph algorithm takes a lot of work. Fortunately, many algorithms can be implemented by defining an algorithm specific struct for the vertices with a method that sends messages to the vertices' out-neighbors. A framework can then provide the infrastructure for the execution of the local method and the delivery of the messages [Mal+10].
In this section, we describe how to formulate a graph algorithm in a message based way (see Section 2.1.1) and how this enables graph algorithm frameworks. We then outline how a graph can be partitioned to process it on a distributed system (c.f. Section 2.1.2). We continue with an important optimization that works with many message based algorithms (in Section 2.1.3) before demonstrating an example implementation (see Section 2.1.4).

2.1.1 Message Based Graph Algorithms

A graph algorithm can be implemented with a message based approach as follows: Every vertex knows its out-neighbors, has a unique id, a state and an `update` method. A vertex can be active or inactive. Its implementation is algorithm specific.
An algorithm runs in several supersteps. In each superstep, the framework executes the `update` method of every active vertex. In its `update` method, a vertex can update its state, see the messages it received in the last superstep, send messages to its out-

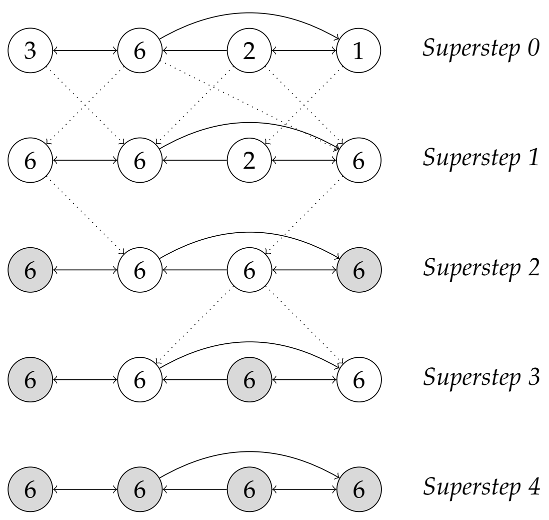


Figure 2.1: Finding the maximum value in a strongly connected graph. Arrows indicate edges, dotted lines messages. Gray vertices are inactive. Shows the activity at the beginning of a superstep.

neighbors and deactivate itself. The framework then takes care of the message delivery. A vertex that receives a message becomes active in the next superstep, even when it deactivated itself in the current superstep. The program ends if all vertices are inactive or a maximal number of supersteps is exceeded.

The execution order of the vertices' `update` methods is undefined. Furthermore, every vertex only works with the data associated to itself. This makes it very easy to process the graph in parallel on multiple machines.

In Figure 2.1, we illustrate how to find the maximal vertex value in a strongly connected graph with a message based approach. A strongly connected graph is a directed graph in which every vertex is reachable from every other vertex [nist1]. In Superstep 0, all vertices are active and send their value to their out-neighbors. In every other superstep, the active vertices check whether they have received a message with a value higher than their own value. If so, they update to that value and send it to their out-neighbors. Otherwise, their out-neighbors already know their value and the vertices refrain from sending messages. At the end of a superstep, the vertices deactivate themselves [Mal+10].

2.1.2 Graph Partitioning

If we want to run a graph algorithm in a distributed system, we must partition the graph. For the message based approach, we simply split the set of vertices into several partitions and assign each partition to a different node. A simple and fast approach is a random split. However, a more sophisticated approach may reduce the number of messages sent over the network and thus speed up the graph processing. On the other hand, finding a perfect partitioning, e.g. regarding to the number of cut edges, is infeasible [GJS76]. For this reason, common graph partitioning algorithms like [Kar] use heuristics. Distributed graph processing is a trade-off between partitioning and processing runtime.

2.1.3 Combining Messages

Sending a message over the network or storing all the messages sent to a vertex for the next superstep is expensive in terms of storage and network traffic. Fortunately, many graph problems allow to combine the messages in a commutative and associative way. In this case, the user can implement a `combine` function that combines two messages into a single one. If we want to, e.g., find the maximum vertex value in a strongly connected graph and send the vertices' values in the messages, then the function would take the higher value. This allows a machine to combine all the messages sent to a remote vertex before sending them as a single message. Additionally, a machine can combine the messages received for a local vertex before making them available in the next superstep [Mal+10].

2.1.4 Example of a Message Based Graph Algorithm

We now show how to find the highest vertex value in a strongly connected graph with a message based approach. We only need to implement a `combine` function and the algorithm specific vertex struct with its `update` method. The framework takes care of the rest.
In the algorithm, every message contains the value of a vertex. The `combine` function takes two messages and returns the higher one. The vertex struct contains the vertex's out-neighborhood and status. The status is the highest value the vertex has learned so far. In Superstep 0, all vertices are active and the `update` method of a vertex sends the vertex's status to the vertex's out-neighbors. In every other superstep, the `update` method checks whether the vertex has received a message in the previous superstep and in case it has, if the combined message is higher than the vertex's current status. If so, it updates the status to the message and sends it to the vertex's out-neighbors. In every superstep, the `update` method finally deactivates the vertex. The pseudocode for

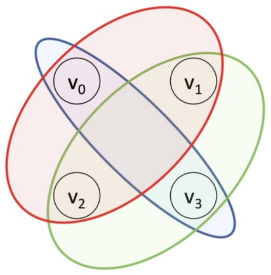
the example is given in Algorithm 1.

Algorithm 1 Propagating the highest vertex value in a strongly connected graph

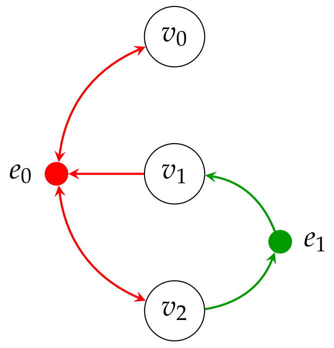
```
function COMBINE($msg1$, $msg2$)
    return MAX($msg1$, $msg2$)
end function

struct VERTEX
    $status$,
    ▷Ids of the vertex's local out-neighbors
    $out\_neighbors$
end struct

implementation for VERTEX
    ▷ $self$ references to the vertex. $comb\_msg$ to the combined message from the last superstep.
    method UPDATE($self$, $comb\_msg$, $superstep$)
        if $superstep = 0$ then
            SEND($self.out\_neighbors$, $self.status$){send status to out-neighbors}
        else if $comb\_msg$ != $nil$ and $self.status < comb\_msg$ then
            $self.status = comb\_msg$
            SEND($self.out\_neighbors$, $self.status$)
        end if
        DEACTIVATE($self$)
    end method
end implementation
```



(a) An undirected hypergraph with hyperedges $e_0 = \{v_0, v_1, v_2\}$, $e_1 = \{v_1, v_2, v_3\}$ and $e_2 = \{v_0, v_3\}$ (the illustration is taken from [Shu20]).



(b) A directed hypergraph with hyperedges $e_0 = (\{v_0, v_1, v_2\}, \{v_0, v_2\})$ and $e_1 = (\{v_2\}, \{v_1\})$. The 1st set of a hyperedge tuple is the in-set, the 2nd the out-set.

Figure 2.2: An undirected and a directed hypergraph.

2.2 Distributed Message Based Hypergraph Algorithms

In this section, we adapt the message based formulation of graph algorithms from Section 2.1 to hypergraph algorithms and show how this enables frameworks for implementing hypergraph algorithms on a distributed system. We first explain what a hypergraph is (c.f. Section 2.2.1) and how it can be represented as a bipartite graph (see Section 2.2.2). We then describe how hypergraph algorithms can often be formulated with a message based approach (in Section 2.2.3) before we give an example of a message based hypergraph algorithm (c.f. Section 2.2.4). Last but not least, we explain how a hypergraph is efficiently partitioned (see Section 2.2.5).

2.2.1 Hypergraphs

A hypergraph is a graph generalization where edges are between an arbitrary set of vertices. Vertices and edges in a hypergraph are called *hypervertices* and *hyperedges*, respectively [Ber84]. A directed hyperedge consists of two sets of hypervertices: An in- and an out-set, containing the hypervertices that have the hyperedge as an outgoing and incoming hyperedge, respectively [Ber84]. Figure 2.2 shows examples of an undirected and directed hypergraph.

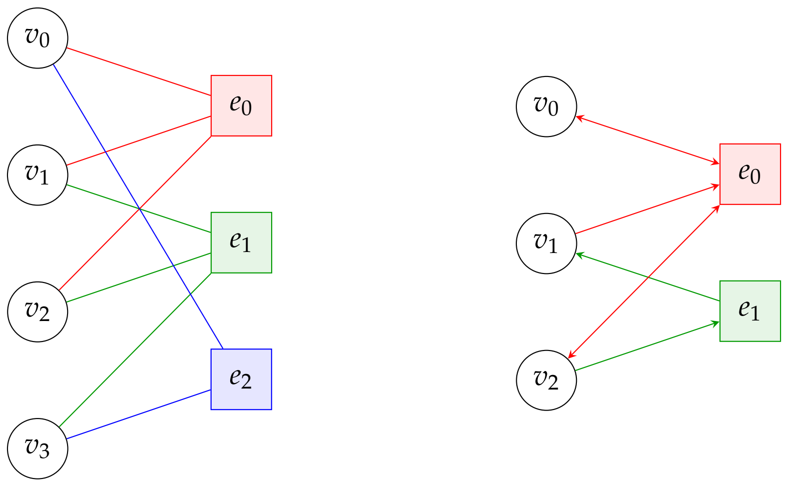


Figure 2.3: The bipartite representation of the undirected and directed hypergraphs from Figure 2.2.

2.2.2 Representing a Hypergraph as a Bipartite Graph

A hypergraph can easily be represented as a bipartite graph. In a bipartite graph, the vertices can be divided into two disjoint and independent subsets V and W. A set of vertices is independent if its members have no edges among themselves. In a bipartite presentation of a hypergraph, every hypervertex v is represented by a vertex v' in V and every hyperedge e by a vertex e' in W. If v is part of an undirected hyperedge e, then v' has an undirected edge with e'. If v has e as an outgoing or incoming hyperedge, then v' has an outgoing or incoming edge to or from e', respectively. Figure 2.3 shows the bipartite representations of the hypergraphs from Figure 2.2 [Shu20].
From now on, we will always view a hypergraph as a bipartite graph. For the sake of simplicity, we will therefore just say *hypervertices* to the vertices in the graph representation representing the hypervertices and *hyperedges* to the vertices representing the hyperedges. When we speak of the neighbors of a hypergraph element, we mean its neighbors in the bipartite graph.

2.2.3 Message Based Hypergraph Algorithms

The representation of hypergraphs as bipartite graphs enables us to implement many hypergraph algorithms in a message based style. To do so, we adapt the framework for message based graph algorithms from Section 2.1. To profit from the special structure of the bipartite graph, we distinguish between the hypervertices and hyperedges. The vertex struct is now implemented twice, once for the hypervertices and once for the hyperedges. The hypervertices' out-neighbors are then hyperedges and the hyperedges' out-neighbors are hypervertices. Ids are only unique among hypervertices or hyperedges, meaning it is possible for a hypervertex and a hyperedge to share the same id. Each superstep now consists of two phases: One phase executes the hypervertices' `update` function and the other one the hyperedges' `update` function [Hei+19]. Furthermore, when a hypergraph element receives a message, it becomes active in the next phase, which is not necessarily in the next superstep.

2.2.4 Example of a Message Based Hypergraph Algorithm

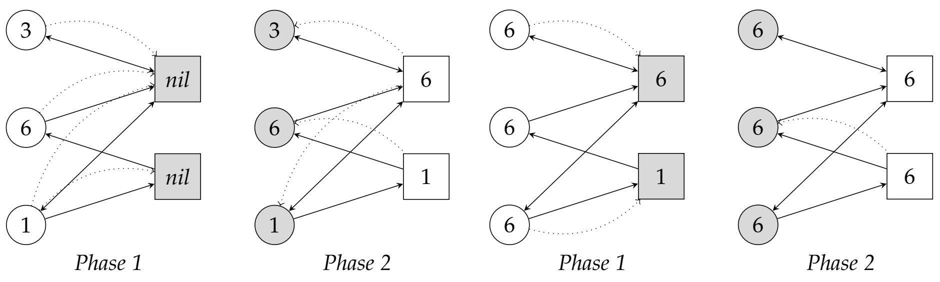
Message based hypergraph algorithms are often quite similar to their graph pendants. This also holds for finding the highest hypervertex value in a strongly connected hypergraph (for the graph version, see Section 2.1.4). The `combine` functions in the graph and hypergraph versions are identical, as is the vertex and hypervertex struct. The hyperedge struct is also very similar to the vertex struct, except that its `update` method misses a special case for Superstep 0.
Algorithm 2, shows the hyperedges' `update` method. The hypervertex and hyperedge struct definition, hypervertex implementation and the `combine` function match with Algorithm 1's vertex struct definition and implementation as well as Algorithm 1's `combine` function. In Figure 2.4, we give a small example execution of the algorithm.

Algorithm 2 The hyperedges' `update` method for propagating the highest hypervertex value in a strongly connected hypergraph

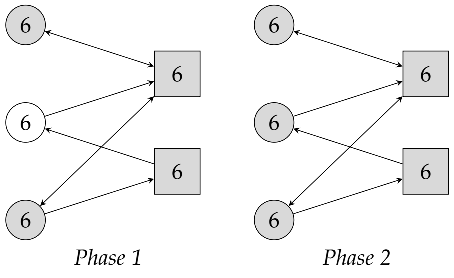
```
implement HYPEREDGE
  method UPDATE($self$, $comb\_msg$)
    if $comb\_msg$ != $nil$ and $self.status < comb\_msg$ then
      $self.status = comb\_msg$
      SEND($self.out\_neighbors$, $self.status$)
    end if
  DEACTIVATE($self$)
  end method
end implementation
```

2.2.5 Hypergraph Partitioning

To partition a hypergraph, one could simply partition its bipartite graph representation like a normal graph (c.f. Section 2.1.2). However, this would not take advantage of typical hypergraph properties. Close hypervertices are, e.g. often part of the same



(a) *Superstep 0* (b) *Superstep 1*



(c) *Superstep 2*

Figure 2.4: Propagating the maximum hypervertex value in a strongly connected hypergraph. Circles represent hypervertices, squares hyperedges. Arrows indicate edges, dotted lines messages. Gray hypergraph elements are inactive. Shows the activities at the beginning of a phase.

hyperedges [LKa; LKc]. The following approach uses this property and avoids network messages in Phase 1: Let either R and U be the set of hypervertices and hyperedges or the set of hyperedges and hypervertices, respectively. In any case, we partition U into different partitions. To every partition, we then add a representation of all the hypergraph elements in W that have an outgoing edge to an U-element in the partition. An element in U is now present on only one node. Thus, we call it a *unique hypergraph element*. Furthermore, we say *unique* to the representation of a unique hypergraph element. An element from R is sometimes present on multiple nodes. We call an element in this set a *replica hypergraph element*. We use the term *replica* for a representation of this element, even though different representations of the same replica hypergraph element are not identical: Their id and status are but they only know their local out-neighborhood, which is different. We also use *partition elements* as an umbrella term for replicas and uniques.
In the superstep of message based hypergraph algorithm, Phase 1 always runs the `update` method of the active replicas and Phase 2 the `update` method of the active uniques. In Phase 1, the replicas send their messages to their local out-neighbors, which are the uniques on the same partition. In Phase 2, the uniques send their messages to all their out-neighbors.
The way of partitioning U strongly influences the number of replications as well as the number of messages sent over the network [Hei+19]. However, just like graph partitioning, hypergraph partitioning cannot generally optimize memory requirement and network traffic in a feasible time [May+18]. As a consequence, hypergraph partitioners like Hype always use heuristics [May+18]. Figure 2.5 gives an example partitioning of the hypergraph from Figure 2.2b.

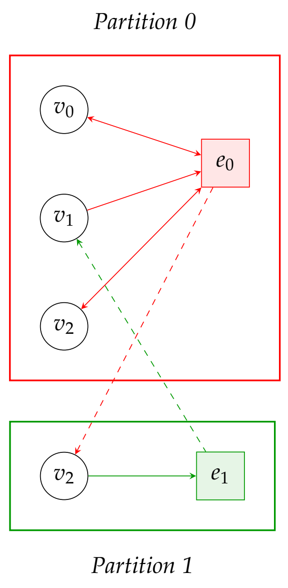


Figure 2.5: A 2-Partitioning of the directed hypergraph from Figure 2.3. The dashed edges are across different partitions.

2.3 Rust

Processing huge hypergraphs is resource intensive. Thus, it is important to use an efficient programming language for our system. We chose Rust since it has a C++ like performance [PLJ23] and guarantees memory safety at compile time. This is achieved by a revolutionary ownership concept [MK14].
In this section, we first describe how Rust programs are organized (in Section 2.3.1). We then outline the different types of data structures in Rust, show how to implement behaviour on them and explain how traits and generics are used (see Section 2.3.2). We continue with Rust's revolutionary memory management system (c.f. Section 2.3.3) before we introduce some ways to implement concurrency in Rust (see Section 2.3.4). We finally give an overview of the Rust types that we will encounter in this thesis (c.f. Section 2.3.5).

2.3.1 Modules

A Rust program is organized in modules. The top module is often called `main`. It contains the program's entry point, the `main` function. Every module is written in its own file and can have submodules, which each have their own file [KN].

2.3.2 Data Structures and Traits

Besides primitive data types like integers and floats, Rust also offers composite data structures like *tuples*, *structs* and *enums* [KN].

Tuples. A tuple is a fixed-sized ordered collection of elements of potentially different types. They are created with round brackets as shown below:

```
let bello: (String, bool) = (String::from("Bello"), true);
```

`String::from("Bello")` calls the static `from` method from the `String` struct to create the `String` `"Bello"` . Here, `bello` is a tuple of type `(String, bool)`, where `bool` is a boolean [KN; std].

Enums. An enum is a data type with a finite set of values [KN]. An example is given below:

```
enum DogBreed {
    Poodle,
    Bulldog
}
```

Structs. A struct is a named type with named fields of potentially different types [KN]. An example is given below:

```
struct Dog {
    name: String,
    type: DogBreed
}
```

Methods. We can also implement static and instance methods on primitive data types, structs and enums [KN]. The code below shows method implementations on `Dog`:

```
impl Dog {
    fn new(name: String, type: DogBreed) -> Self {
        Dog {
            name: name,
            type: type
        }
    }
    fn bark(&self){
        println("Woof! My name is {}.", self.name);
    }
}
```

`new` is a static method and constructs a `Dog` instance. The method returns an instance of type `Self`, where `Self` is the type on which the method is implemented. `bark` is an instance method and therefore takes `&self`, a reference to the `Dog` instance on which it is called [KN].

Traits. Interfaces are called *trait*s in Rust. They are defined with the `Trait` keyword and can be implemented by any named type (which excludes tuples) [KN]. An example is given below:

```
trait Animal {
    fn eat(&self);
}
```

We use the syntax `impl <trait> for <type> {...}` to implement a trait for a type. The implementation must implement every trait method unless the trait has a default implementation for the method [KN]. An example is shown below:

```
impl Animal for Dog {
   fn eat(&self) {
        println("Mmhm. {} loves eating bones.", self.name);
    }
}
```

Generics and Constants. Generics and constants are placeholders for different data types and constants, respectively. They are specified inside angel brackets after the name of a struct, enum, method or function. A constant `N` is preceded by the `const` keyword and followed by its type, like `const N: u8`. The types represented by a generic `T` can be restricted to implementations of a specific trait by writing `T: <trait>` [KN]. An example is given below.

```
struct Cage<const N: usize, T: Animal> {
    inhabitants: [T; N]
}
```

`usize` is an unsigned integer whose size is platform dependent and equals to the address size of the machine. `[T; N]` is an array with `N` elements of type `T` [KN; std]:

2.3.3 Memory Management

Ownership. In Rust, each value is owned by a single variable. The value is automatically deallocated from memory once the owner goes out of scope. A variable can move its ownership to another variable by passing its value to the other variable. The old owner variable then becomes invalid [KN]. A short code snippet illustrates this:

```
{
    let a = String::from("Bello");
    let b = a; //ownership transfer
    println("{}", a); //compile error. a is invalid at this point
} //The String owned by b is deallocated from memory
```

Borrowing rules. An owner can lend references to its value. It can either lend multiple immutable references or a single mutable reference at a time. An owner can only mutate its value when there are no references to it. These rules ensure that the holder of a reference has a guarantee that the underlying value does not change unexpectedly. A reference becomes invalid when the owner goes out of scope or transfers its ownership. Rust ensures these rules at compile time [KN]. The code below illustrates them:

```
let a = String::from("Bello");
{
    let a_ref1 = &a; //creates a reference to a
    let a_ref2 = &a;
    let a_ref_mut = &mut a; //Compile error. Cannot create a mutable
        reference to a variable that is already borrowed.
} //references go out of scope
let a_ref_mut1 = &mut a;
let a_ref_mut2 = &mut a; //Compile error. b already has a mutable
    reference.
```

Inner Mutability. In some cases, a single mutable reference to a value may not be enough. For this reason, some data structures implement the concept of inner mutability, where one can change the state of the data structure with an immutable reference to it. This is possible by explicitly ignoring the strict borrowing rules. Usually, developers do not disable these rules directly but instead use data structures with inner mutability from well tested libraries [KN].

2.3.4 Concurrency

Asynchronous programming. We developed a distributed system where every node is always connected to every peer at the same time. Managing every connection by an own operating system thread is very expensive, since it results in a lot of thread context switching [Bla]. Instead, we handle these connections with the asynchronous runtime `tokio` which is able to efficiently manage multiple concurrent tasks on just a few threads [toktut].
To run a code section concurrently, we spawn a `tokio task`. The values sent to the task must have a `'static` lifetime and implement the trait `Send`. The reason is that `tokio` may execute the task on different threads. `Send` indicates that a type can safely be sent to another thread. It is a marker trait and thus has no methods. A type is `Send` if it either implements `Send` explicitly or if its members are all `Send`. A `'static` lifetime ensures that the lifetime of the value is independent from any other thread. A simple reference to a non-static value is, for example, not `'static`. The compiler cannot detect whether a reference on one thread outlives a value on another thread [toktut; toklib].

Accessing values concurrently. Many use cases require to access the same value concurrently from different threads. The `Arc` struct - `Arc` stands for atomic reference counter - makes this possible. An `Arc<T>` instance is a thread-safe immutable reference that points to a value of type `T`. Such a reference can be cloned and, if `T` implements `Sync` and `Send`, also sent to another thread. The `Sync` trait marks a type as thread-safe. A type `T` is `Sync` if it implements `Sync` or if the compiler decides that a reference to `T` is `Send`. An instance of such a type can be immutably referenced by multiple threads at the same time. `Arc` references pointing to the same value share an atomic counter to track the number of references to that value. The value is deallocated once the last reference is dropped.

Mutating values concurrently. We can mutate a value concurrently from different threads by letting an `Arc` instance hold an instance of a type that implements `Sync`, `Send` and inner mutability. An example for such a type is `Mutex<T>`. It wraps a value of type `T` and uses a lock to ensure that only one thread accesses the value at a time.

2.3.5 Some Rust Types

In this section, we give an overview of Rust types that we will encounter in this thesis. All types implement `Send` and `Sync` unless stated otherwise. The types are all from the standard library [std], except for `DashMap` and `DashSet`, which are from the `dashmap` library [dash] and `UnboundedSender`, `UnboundedReceiver`, `BufWriter`, `BufReader`, `OwnedWriteHalf`, `OwnedReadHalf`, and `JoinHandle`, which are from the `tokio` library [toklib].

Type or Trait	**Description**
`bool`	A boolean.
`u8`, `u16`, `u32`	An unsigned 8, 16 and 32-bit integer. `u8` is also used as a byte.
`usize`	An unsigned integer whose size is platform dependent and equals to the address size of the machine.
`str`	A primitive string.
`Option<T>`	An enum used to hold an optional value. The enum is `Some(T)` if a value is present and `None` otherwise. Only `Send` or `Sync` if `T` implements `Send` or `Sync`, respectively.
`[T; N]`	A fixed-sized array holding `N` values of type `T`. Only `Send` or `Sync` if `T` implements `Send` or `Sync`, respectively.
`Vec<T>`	A dynamic array list holding values of type `T`. Only `Send` or `Sync` if `T` implements `Send` or `Sync`, respectively.
`HashMap<K, V>`	A map that maps values of type `K` to values of type `V`. Only `Send` or `Sync` if both, `K` and `V` implement `Send` or `Sync`, respectively.
`Fn(T1,...,Tn) -> R`	A trait for a closure that takes parameters of type `T1,...,Tn` and returns a parameter of type `R`. A closure is a function with a state. Closures that return nothing omit the `-> R`. The compiler determines whether a closure is `Sync` or `Send` based on certain rules that are beyond the scope of this thesis.
`Send`	A trait that marks types that are safe to send across threads.

`Sync`	A trait that marks types that are thread-safe to access concurrently.
`Arc<T>`	A thread-safe smart reference that manages an instance of type `T`. Only `Send` and `Sync` if `T` implements both `Send` and `Sync`.
`Mutex<T>`	A mutex that holds an inner value of type `T`. Entities must acquire a lock from the mutex if they want to access its inner value. Holding a lock also allows to mutate the value. Only one entity can hold a lock at a time. The mutex uses inner mutability. To mutate the value, it is therefore enough to have an immutable reference to the mutex.
`RwLock<T>`	A read-write lock that holds an inner value of type `T`. Similar to `Mutex<T>`, except that it either issues multiple read locks or a single write lock to its inner value. A read lock only enables immutable access to the inner value. A write lock allows both, mutable and immutable access.
`AtomicBool`	An atomic boolean. Implements inner mutability to allow immutable references to update the boolean.
`DashMap<K, V>`	A map that uses inner mutability to allow immutable references to mutate the map.
`DashSet<T>`	A set that implements inner mutability to allow immutable references to mutate the set.
`UnboundedSender<T>` and `UnboundedReceiver<T>`	The write and read part of an unbounded `mpsc` channel that transmits values of type `T`. `mpsc` stands for multi producer single consumer. Only `Send` and `Sync` if `T` implements `Send`. `UnboundedSender` implements inner mutability to enable the sending of messages from immutable references.
`BufWriter<W>`	Buffers writes to a writer with type `W`. `W` must implement the `Write` trait. Only `Send` or `Sync` if `W` implements `Send` or `Sync`, respectively.
`BufReader<R>`	Buffers reads to a reader with type `R`. `R` must implement the `Read` trait. Only `Send` or `Sync` if `R` implements `Send` or `Sync`, respectively.
`OwnedWriteHalf` and `OwnedReadHalf`	The write and read half to a TCP socket.
`JoinHandle<()>`	A handle to a `tokio` `task` that allows to check the status of the task and wait for its termination.

3 Related Work

To the best of our knowledge, there are only three other hypergraph processing systems: *Hygra* [Shu20], *MESH* [Hei+19] and *HyperX* [Jia+18]. Hygra can only run on a single machine [Shu20]. MESH and HyperX, on the other hand, build on top of Apache Spark, a framework for running data analytics on a cluster [Hei+19; Jia+18]. MESH is able to partition by the set of hypervertices and hyperedges [Hei+19], HyperX only by the set of hypervertices [Jia+18]. Hygra and MESH internally work with the bipartite graph version of a hypergraph [Shu20; Hei+19]. They both use an adapted version of an already existing graph processing framework [Shu20; Hei+19]. Hygra builds on top of Ligra [Shu20], MESH on top of the Apache Spark extension GraphX [Hei+19]. HyperX works directly on hypergraphs [Jia+18].
All frameworks iterate over the hyperedges and hypervertices in parallel [Shu20; Hei+19; Jia+18]. HyperX and MESH always traverse the complete hypergraph, even if only a small part is active [Hei+19; Jia+18]. Hygra switches between a sparse and a dense iteration mode for both sets, the hypervertex and the hyperedge set.
HyperX, MESH, GraphX and Apache Spark are mainly written in Scala and run on the Java Virtual Runtime [Hei+19; Jia+18]. Hygra and Ligra are written in C++ [Shu20]. This and the fact that Hygra is not distributed is probably the reason why it by far the fastest of the three frameworks [Shu20; Hei+19; Jia+18].

4 Implementation

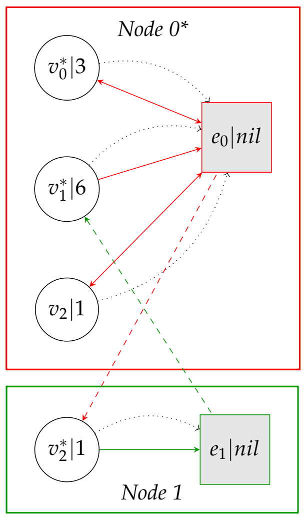
In this chapter, we outline the implementation of our program. We start with a short overview (in Section 4.1) and then explain the modules of our program. These are: The `main` module which parses the command line arguments and starts the program (c.f. Section 4.2). The `partitioner` module which partitions a hypergraph (see Section 4.3). The `algorithms` module which defines the `Replica` and `Unique` traits (c.f. Section 4.4). The `algorithms` module's submodules which each contain the algorithm specific code for one of the algorithms (see Section 4.5). The `importer` module which imports a partition (c.f. Section 4.6). The `peers` module which connects with the other nodes of the system (c.f. Section 4.7). And finally the `hypergraph` module which runs an algorithm together with the other nodes (see Section 4.8).

4.1 Overview

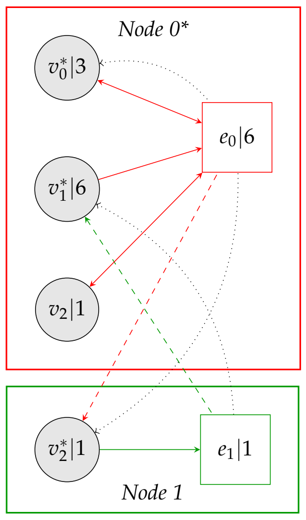
Dyper consists of two parts: The first part runs on a single node and partitions a hypergraph into N partitions, where N is the number of nodes the algorithm should run on.
The second part runs an algorithm distributed on several nodes. Each of the nodes is responsible for one of the partitions. The algorithm is based on the message based hypergraph algorithm outlined in Section 2.2 but has some modifications: A superstep consists of four instead of two phases. As in Section 2.2, Phase 1 and 2 execute the `update` methods of the active replicas and uniques, respectively. Unlike in Section 2.2, messages are received and processed in the same phase in which they are sent. Moreover, receiving a message does not automatically activate the receiver anymore. Instead, the receiver decides on the basis of the message whether to activate itself for the next phase.
In Phase 1, the replicas send their messages only to local out-neighbors. Phase 2 is more complicated. The uniques' send their messages to local and remote out-neighbors. A unique's out-neighbor may have replicas on multiple nodes. Sending the same message to all these replicas could lead to a lot of network traffic. For each replica hypergraph element, we therefore define one of its replicas as the *master replica*. The uniques then only send messages to the master replicas of their out-neighbors.
In Phase 3, every active master replica sends its status to the other replicas of the

hypergraph element it represents. We call these replicas *servant replicas*. Since the servant replicas belong to the same hypergraph element as their master and since the master is active, they also become active.
In Phase 4, we define the node with partition 0 as the *master node* and the other nodes as the *servant nodes*. Phase 4 determines whether the hypergraph is still active. This is the case when at least one of the nodes is active, meaning it has at least one active partition element. The servant nodes start the phase by telling the master partition whether they are still active. The master partition now knows the activity of the hypergraph and passes this information to the servant nodes. All nodes then terminate if the hypergraph is inactive. Otherwise, they continue with the next superstep unless they have reached the superstep limit defined by the user.

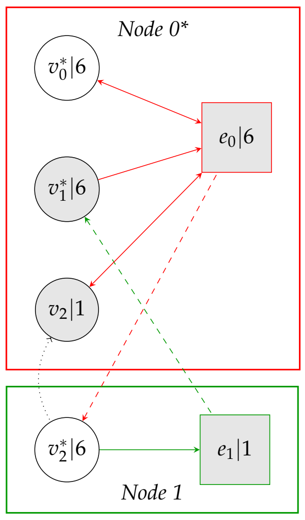
An example superstep. Figure 4.1 illustrates the four phases of the first superstep for propagating the highest hypervertex value in a strongly connected hypergraph with two nodes. In the given example, the hypervertices are replicas and the hyperedges uniques. The superstep works as follows: In the beginning, the replicas are active while the uniques are not. The replicas start with their initial value, the uniques with no value. If a partition element receives a message, it checks whether the value is higher than its current one. If so, it updates to the received value and activates itself. In Phase 1, the replicas send their status to their local out-neighbors and become inactive. In Phase 2, the active uniques send their status to the master replicas of their out-neighbors and become inactive. In Phase 3, the master replicas send their values to their servants. In Phase 4, the nodes determine whether the hypergraph is still active. If not, they terminate.



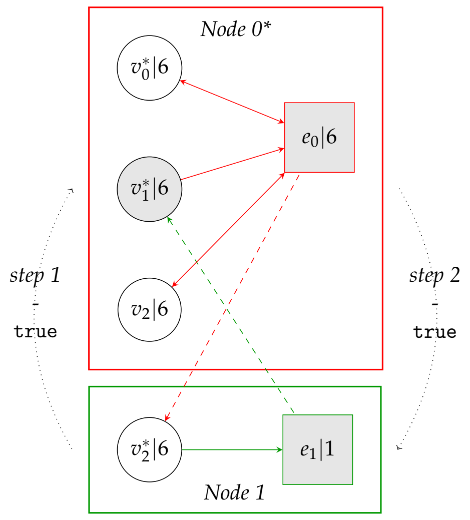
(a) *Phase 1:* The active replicas send their values to their local out-neighbors.



(b) *Phase 2:* The active uniques send their values to the master replicas of their out-neighbors.



(c) *Phase 3:* The active master replicas send their values to their servants.



(d) *Phase 4:* In *step 1*, the servant nodes send their activity to the master. In *step 2*, the master node sends the activity of the whole hypergraph to its servants.

Figure 4.1: Propagating the maximum hypervertex value. The circles represent replicas, the squares uniques. Gray partition elements are inactive. Dotted lines are messages. The master replicas and the master node are marked with an asterisk (*). The figure shows the elements' activities and statuses at the beginning of each phase.

4.2 The `main` Module

The `main` module defines the command line interface (c.f. Section 4.2.1) with its two subcommands `partition` (see Section 4.2.2) and `node` (see Section 4.2.3). It also contains the program's entry point, the `main` function. This function first parses the command line arguments. Depending on the provided subcommand, it then either directs the `partitioner` module to partition a given hypergraph or instructs the `algorithm` module to run a chosen algorithm.

4.2.1 The Command Line Interface

Our command line interface expects the arguments to be in the following order:

```
<subcommand> <positional arguments> [<optional arguments>]
```

The subcommand argument and the positional arguments are identified by their position and must always be provided. The optional arguments are optional. Thus, they cannot be found by their position and must be preceded by their name. The names are started with double hyphens to distinguish them from the argument parameters, like `--<command-name>`.

4.2.2 The `partition` Subcommand

The `partition` subcommand has the following parameters:

- `HYPERGRAPH`, the path to the file containing the hypergraph. The expected file format is described in Section 4.3.1.
- `PARTITIONS`, the number of partitions to partition the hypergraph into.
- `--out`, the directory to which the partitions are written to. Defaults to the current working directory.
- `--partition-hyperedges`, a flag indicating that the set of hyperedges should be partitioned. In this case, the hyperedges are uniques and the hypervertices replicas. The partitioner partitions by the set of hypervertices if this option is omitted.
- `--partitioning`, containing the path to a directory with a custom partitioning. For every partition *i*, the directory must have a file `partition-<i>.txt` which contains the ids of the uniques that should be assigned to that partition.

4.2.3 The `node` Subcommand

The `node` subcommand has the following parameters:

- `ALGORITHM`, the name of the algorithm the user wants to run.
- `NODE_ID`, the node's id.
- `PARTITION`, the directory to the partition that is assigned to the node.
- `ADDRESSES`, the path to a text file containing the IP addresses and ports the network nodes are operating on. The *i*th line contains the address and port of node $i - 1$. The IP addresses are in the format `<ip address>:<port>`.
- `--superstep-limit`, the maximal number of supersteps the algorithm should run. Defaults to no limit.
- `--source`, the id of the source hypervertex for algorithms that start at a source hypervertex.

4.3 The `partitioner` Module

The `partitioner` module takes a hypergraph and partitions it. In this section, we outline the input format of the hypergraph (see Section 4.3.1), specify the `partitioner`'s output (in Section 4.3.2) and explain how the partitioner works (c.f. Section 4.3.3).

4.3.1 The Input Format

The user must provide the hypergraph in a file with a syntax as defined in the Hygra project [Shu20]. It only supports directed hypergraphs, undirected hypergraphs must be encoded as directed hypergraphs. The hypergraph is given as a bipartite graph. The format looks as follows:

```
AdjacencyHypergraph|WeightedAdjacencyHypergraph
<nv>
<mv>
<nh>
<mh>
<ov0>
...
<ov(nv-1)>
<ev0>
...
<ev(mv-1)>
[<wev0>
...
<wev(mv-1)>]
<oh0>
...
<oh(nh-1)>
<eh0>
...
<eh(mh-1)>
[<weh0>
...
<weh(mh-1)>]
```

The header. The first line contains `AdjacencyHypergraph` for unweighted and `Weighted-AdjacencyHypergraph` for weighted hypergraphs. `<nv>` and `<nh>` are the number of hypervertices and hyperedges, respectively. `<mv>` and `<mh>` is the sum of the out degrees of all hypervertices and hyperedges, respectively.

The hypervertices. `<ovi>` gives the line offset for the beginning of hypervertex *i*'s out-neighborhood. `ev0,...,ev(mv-1)` contain the ids of the hypervertices' out-neighbors. The line offsets are regarding to the line containing `<ev0>`. Thus, if `<ev0>` is in line l, then hypervertex *i*'s out-neighborhood starts at line $l +$ `<ovi>`. For $i < nv - 1$, it ends at line $l +$ `<ov(i+1)>` $- 1$. Otherwise, it ends at the line containing `<ev<mv-1>>`.
If the hypergraph is weighed, then the hypervertices' out-neighborhoods are listed once again in `<wev0>,...,<wev(mh-1)>`. Only this time, the lines contain the weighs of the out-neighbors, not their ids. Hence, a weigh is not given once per hyperedge but every time the hyperedge is in a hypervertex's out-neighborhood.

The hyperedges. The encoding of the hyperedges is analogue to the encoding of the hypervertices. `<oh0>,...,<oh(nh-1)>` contain the line offsets of the hyperedges' out-neighborhoods, `<eh0>,...,<eh(mh-1)>` and `<weh0>,...,<weh(mh-1)>` the ids and weights, respectively, of the hyperedges' out-neighbors.

4.3.2 The Output Files

The `partitioner` writes each partition into an own directory. A directory consists of the following files:

- `meta.yaml`, a file with meta information about the graph.
- `replicas.bin`, containing the replicas and their out-neighbors.
- `uniques.bin`, consisting of the uniques and their out-neighbors.
- `replica_weights.csv`, containing the weighs of the replicas. Only if the hypergraph is weighed.
- `unique_weights.csv`, the unique pendant to the `replica_weights.csv`.

4.3.3 The `partition` Function

The `partition` module's `partition` function partitions the hypergraph. It has the following signature:

```
fn partition(hypergraph_filepath: &str, out_dir: &str, total_partitions:
    u8, uniques_as_hyperedges: bool, unique_partitions_dir: Option<&str>)
```

`hypergraph_filepath` contains the path to the hypergraph file, `out_dir` the path to which the partition directories are written to. `total_partitions` gives the number of partitions the hypergraph should be partitioned into, `uniques_as_hyperedges` indicates if the partitioning should be by the set of hyperedges or not. If the user provided a custom unique partitioning, then this is given in the directory behind the `unique_partitions_dir` path.

The `get_partition_id` closure. At the beginning, the partition function creates the `get_partition_id` closure. The closure takes the id of a unique and returns the id of the partition the unique belongs to. The assignment of uniques to partitions is either set by the directory provided by `unique_partitions_dirpath`, or by the modulo of the pseudorandomly hashed unique id and the number of partitions.

The partition directories and the `meta.yaml` file. For each partition *i*, we then add a directory `partition-<i>` in the directory given by `out_dir`. Afterwards, we insert a `meta.yaml` file into every partition directory. The file contains three parameters: The `uniques_as_hyperedges bool` and the total number of hypervertices and hyperedges from the hygra file.

The `replicas.bin` and `master_replicas.bin` files. Next, we import the offsets of the replica hypergraph elements from the hygra file. For each element, we then use the corresponding offset to load the ids of its out-neighborhood. We afterwards use the `get_partition_id` closure to determine the partitions that the uniques of the out-neighborhood are assigned to. These are the partitions on which the element must be replicated to. We randomly pick one of the partitions as the element's master partition. This is the partition the master replica is assigned to. A unique must know where the master replicas of its out-neighbors are. Hence, we assign a new id to the element that encodes this information. We choose the id so that the new id modulo the number of partitions equals to the element's master partition.
In Phase 3, the master replicas send their statuses to their servant replicas. They must know where to find them. For this reason, we map the id of the master replica to the partitions its servants are on. We then append this entry, encoded into bytes, to the `master_replicas.bin` file of the element's master partition. We ignore master replicas without servants.
Finally, we add the replica to the `replica.bin` files of the partitions the replica is on. To each file, we write the binary encoding of the new and old id, the size of the replica's complete out-neighborhood as well as the size and ids of its local out-neighborhood. We include the size of the local out-neighborhood so that the `partition_importer` knows where the encoding of the replica ends when it later imports the replicas.

The `uniques.bin` file. Next, we import the unique hypergraph elements and their out-neighborhoods analogue to the import of the replicas. However, we keep the elements' original ids and replace the out-neighbors' old replica ids with their new ones. In the last step, we write the id of the unique hypergraph element as well as the size and ids of its out-neighborhood to the `unique.bin` file of the partition the element is on.

The weigh files. If the hypergraph is weighed, we must still import the weighs. If the replicas represent hypervertices, then `<whi>` contains the weigh of the replica with id `<ehi>`. If they represent hyperedges, then `<wvi>` holds the weigh of replica `<evi>`. We use this fact to import the weighs of the replicas and write them to the `replica.csv` files of the partitions the replica is on. Each line contains the id of a replica and its weigh as a string. The parsing depends on the algorithm and is therefore done when we start a node.
The creation of the `unique_weighs.csv` file is analogue.

4.4 The `algorithms` Top Module

The `algorithms` top module defines and implements the `ToFromBytes` trait that is used to en- and decode Phase 2 messages and Phase 3 statuses (see Section 4.4.1). Furthermore, it defines the traits for the algorithm specific replica and unique implementations (in Section 4.4.2). It also implements the `run` function which is the function that the `main` module calls when the user runs the `node` subcommand. This function initializes some global variables, loads the algorithm specific replica and unique types and, if present, the algorithm specific `combine` function and finally tells the `hypergraph` module to run the algorithm (c.f. Section 4.4.3).

4.4.1 The `ToFromBytes` Trait

The `ToFromBytes` trait defines the encoding and decoding of messages and statuses sent in Phase 2 and 3, respectively. The messages and statuses must be `Send` and `'static` since they are sent across different threads. The implementations of the trait are also contained in the `algorithms` module. We do not specify them further since they are straightforward. The definition of the trait is given below. The constant `N` defines the number of bytes the values are converted to or from:

```
trait ToFromBytes<const N: usize> + Send + 'static {
    fn to_bytes(&self) -> [u8; N];
    fn from_bytes(bytes: [u8; N]) -> Self;
}
```

4.4.2 The `Replica` and `Unique` Traits

The `Replica` and `Unique` traits define the interface of the algorithm specific structs representing the replicas or uniques, respectively. Their instances are accessed in different threads concurrently. Hence, they must be `Send`, `Sync` and have a `'static` lifetime. `M1` and `M2` are generics for the message types of the messages sent in Phase 1 and 2, respectively. `S` is the type of the statuses sent in Phase 3:

```
trait Replica<const LM2: usize, const LS: usize, M1, M2: ToFromBytes<LM2>,
     S: ToFromBytes<LS>, F: Fn(u32, M1)>: Sync + Send + 'static {
    fn new(old_id: u32, id: u32, total_out_neighborhood_size: u32,
        local_out_neighbors: Vec<u32>, weigh: Option<&str>) -> (Self, bool
        );
    fn update(&self, send_msg: F, superstep: u16) -> bool;
    fn rcv_msg(&self, msg: M2) -> bool;
    fn get_status(&self) -> S;
    fn rcv_status(&self, status: S);
}

trait Unique<const LM2: usize, M1, M2: ToFromBytes<LM2>, F: Fn(u32, M2)>:
     Sync + Send + 'static {
    fn new(id: u32, out_neighbors: Vec<u32>, weigh: Option<&str>) -> (Self
        , bool);
    fn update(&self, send_msg: F, superstep: u16) -> bool;
    fn rcv_msg(&self, msg: M1) -> bool;
}
```

The constructors. The optional `weigh` parameters of the `Replica` and `Unique` constructor (the `new` methods) hold a weigh in weighed hypergraphs and are empty otherwise. In the `Replica` constructor, `old_id` and `id` give the old and new id of the hypergraph element the replica represents (see Section 4.3.3). The `old_id` is only used in the constructor to identify a replica by its original id from the hygra file. `local_out_neighbors` contains the ids of the replica's local out-neighborhood and `total_out_neighbors` the size of the replica's complete out-neighborhood. In the `Unique` constructor, `id` holds the unique's id and `out_neighbors` the ids of the unique's out-neighbors. Both constructors do not only return an instance of the struct they are implemented on but also the activity the instance should start with.

The instance methods. The `update` methods use the `send_msg` closure to send messages to their out-neighbors. Some algorithms need the current superstep, which is why it is provided in `superstep`.
The `rcv_msg` methods receive a message `msg` and return whether the instance should become active.
The `Replica` trait additionally defines the `get_status` and `rcv_status` methods to get or set the status of a master or servant replica, respectively.

4.4.3 The `run` Function

The signature of the `run` function is given below:

```
fn start(algorithm: Algorithm, partition_dir: &str,
    node_addresses_filepath: &str, node_id: u8, superstep_limit: Option<
    u16>, source_hypervertex: Option<u32>)
```

`algorithms` is an enum that specifies the algorithm the user wants to run. `partition_dir` is the path to the directory containing the node's partition. `node_addresses_filepath` is the path to the file with the node addresses. `node_id` is the node's id, `superstep_limit` the optional superstep limit and `source_hypervertex` an optional source hypervertex.

Initializing the global variables. In the `run` function, we first initialize four global variables that are needed by some of the algorithms: `SOURCE_VERTEX` which we set to `source_vertex`. Moreover `N` and `M`, the number of hypervertices and hyperedges, respectively, and `UNIQUES_AS_HYPEREDGES`, which tells if the partitioning was by the set of hyperedges or not. These three parameters are all imported from the partition directory's `meta.yaml` file (see Section 4.3.3).

Calling the `hypergraph` module's `run` function. In the last step, we instruct the `hypergraph` module's generic `run` function to process the given hypergraph. When doing so, we must specify the concrete types the generics should be replaced with. The reason is that, in this case, the compiler cannot derive them from the function's signature. The generic types are all algorithm specific which is why the `hypergraph` module's `run` function is called in the `algorithms` module. They are: `M1`, `M2` for the Phase 1 and 2 messages, respectively. Furthermore, `S` for the Phase 3 statuses and `R` and `U` for the concrete `Replica` and `Unique` types. We will discuss more about `hypergraph`'s `run` function in Section 4.8.8.

4.5 The Implemented Algorithms

The algorithm specific code for each algorithm is implemented in a submodule of the `algorithms` module. In this section, we describe the implemented algorithms. The algorithms are ConnectedComponent (c.f. Section 4.5.1), PageRank (see Section 4.5.2), SingleSourceShortestPath (in Section 4.5.3) and LabelPropagation (c.f. Section. 4.5.4). We furthermore provide pseudocode for the algorithm's replicas (see Algorithm 3-6). We neither show the pseudocode for the uniques nor the `combine` functions. Their implementations become obvious with the descriptions of the algorithms and the pseudocode of the replicas.

4.5.1 ConnectedComponent

In a hypergraph, a connected component is a set of hypervertices where every hypervertex has a path to every other hypervertex. ConnectedComponent is only defined for undirected hypergraphs [Shu20].

The implementation. Our implementation is similar to finding the maximum value in a hypergraph (c.f. Section 2.2.4): All replicas and uniques store their local or complete out-neighbors, respectively, as well as a *value* field with the highest unique id they have seen so far. For the replicas, this is initially the highest id among their out-neighbors. For the uniques, it is initially their own id. All elements start as active. In the `update` method of a replica or unique, the replica or unique broadcasts its *value* to its local or complete out-neighbors, respectively. Afterwards, it deactivates itself. It becomes active as soon as its `rcv_msg` method receives a value higher than its current one. In this case, it also updates to the higher value. A replica also becomes active when its `rcv_status` method is called. The replicas' `get_status` and `rcv_status` methods are simple getters and setters, respectively, for *value*. The algorithm's `combine` function takes two messages and returns the higher one. The pseudocode for the replicas is given in Algorithm 3.

4.5.2 PageRank

PageRank is originally a graph algorithm which assigns an importance to every vertex in a graph [BP98]. A message based approach could work as follows: All vertices begin with the same importance. They are active during the complete run of the algorithm. The messages contain importances. In each superstep, the `update` method of all vertices is executed. In the first superstep, the method already knows the vertex's current importance. In the other supersteps, it calculates the importance by summing

Algorithm 3 The ConnectedComponentReplica

```
struct ConnectedComponentReplica
    value,
    local_out_neighbors
end struct

Replica implementation for ConnectedComponentReplica
    ▷ Returns a tuple of a new ConnectedComponentReplica and its initial activity.
    method new(_, _, local_out_neighbors, _)
        return (ConnectedComponentReplica{
            max(local_out_neighbors),
            local_out_neighbors},
            true)
    end method

    method update(self, send_msg, _)
        for id in self.local_out_neighbors do
            send_msg(id, value)
        end for
        return false
    end method

    method rcv_msg(self, msg)
        if $self.value < msg$ then
            $self.value = msg$
            return true
        end if
        return false
    end method

    ▷We skip showing the implementation of get_status and rcv_status since they
    are just a getter or setter, respectively, for value.
end implementation
```

up the received messages. In all supersteps, the method then takes d times its own importance and adds it with $1 - d$ times the average vertex importance. d is called the damping factor and usually set to 0.85 [BP98]. Afterwards, the method distributes the new importance equally among its out-neighbors. The algorithm either runs until convergence or for a predefined number of iterations. It has a `combine` function which takes two messages and returns the sum of the messages [Shu20].

The implementation. We can easily adapt PageRank to hypergraphs: All replicas and uniques store their local or complete out-neighbors, respectively. In our framework, the partition elements cannot access the received messages in their `update` methods. Instead, they have an *importance* field which stores the sum of the received importances. A replica also holds the size of its complete out-neighborhood. It needs this to calculate the importance that it has to send to its local out-neighbors. At the beginning, all hypervertices start with the same importance, the hyperedges with no importance. All partition elements are active during the complete run of the algorithm. In their `update` methods, the replicas and uniques first check whether they are hypervertices or not. They do so with the global `HYPEREDGES_AS_UNIQUES` variable (see Section 4.4.3). If they are hypervertices, then they spread their importance in the same way as the vertex in the graph version. Otherwise, they ignore the damping factor and directly distribute their importance among their out-neighbors [Shu20]. The `rcv_msg` method adds the received importance to *importance*. The replicas' `get_status` and `rcv_status` methods are simple getters and setters, respectively, for *importance*. The `combine` function is identical to its pendant in the algorithm's graph version. The pseudocode for the replicas is given in Algorithm 4.

4.5.3 SingleSourceShortestPath

The SingleSourceShortestPath (SSSP) algorithm takes a weighed hypergraph and a source hypervertex s and calculates the distance from s to any other hypervertex [Shu20].

The implementation. In our implementation, all replicas and uniques store their local or complete out-neighbors, respectively. They also hold a *distance* and a *length* field. For a hypervertex, the distance stores the shortest currently known distance from s to the hypervertex. For a hyperedge, it stores the smallest distance the hyperedge has received so far from its in-neighbors. The *length* of a hypervertex is 0, the *length* of a hyperedge is its weigh.
At the beginning, s is active with distance 0. All other partition elements are inactive with distance ∞. In the `update` method of a replica or unique, the partition element

Algorithm 4 The PageRankReplica

```
struct PageRankReplica
    $importance,$
    $local\_out\_neighbors,$
    $total\_out\_neighborhood\_size$
end struct

Replica implementation for PageRankReplica
    method new(_, _, $local\_out\_neighbors, total\_out\_neighborhood\_size$)
        ▷uniques_as_hyperedges is a global variable (see Section 4.4.3)
        if uniques_as_hyperedges then
            $importance = 1$
        else
            $importance = 0$
        end if
        return (PageRankReplica{
            $importance,$
            $local\_out\_neighbors,$
            $total\_out\_neighborhood\_size\}$,
            true)
    end method

    method update($self$, send_msg, _)
        ▷ We assume $total\_out\_neighborhood\_size$ to never be 0
        if uniques_as_hyperedges then
            ▷ N is a global variable
            $msg = 0.15/N + (0.85 \cdot self.importance)/self.total\_out\_neighborhood\_size$
        else
            $msg = self.importance/self.total\_out\_neighborhood\_size$
        end if
        for $id$ in $local\_out\_neighbors$ do
                send_msg($id$,$msg$)
        end for
        ▷ Resets $importance$ for the next step
        $self.importance = 0$
        return true
    end method
```

```
  method REPLICA_RCV_MSG($self$, $msg$)
    $self.importance += msg$
    return true
  end method

  ▷ We omit GET_STATUS and RCV_STATUS since they are just a getter and setter,
respectively, for $importance$.
end implementation
```

broadcasts the sum of its *distance* and *length* to its local or complete out-neighborhood, respectively. Afterwards, it deactivates itself. It becomes active again when its `rcv_msg` method receives a distance smaller than its current one. In this case, it updates *distance* to the received distance. A replica also becomes active when its `rcv_status` method is called. The replicas' `get_status` and `rcv_status` methods are simple getters and setters, respectively, for *distance*. The `combine` function of the algorithm takes two distances and returns the smaller one. The algorithm terminates when all elements are inactive. A pseudocode version of the replicas is given in Algorithm 5.

4.5.4 LabelPropagation

The LabelPropagation algorithm finds communities in hypergraphs. A community is a set of hypervertices whose members are closely connected to each other but sparsely connected to the outside of the community. The idea of the algorithm is that every hypervertex starts with its own label. In each superstep, every hypervertex then updates its label to the label most frequent among its hypervertex in-neighbors. With hypervertex in-neighbors, we mean the hypervertices that have an hyperedge to the updating hypervertex. Frequency ties are broken randomly. Usually, the labels converge. If they do, they represent the community the hypervertices are in [RAK07].

The implementation. We have implemented the algorithm as follows: The replicas and uniques store their local or complete out-neighbors, respectively. Every partition element additionally holds a map that contains which labels it has received how often since the last call to its `update` method. The initial map of a hypervertex contains the hypervertex's id with frequency one. The initial map of a hyperedge is empty. All partition elements are active during the complete run of the algorithm. In their `update` methods, the partition elements do nothing if their map is empty. Otherwise, they get the map's most frequent label and send it to their out-neighbors. At the end of the method, they clear the map. If the `rcv_msg` method receives a label that is not in

Algorithm 5 The SingleSourceShortestPathReplica

```
struct SSSPReplica
    distance,
    length,
    local_out_neighbors
end struct

Replica implementation for SSSPReplica
    method new(old_id, _, local_out_neighbors, weigh)
        if uniques_as_hyperedges then
            $length = 0$
        else
            $length = weigh$
        end if
        ▷ Like uniques_as_hyperedges, source_hypervertex is a global variable
        if uniques_as_hyperedges and $old\_id =$ source_hypervertex then
            $distance = 0$, $active =$ true
        else
            $distance = \infty$, $active =$ false
        end if
        return (SSSPReplica {
            distance,
            length,
            local_out_neighbors},
            active)
    end method

    method update(self, send_msg, _)
        for id in self.local_out_neighbors do
            send_msg(id, $self.distance + self.length$)
        end for
        return false
    end method

    method rcv_msg(self, msg)
        if $msg < self.distance$ then
            $self.distance = msg$
            return true
        end if
        return false
    end method
```

```
  ▷ We omit GET_STATUS and RCV_STATUS since they are just a getter and setter, respectively, for $distance$.
  end implementation
```

the map, it adds the label to the map with frequency one. Otherwise, it increments the label's frequency by one. The replicas' `get_status` method returns the label with the highest frequency from the map. When a servant replica receives a label with the `rcv_status` method, then the servant adds the label to the map with frequency 1. The map is empty otherwise since servant replicas never receive messages. Hence, in Phase 1, the servants will send the same label as their masters. This is a small work-around to avoid that replicas need an extra field to store their status explicitly.
The algorithm stops as soon as a predefined stop criteria like a certain number of iterations is met [RAK07]. LabelPropagation does not have a `combine` function. Algorithm 6 shows the pseudocode for the replicas.

Algorithm 6 The LabelPropagation replica

```
struct LabelPropagationReplica
    $label\_to\_frequency$,
    $local\_out\_neighbors$
end struct

Replica implementation for LabelPropagationReplica
    method NEW(_, $id$, $local\_out\_neighbors$, _)
        ▷ UNIQUES_AS_HYPEREDGES is a global variable
        if UNIQUES_AS_HYPEREDGES then
            ▷ initializes an empty map
            $label\_to\_frequency$ =MAP(())
        else
            ▷ Creates a map with the initial entry "$id$ to 1"
            $label\_to\_frequency$: MAP(($id$,1)),
        return (LabelPropagationReplica {
            $label\_to\_frequency$
            $local\_out\_neighbors$},
            true)
    end method
```

```
  method UPDATE($self$, SEND_MSG, _)
    ▷ Gets the key of the entry with the highest value
    label = KEY_BY_MAX_VALUE($self.label\_to\_frequency$)
    for $id$ in $self.local\_out\_neighbors$ do
      SEND_MSG($id$, $label$)
    end for
    ▷ Clears the map
    CLEAR($self.label\_to\_frequency$)
    return true
  end method

  method RCV_MSG($self$, $msg$)
    ▷ Increments the frequency of the label contained in $msg$ by one. Creates a new
entry for the map with frequency one if the label is not yet in the map.
    INCREMENT($self.label\_to\_frequency$, $msg$)
    return true
  end method

  method GET_STATUS(self)
    return KEY_BY_MAX_VALUE($self.label\_to\_frequency$)
  end method

  method RCV_STATUS($self$, $status$)
    INCREMENT($self.label\_to\_frequency$, $msg$)
    return true
  end method
end implementation
```

4.6 The `importer` Module

The `importer` module imports the node's partition with its `import` function. The function has the following signature:

```
fn import<const LM2: usize, const LS: usize, M1, M2: ToFromBytes<LM2>, S:
     ToFromBytes<LS>, R: Replica<LM2, LS, M1, M2, S>, U: Unique<LM2, M1,
    M2>>(partition_dir: &str) -> (HashMap<u32, (AtomicBool, R)>,
HashMap<u32, (AtomicBool, U)>, HashMap<u32, Vec<u8>>)
```

`partition_dir` holds the path to the partition directory. The function returns the maps `replicas`, `uniques` and `master_replicas`. `replicas` maps the id of every local replica to a tuple of its activity and the struct that represents the replica. `uniques` does the same for the local uniques. `master_replicas` contains the local master replicas and the ids of the partitions their servants are on. The map omits master replicas without servants.

Importing the replicas. If the hypergraph is weighed, we first read the weighs from the `replica_weighs.csv` file into a map. Regardless of whether the hypergraph is weighed or not, we then import the node's replicas from the `replicas.bin` file. For each replica, we load its id, the ids of its local out-neighborhood and the total size of its out-neighborhood. We pass these arguments, if applicable together with the replica's weight string, to the `Replica` constructor. We then add the returned `Replica` instance and its initial activation status to the `replicas` map.

Importing the uniques. The import of the uniques is similar. If the hypergraph is weighed, we load the unique weighs from the `unique_weighs.csv` file into a map. In any case, we then read the uniques from the `uniques.bin` file. For each unique, we load its id and the ids of its out-neighborhood and pass these parameters, eventually with the unique's weigh, to the `Unique` constructor. We finally add the returned instance and its activation status to the `uniques` map.

4.7 The peer Module

The peer module manages the communication to the peer nodes with its Peer struct. We first introduce the Peer struct's fields (in Section 4.7.1) and its constructor (see Section 4.7.2). Next, we show how other modules tell a Peer instance to send Phase 2 messages and Phase 3 statuses to the peer it is responsible for (c.f. Section 4.7.3). Afterwards we describe the actual sending and receiving of the messages and statuses (in Section 4.7.4). We continue with methods to exchange a bool with other peers, which is needed in Phase 4 (see Section 4.7.5). Last but not least, we explain how the Peer instances are created and connected to their peers (in Section 4.7.6).

4.7.1 The Peer Struct

The Peer struct is defined as follows:

```
struct Peer<const LM2: usize, const LS: usize, M1, M2: ToFromBytes<LM2>,
    S: ToFromBytes<LS>> {
    sender: Mutex<BufWriter<OwnedWriteHalf>>,
    receiver: Mutex<BufReader<OwnedReadHalf>>,
    receive_task: Mutex<Option<JoinHandle<()>>>,
    send_task: Mutex<Option<JoinHandle<()>>>,
    complete_sending: AtomicBool,
    status_queue_sender: UnboundedSender<(u32, S)>,
    status_queue_receiver: Mutex<UnboundedReceiver<(u32, S)>>,
    combine: Option<fn(M2, M2) -> M2>,
    buf: Option<RwLock<DashMap<u32, M2>>>,
    msg_queue_sender: Option<UnboundedSender<(u32, M2)>>,
    msg_queue_receiver: Option<Mutex<UnboundedReceiver<(u32, M2)>>>,
}
```

We access a Peer instance in multiple threads concurrently. Thus, we can only have immutable references to it. We must therefore wrap members that we mutate in a Mutex, RwLock or AtomicBool. The struct members are:

- sender and receiver, the send and receive part of the TCP connection connected to the peer.
- send_task and receive_task, which store handles to tokio tasks that are responsible for sending and receiving to or from the peer, respectively.
- combine and buf, which are present for algorithms with a combine function. They are used to combine and buffer messages to the same replica. buf maps the id of

a replica to a combination of the messages that must still be sent to the replica.

- `msg_queue_sender` and `msg_queue_receiver`, which are the sender and receiver part of an `mpsc` channel. We use the channel in algorithms without a `combine` function as a thread-safe message queue to buffer Phase 2 messages before sending them.
- `status_queue_sender` and `status_queue_receiver`, which serve as a status queue for statuses akin to the message queue for messages.
- `complete_sending`, a flag which is set to `true` if all messages or statuses of the currently running Phase 2 or 3, respectively, have been sent to the `Peer` instance.

4.7.2 The `Peer` Struct's Constructor

The `Peer` constructor looks as follows:

```
fn new(tcp_stream: TcpStream, combine: Option<fn(M2, M2) -> M2>) -> Self
```

`tcp_stream` holds the connection to the peer the `Peer` instance is responsible for. `combine` contains the `combine` function, if the running algorithm has one.

Initializing a `Peer` instance. In the constructor, we split the `TcpStream` into a sender and receiver part for the `Peer`'s `sender` and `receiver` fields. The `send_task` and `receive_task` are initialized as empty, `complete_sending` as `false`. `combine` is passed to the `Peer`'s `combine` field. If `combine` is present, we initialize `buf` with an empty map, wrapped in a `RwLock`. Otherwise, we create a sender and receiver for a channel and put them into `msg_queue_sender` and `msg_queue_receiver`, respectively. The `status_queue_sender` and `status_queue_receiver` fields are always initialized with the sender and receiver part of an `mpsc` channel.

4.7.3 The `send_msg` and `send_status` Methods

A call to the `send_msg` and `send_status` method tells a `Peer` instance to send a Phase 2 message or a Phase 3 status, respectively, to a replica on the associated peer. The signatures of the two methods are similar:

```
fn send_msg(&self, replica_id: u32, msg: M)
fn send_status(&self, replica_id: u32, status: S)
```

`replica_id` is the id of the replica, `msg` a message and `status` a status.

Adding a message to `buf`. The `buf` map is used when the executed algorithm has a combine function. In this case, a call to send_msg checks whether buf contains a message for the given replica_id. If so, it updates the value of the buf entry by combining it with msg. Otherwise, it creates a new entry with replica_id as its key and msg as its value. In a DashMap, updating or inserting an entry just requires a read-only reference. Hence, we only need a read lock to buf and do not block other calls to the method.

Adding to the message and status queue. A call to send_msg for algorithms without a combine function and a call to send_status adds a message or status to the message or status queue, respectively. This is done by sending it to the appropriate channel via msg_queue_sender or status_queue_sender, respectively.

4.7.4 Sending and Receiving Messages and Statuses

The messages and statuses buffered in buf and the queues must somehow be sent to the peer associated to the Peer instance. At the same time, the instance must receive the messages and statuses sent from the peer. For this reason, Phase 2 and 3 each start a send and a receive tokio task. The tasks are started at the beginning of the corresponding phase and run until its end. Hence, the Phase 2 tasks never coexist with the Phase 3 tasks. They can therefore all be stored in the same send_task and receive_task fields. The sending and receiving to and from the peer is done with Peer's sender and receiver fields, respectively. Phase 2 starts its tasks with start_phase2, Phase 3 with start_phase3. Both phases end their tasks with end_phase. The signatures of the methods are shown below:

```
fn start_phase2(&self, replica_rcv: Arc<impl Fn(u32, M)
    + Send + Sync + 'static>)
fn start_phase3(&self, replica_rcv: Arc<impl Fn(u32, S)
    + Send + Sync + 'static>)
fn end_phase(&self)
```

The replica_rcv parameter in the start_phase2 and start_phase3 method references a closure that takes the id of a local replica and a message or status, respectively. The Phase 2 or Phase 3 receive task calls it to pass a received message or status, respectively, to the message's or status's local recipient. The closure passes the message or status to the replica using the replica's rcv_msg or rcv_status method.

Starting the tasks. start_phase2 and start_phase3 spawn the sending and receiving tasks for Phase 2 or 3, respectively, and store them in the send_task and receive_task fields.

Sending the `buf` map. If the executed algorithm has a `combine` function, messages are buffered in `buf` before they are sent. In this case, the send task started in `start_phase2` periodically checks whether `buf` has reached a certain size. If so, it acquires a write lock to `buf` and swaps the `buf` map with an empty map. This only locks the `buf` for a short time and thus minimizes any blocking of the `send_msg` method. The task then sends the messages of the full map to the peer. Regardless of the map size, it afterwards checks whether the `complete_sending` flag is set. If so, it sends the contents of the map, if not already done, as well as a special message to tell the peer that we have now sent all messages from Phase 2.

Sending the queues. Messages in algorithms without a `combine` function and statuses are added to the message or status queue, respectively, before they are sent. Therefore, the send task started in `start_phase2` for algorithms without a `combine` function and the send task started in `start_phase3` both continually empty the queue. They periodically remove the messages and statuses from the `msg_queue_receiver` or `status_queue_receiver`, respectively and send them to the peer. If the queue is empty, they check the `complete_sending` flag. If it is set, they send a special message to the peer to tell it that it has received all the messages or statuses from the current phase.

Receiving messages or statuses. The receive tasks in Phase 2 and 3 receive the messages or statuses, respectively, from the peer and forward them to the corresponding replica via the appropriate `replica_rcv` closure. The tasks terminate as soon as they receive the special message from the peer's send task.

The `end_phase` Method. Phase 2 and 3 call the `end_phase` method at their end. The method sets the `complete_sending` flag to `true` and waits for the sending and receiving tasks to finish.

4.7.5 Sending and Receiving `bool`s

In Phase 4, the peers exchange the activity of their partitions as `bool`s. They use the `send_bool` and `rcv_bool` methods to send and receive bools, respectively. The methods send a `bool` as a byte since `bool`s cannot be sent directly. Their signature is given below, their functionality is self explanatory:

```
fn send_bool(&self, bool_: bool)
fn rcv_bool(&self) -> bool
```

4.7.6 Establishing the Connections to the Peers and Creating the `Peer` instances

The `create_peers` function establishes the connections to the peers and creates the `Peer` instances that manage the connections. Its signature is given below:

```
fn create_peers<const LM2: usize, const LS: usize, M1, M2: ToFromBytes<
    LM2>, S: ToFromBytes<LS>>(node_addresses_filepath: &str, node_id: u8,
    combine: Option<fn(M2, M2) -> M2>) -> HashMap<u8, Arc<Peer<LM2, LS, M1
    , M2, S>>>
```

`node_addresses_filepath` holds the path to the file with the node addresses. `node_id` contains the id of the node and `combine` the `combine` function of the executed algorithm, if present.
The `create_peers` function parses the nodes' addresses and connects to every peer with an id lower than `node_id`. Next, it listens for connections from the peers with ids higher than `node_id`. The connections are managed by `TcpStream` instances. The function uses them to construct the `Peer` instances, together with the provided `combine` function. It finally returns the `Peer` instances as a map, with the node ids as keys.

4.8 The `hypergraph` Module

The `hypergraph` module contains the `Hypergraph` struct. The struct manages the local partition elements and runs the selected algorithm together with the peers.
In this section, we first explain the modes the struct uses to track the active partition elements (see Section 4.8.1). We then introduce the struct (c.f. Section 4.8.2) and its constructor (c.f. Section 4.8.3). Next, we explain its methods. We start with the methods that manage the delivery of messages and statuses to the local partition elements (in Section 4.8.4). We continue with the methods responsible for calling the elements' `update` methods (see Section 4.8.5). Furthermore, we explain the method that broadcasts the statuses of the local master replicas to their servants (c.f. Section 4.8.6). Next, we outline the method responsible for coordinating the complete run of the algorithm (see Section 4.8.7). At the very end, we describe how the `Hypergraph` instance is created and started (in Section 4.8.8).

4.8.1 Tracking Active Partition Elements

A straightforward way to call the `update` method of the active replicas is to iterate over all replicas and check their activity before calling it. However, this becomes too expensive when most replicas are inactive. In this case, the `Hypergraph` instance tracks the active replicas explicitly. As a result, Phase 1's running time only depends on the number of active replicas and not on the number of all replicas. However, tracking the active replicas explicitly has an overhead, which is why the `Hypergraph` instance uses the straightforward approach when a significant proportion of the replicas is active. The activity of the uniques is tracked in the same way.
We call the explicit tracking of the replicas the *replica sparse mode* and the implicit tracking the *replica dense mode*. Furthermore, we say that we operate in the *unique sparse mode* or *unique dense mode* if we track the uniques' activity explicitly or implicitly, respectively.

4.8.2 The `Hypergraph` Struct

The definition of the `Hypergraph` struct is given below:

```
struct Hypergraph<const LM2: usize, const LS: usize, M1, M2: ToFromBytes<
    LM2>, S: ToFromBytes<LS>, R: Replica<LM2, LS, M1, M2, S>, U: Unique<
    LM2, M1, M2>> {
    replicas: HashMap<u32, (AtomicBool, R)>,
    uniques: HashMap<u32, (AtomicBool, U)>,
    master_replicas: HashMap<u32, Vec<u8>>,
    active_replicas: Option<DashSet<u32>>,
    active_uniques: Option<DashSet<u32>>,
    peers: HashMap<u8, Arc<Peer<LM2, LS, M2, S>>>,
    node_id: u8,
}
```

We now outline the fields of the `Hypergraph` struct:

- `replicas` and `uniques` map the ids of every local replica or unique, respectively, to a tuple of the partition element's activity and the instance it represents.
- `active_replicas` and `active_uniques` track the ids of the active replicas or uniques when the struct operates in the replica or unique sparse mode, respectively.
- `master_replicas` maps the ids of the local master replicas to the node ids their servants are on. The map only holds master replicas that have servants.
- `node_id` holds the id of the current node.
- `peers` maps the node ids of the peers to the corresponding `Peer` instance.

4.8.3 The `Hypergraph` Struct's Constructor

The `Hypergraph` constructor takes all the fields of the `Hypergraph` struct as parameters, except for the `active_replicas` and `active_uniques` sets. It uses the parameters to initialize the corresponding `Hypergraph` fields. If less than five percent of all replicas or uniques are active, it initializes the `active_replicas` or `active_uniques` sets with the ids of the active replicas or uniques, respectively. Otherwise, it initializes them as empty.

4.8.4 Receiving Messages

Sending a message to a replica or unique may activate it. Sending a status to a servant replica always activates it. Thus, sending to a partition element requires to track its activity. This is done in the `replica_rcv_msg`, `unique_rcv_msg` and `replica_rcv_status` methods. They pass a message or status to its recipient and update the recipient's activity. Their signatures are shown below:

```
fn replica_rcv_msg(&self, id: u32, msg: M2);
fn unique_rcv_msg(&self, id: u32, msg: M1);
fn replica_rcv_status(&self, id: u32, status: S);
```

id is the recipient's id, msg a message and status a status.

In all these methods, we first use the id to get the corresponding element and its activity from the replicas or uniques map. We then call the element's rcv_msg or rcv_status method and update its activity. If we are in the appropriate sparse mode and need to activate the element, we still add its id to the active_replicas or active_uniques set, respectively.

4.8.5 Updating the Active Replicas and Uniques

The update_active_replicas and update_active_uniques methods update the active replicas or uniques, respectively:

```
fn update_active_replicas(&self, superstep: u16) -> u32;
fn update_active_uniques(&self, superstep: u16) -> u32;
```

superstep is the current superstep. The methods return the number of replicas or uniques that were active at the beginning of the method.

The update_active_replicas method. To update the replicas in the replica dense mode, we iterate over the replicas map and call the update method of the active replicas. In the sparse mode, we execute the update method of every replica whose id is present in the replica_active set.
When we call the update method of a replica, we pass Hypergraph's unique_rcv_msg method as the update method's rcv_msg parameter and superstep as its superstep parameter. The update method returns whether the replica should stay active or not. We use the returned bool to update the activity AtomicBool stored with the replica in the replicas map. In the sparse mode, we additionally update the active_replicas set. The execution of the replicas' update methods is parallelized.
At the end of the update_active_replicas method, we still return the number of replicas that were active at the beginning of the method. In the replica sparse mode, this is the initial size of the active_replicas set. We get the number in the dense mode by counting the initially active replicas when we iterate over the replicas map. We do not count them separately in an own iteration to avoid going over the map twice.

The update_active_uniques method. The update_active_uniques method is quite similar to the update_active_replicas method. However, it updates the uniques and therefore uses the uniques map and the active_uniques set whenever the update_active_replicas method uses the replicas map or active_replicas set. Furthermore, the rcv_msg closure passed to the update method is much more complex: It sends a message to a replica hypergraph element which may be replicated on different nodes. Hence, the closure first uses the element's id to identify its master partition (c.f. Section 4.3.3). If this is the partition of the local node, it calls `Hypergraph`'s `replica_rcv_msg` method. Otherwise, it sends the message to the master replica by calling the `send_msg` method of the appropriate `Peer` instance.

4.8.6 Broadcasting the Statuses

We use the `broadcast_statuses` method to send the statuses of the active master replicas to their servant replicas. To do so, we first access the active master replicas. In the replica sparse mode, we get them with the ids that are present in both, the `active_replicas` set and the `master_replicas` map. In the replica dense mode, we access them by checking the activity of the replicas whose ids are in the `master_replicas` map.
For each active master, we then get the node ids of the servants from the `master_replicas` map. Next, we send the status of the master replicas to the nodes their servants are on by using the `send_status` method of the corresponding `Peer` instances.

4.8.7 Executing the Algorithm

`Hypergraph`'s `run` method executes the algorithm. Its only parameter is `superstep_limit`, the optional superstep limit. In the run method, we repeat the four superstep phases until the hypergraph becomes inactive or we reach the superstep limit.

Phase 1. In Phase 1, we first update the active replicas with the `update_active_replica` method. The method returns the number of active replicas at the beginning of the phase. We use this number to determine whether we still track the active replicas in the right mode. This is not the case if less than five percent of the replicas are active and we are in the dense mode or when more then five percent are active and we are in the sparse mode. We switch the mode if it is wrong and we haven't changed it in the last five supersteps. The blocking period of five supersteps avoids switching the mode back and forth when the proportion of active replicas fluctuates around the five percent threshold. We switch from the sparse to the dense mode by emptying the `active_replicas` option and from the dense to the sparse mode by creating a set with the ids of the active replicas and assigning it to the `replicas_active` field.

Phase 2. In Phase 2, we first start the communication with the peers by calling the `start_phase2` method of the corresponding `Peer` instances. We then pass the

Hypergraph's replica_rcv_msg method as start_phase2's replica_rcv closure parameter. The Peer instances use this parameter to forward the messages they receive to the correct master replica.
In the next step, we update the active uniques with the update_active_uniques method. After that, we call the Peer instances' end_phase method. Eventually, we update the unique activity tracking mode in the same way as in Phase 1 for the replica activity tracking mode.

Phase 3. In Phase 3, we again start with setting up the communication with the peers. To do so, we call the Peer instances' start_phase3 method with Hypergraph's replica_rcv_status method as start_phase3's replica_rcv parameter. Thereafter, we broadcast the statuses of the active master replicas with the broadcast_statuses method and finally call the Peer instances' end_phase method.

Phase 4. In Phase 4, every node first checks whether it is still active. The servant nodes then send their activity to the master node. To do so, they use the send_bool method of the appropriate Peer instance. The master receives these values with the Peer instances' rcv_bool method. Next, it checks whether the hypergraph is still active and broadcasts the activity to its servants. In the last step, all nodes check if the hypergraph is inactive. If so, they terminate the run method.

4.8.8 Running the Algorithm

The run function runs the algorithm. Its signature is given below:

```
fn run<const LM2: usize, const LS: usize, M1, M2: ToFromBytes<LM2>, S:
    ToFromBytes<LS>, R: Replica<LM2, LS, M1, M2, S>, U: Unique<LM2, M1, M2
    >>(partition_dir: &str, node_addresses_filepath: &str, node_id: u8,
    superstep_limit: Option<u16>, combine: Option<fn(M2, M2) -> M2>) ->
    Hypergraph<LM2, LS, M1, M2, S, R, U>
```

node_addresses_filepath contains the path to the file with the node addresses, node_id the node's id, partition_dir the path to the directory of the local partition, superstep_limit the optional superstep limit and combine the optional combine function.

The run function first creates the peers, replicas, uniques and master_replicas parameters for the Hypergraph constructor. peers is created by the peer module's create_peers function, replicas, uniques and master_replicas by the importer module's import function. The run function uses these parameters, together with node_id, to construct a Hypergraph instance. It finally calls the Hypergraph instance's `run` function to execute the algorithm.

5 Evaluation

We now evaluate our program. First, we outline our experimental setup (c.f. Section 5.1) and the hypergraphs we used (see Section 5.2). Next, we introduce and analyse the partition strategies that we partitioned the hypergraphs with (in Section 5.3). For each partition strategy and algorithm, we then compare the average superstep running time (c.f. Section 5.4) and later the average phase running time on the tested hypergraphs (in Section 5.5). Afterwards, we take a short look at the memory consumption of our nodes (c.f. Section 5.6) before we finally compare the performance of our program with the performance of the frameworks mentioned in Chapter 3 (see Section 5.7).

5.1 Experimental Setup

The code was compiled with the `rustc` compiler `1.6.6` [rustc]. We conduct our experiments on 8 2-core nodes with 32 GB memory per node. PageRank and LabelPropagation never terminate on their own, hence we set the superstep limit to `3` when testing them. ConnectedComponent and SingleSourceShortestPath, on the other hand, terminate as soon as they found all connected components or shortest distances, respectively. Thus, we do not set a superstep limit for them.

5.2 The Test Hypergraphs

We evaluate Dyper with the hypergraphs *com-Orkut*, *Friendster* and *LiveJournal*. They are created from data of the social networks Orkut [LKc], Friendster [LKa], and LiveJournal [LKb], respectively. Orkut and Friendster were discontinued in 2014 and 2011, respectively, LiveJournal exists to this day [tea20; Raj21; LKb]. In all these networks, you can or could form friendships and join groups with other users of the platform [tea20; Raj21; LKb]. The hypervertices and hyperedges in the hypergraphs represent users and their group memberships, respectively [LKa; LKc; LKb]. All hypergraphs are undirected. We used the same randomly generated weighs as the Hygra paper for the SingleSourceShortestPath algorithm. The com-Orkut and Friendster hypergraphs are taken from the Stanford Network Analysis Project [LK14], the LiveJournal hypergraph from the Koblenz Network Collection [Kun].

Hypergraph	$\|V\|$	$\|E\|$	$\sum_{e\in E} deg(e)$	$\max_{v\in V} deg(v)$	$max_{e\in E} deg(e)$
com-Orkut	2.32×10^6	1.53×10^7	1.07×10^8	2958	9120
Friendster	7.94×10^6	1.62×10^6	2.35×10^7	1700	9299
LiveJournal	3.20×10^6	7.49×10^6	1.12×10^8	300	1.05×10^6

Table 5.1: Our test data sets. V and E are the sets of hypervertices and hyperedges, respectively.

An overview of some of their metrics is given in Table 5.1. Note that the size of a hypergraph does not only depend on the number of hypervertices and hyperedges but also on the total size of the hyperedges, which is given with $\sum_{e\in E} deg(e)$.

5.3 Partitioning

The partition strategies. We use two partitioner: The random partitioner we described in Section 4.3.3 and *Hype* [May+18], a partitioner that tries to minimize the edges between the partitions. Both partitioners partition the uniques in a perfectly balanced way. We partitioned the hypergraphs into 2, 4 and 8 partitions. For the random partitioner, we partitioned by the set of hypervertices and hyperedges. Hype, on the other hand, only allows to partition by the set of hypervertices. Unfortunately, we were not able to partition the Friendster hypergraph with Hype.

Metrics. In Table 5.2-5.9, we show some metrics of the partitions of the tested hypergraphs. We focus on metrics that will later help us to understand the performances of the partition strategies. For example, we often concentrate on the partition with the highest number in a certain category. The reason is that Dyper's phases never overlap. A new phase can only start when all nodes are done with the current one. As a consequence, the running time of a phase always depends on the slowest node. Since all our nodes have the same resources, this is usually the node with the most work to do.

Note, that the metrics only have a limited expressiveness since they do not consider the structure of the hypergraph, the activity of the hypergraph elements or the exact algorithm. For each number of partitions, the tables contain the following information:

1. The number of partitions.
2. The total number of replicas across all nodes.
3. The maximum number of replicas a node has.

4. The highest sum of unique out-edges any of the partitions has.
5. $\max_{p \in P} r_p$. Here, P is the set of partitions and r_p the number of *distinct* remote master replicas in the out-neighborhoods of the uniques in partition p.
6. The same as Column 5, except that a remote master replica is now counted every time it appears in a unique's out-neighborhood.
7. $\max_{p \in P} s_p$, where P is the set of partitions and s_p the number of servant replicas of partition p's local master replicas.

Partition overhead and balancing. Column 2 gives a good indicator of the replication overhead a partitioning produces.
Obviously, the overhead increases with the number of partitions. Furthermore, it is much higher for the random partitioner than for Hype. On the other hand, the random partitioner balances the replicas much better than Hype, which we see when comparing Column 3 and 4 in the Tables 5.2 and 5.4 or 5.7 and 5.9. We can compare Column 4 with the total number of unique out-neighbors across all partitions by taking a look at the second or third column in Table 5.1, depending on whether the partitioning is by the set of hypervertices or hyperedges, respectively. This column is equal to the total number of unique out-neighbors since every unique vertex is only represented by a single unique and every unique stores the complete out-neighborhood of the corresponding unique vertex (see Section 2.2.5).

Network traffic in Phase 2. In Phase 2, the active uniques send messages to the master replicas of their out-neighbors. If the algorithm has a `combine` function, then each node combines and buffers messages to the same master replica before sending them (see Section 4.7.3 and 4.7.4). However, it is still possible that a node sends several messages to the same remote master replica since the buffer that combines messages to the same remote master is usually emptied several times in Phase 2. Column 5 and 6 therefore give us some idea about the network traffic that we should expect in the busiest node for algorithms that have a `combine` function and where hypergraph elements are always active. Similar, Column 6 provides us with a sense of the network traffic of the busiest node for algorithms that have no `combine` function but where all hypergraph elements are always active.

Network traffic in Phase 3. In Phase 3, the active local master replica send their status to their servants. Hence, Column 7 is a good indicator for the network traffic of the busiest node for algorithms where all nodes are always active.

Note that, with our metrics, we can only make vague assumptions for algorithms that deactivate parts of the hypergraph.

Table 5.2: The *com-Orkut* partitions created by the *random* partitioner. Partitioned by the set of *hypervertices*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	25.31	12.66	53.58	5.01	23.89	5.01
4	34.31	8.59	26.81	4.76	17.74	4.75
8	42.56	5.34	13.45	3.41	10.25	3.42

Table 5.3: The *com-Orkut* partitions created by the *random* partitioner. Partitioned by the set of *hyperedges*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	4.41	2.20	53.58	1.04	26.68	1.04
4	8.01	2.01	26.91	1.42	20.06	1.42
8	13.77	1.73	13.49	1.44	11.70	1.44

Table 5.4: The *com-Orkut* partitions created by the *Hype* partitioner. Partitioned by the set of *hypervertices*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	19.34	14.53	84.97	2.02	25.73	2.02
4	26.64	11.49	49.04	4.19	25.41	4.19
8	33.29	7.93	25.23	4.07	16.91	4.07

Table 5.5: The *Friendster* partitions created by the *random* partitioner. Partitioned by the set of *hypervertices*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	2.84	1.42	11.74	0.61	5.71	0.61
4	4.33	1.08	5.88	0.68	4.21	0.68
8	6.15	0.77	2.94	0.57	2.44	0.57

Table 5.6: The *Friendster* partitions created by the *random* partitioner. Partitioned by the set of *hyperedges*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	11.18	5.60	11.79	1.62	4.29	1.62
4	14.45	3.65	5.97	1.64	3.15	1.64
8	17.30	2.19	3.00	1.19	1.80	1.19

Table 5.7: The *LiveJournal* partitions created by the *random* partitioner. Partitioned by the set of *hypervertices*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	9.14	4.58	56.21	0.83	26.49	0.83
4	11.09	2.78	28.12	0.90	20.24	0.90
8	13.39	1.68	14.06	0.74	11.69	0.74

Table 5.8: The *LiveJournal* partitions created by the *random* partitioner. Partitioned by the set of *hyperedges*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	6.12	3.08	57.87	1.46	28.76	1.46
4	11.28	2.87	29.23	2.05	21.77	2.05
8	19.57	2.64	15.54	2.19	13.41	2.19

Table 5.9: The *LiveJournal* partitions created by the *Hype* partitioner. Partitioned by the set of *hypervertices*. The maxima are per node. The numbers are in million (except for the first column).

Partitions	Replicas (Total)	Replicas (Max)	Unique Out-Neighbors (Max)	U. Out-Neighb. w. Rem. Mast. no duplicates (Max)	U. Out-Neighb. w. Rem. Mast. w. duplicates (Max)	Servants (Max)
2	7.96	6.68	92.59	?[1]	?[2]	?[3]
4	17.80	5.39	52.96	0.55	30.97	0.55
8	20.29	3.74	27.90	0.66	22.01	0.66

5.4 The Average Superstep Running Times

Figure 5.1, 5.2 and 5.3 show the average superstep running times for running the com-Orkut, Friendster and LiveJournal hypergraphs, respectively, on one, two, four and eight nodes. Each figure lists the running times for every tested combination of partition strategy and algorithm.

Comparing the different hypergraphs and algorithms. As expected, the running time increases with the size of the hypergraph. The fastest algorithm is SINGLE-

[3]After running the evaluations and before calculating these numbers our disk was deleted. Unfortunately, we were not able to partition the LiveJournal hypergraph with Hype into two partitions for a second time.

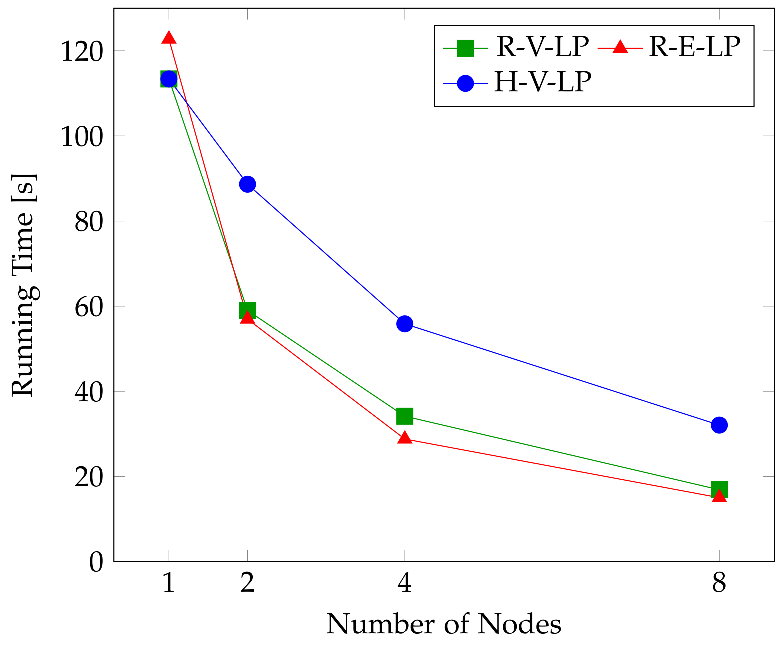
SourceShortestPath, followed by ConnectedComponent, PageRank and, with a huge distance, LabelPropagation. LabelPropagation is also the only algorithm without a `combine` function and the only algorithm where the status of a hypergraph element is a map instead of a single value (see Section 4.5.4). This is probably the reason why it is by far the slowest algorithm. The performance ranking of the remaining algorithms can most likely be explained by the different activities of their partition elements. In PageRank, the second slowest algorithm, all elements are always active (c.f. Section 4.5.2). ConnectedComponent and SingleSourceShortestPath, on the other hand, deactivate parts of their hypergraph (see Section 4.5.1 and 4.5.3). In our experiments, ConnectedComponent had more active elements in an average superstep, which explains why it is slower than SingleSourceShortestPath.

Comparing the different partition strategies. The superstep running times almost always decrease with the number of nodes. An exception is when we use the partitions created by the Hype partitioner. In this case, a single node is often faster than two nodes. Running on the partitions created by the Hype partitioner is generally more time-consuming than running on a random partitioning. The slow running times are probably due to Hype's unbalanced replica distribution (see Section 5.3). When we compare the random partitioning by the hypervertices with the random partitioning by the hyperedges, we see that the former is faster for the Friendster and LiveJournal and the latter for the com-Orkut hypergraph. In Table 5.1, we see that the hyperedges outnumber the hypervertices for Friendster and LiveJournal while the opposite happens for com-Orkut. This could have to do with the different performances between the partitioning by the hypervertices and the partitioning by the hyperedges. However, we cannot be sure from the metrics we measured.

5.5 The Average Phase Running Times

Figure 5.4-5.15 compare the average phase running times to the number of used nodes. Each figure focuses on a single hypergraph and algorithm and plots each tested partition strategy in an own chart. The charts omit Phase 4 since its running time is always negligible.

Comparing the different hypergraphs and algorithms. Obviously, the average phase times are slower for hypergraphs and algorithms with a slower average superstep running time. Asides from that, we did not gain any patterns when comparing the average phase times for the different hypergraphs and algorithms.



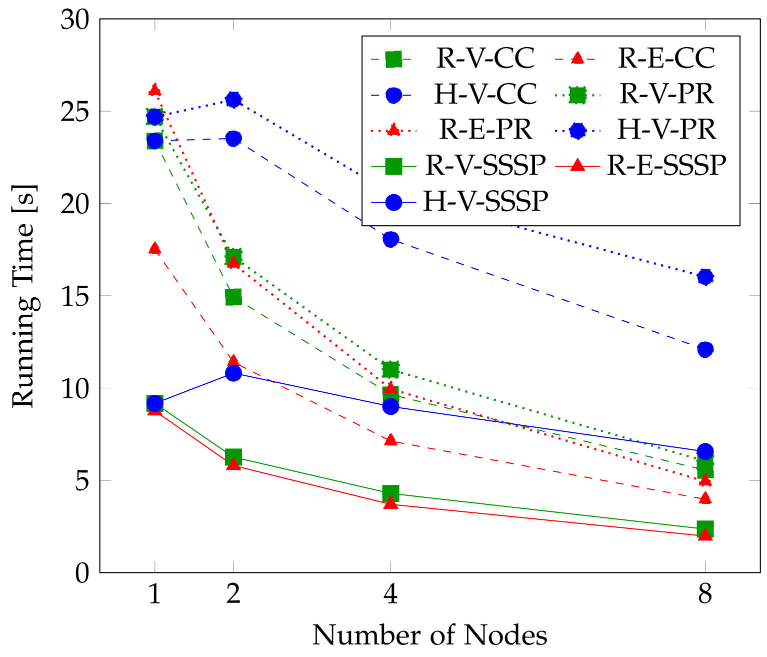
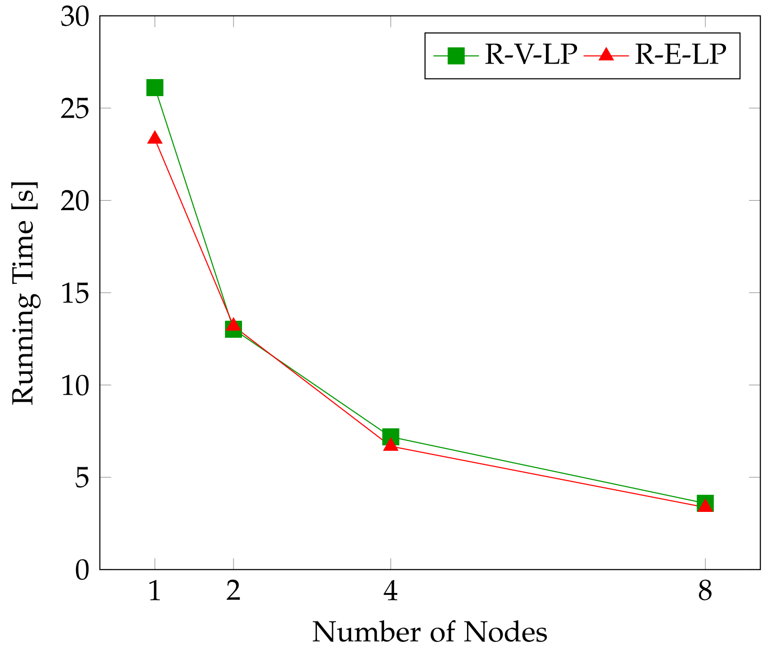


Figure 5.1: Average Superstep Running Time per Algorithm for the *com-Orkut* hypergraph. *R stands for the Random partitioner, H for the Hype partitioner, V for a partitioning by the Hypervertices, E for a partitioning by the Hyperedges.* CC *stands for* ConnectedComponent, LP *for* LabelPropagation, PR *for* PageRank *and* SSSP *for* SingleSourceShortestPath.



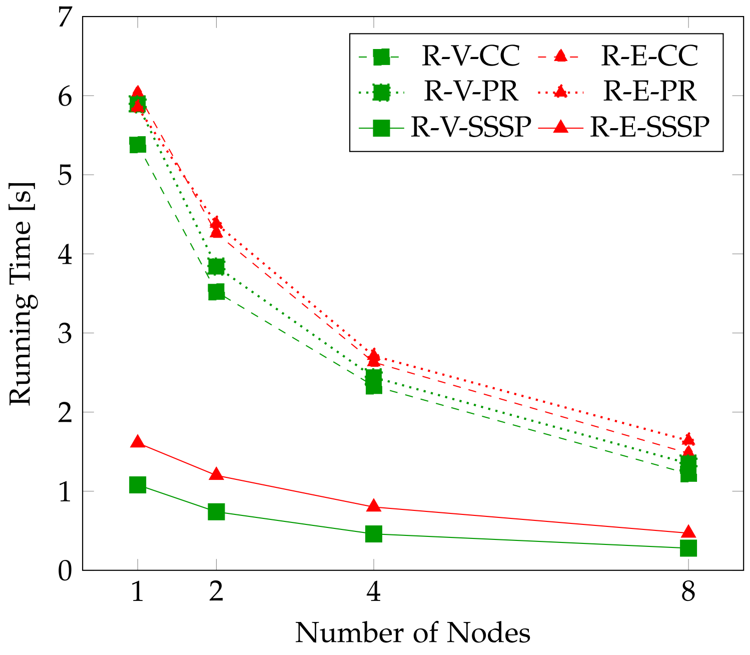
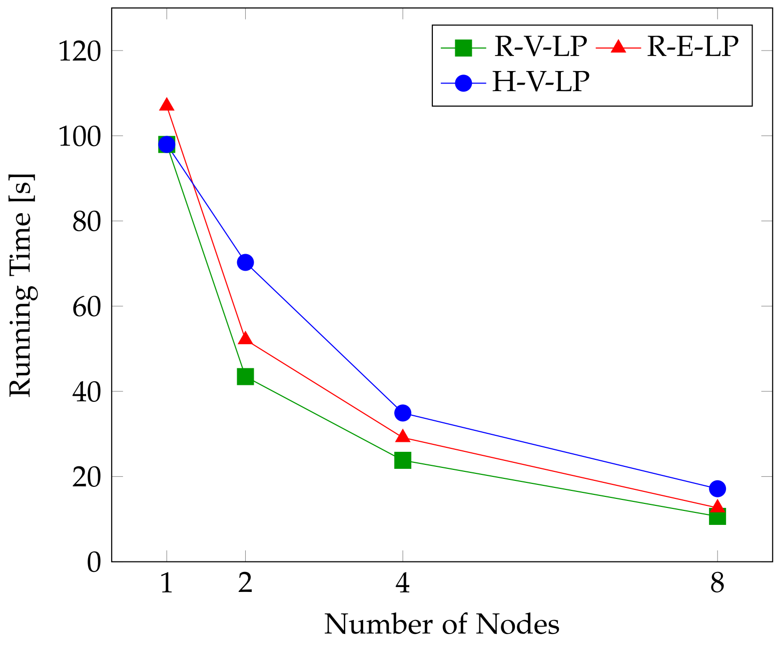


Figure 5.2: Average Superstep Running Time per Algorithm for the *Friendster* hypergraph. Unfortunately, we were not able to partition the Friendster hypergraph with Hype. *R stands for the Random partitioner, V for a partitioning by the Hypervertices, E for a partitioning by the Hyperedges.* CC *stands for* ConnectedComponent, LP *for* LabelPropagation, PR *for* PageRank *and* SSSP *for* SingleSourceShortestPath.



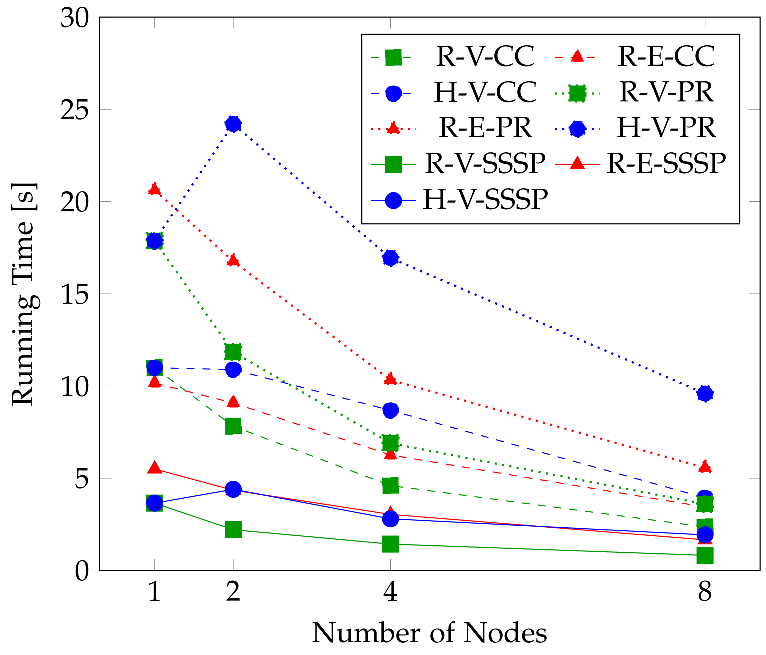


Figure 5.3: Average Superstep Running Time per Algorithm for the *LiveJournal* hypergraph. *R stands for the Random partitioner, H for the Hype partitioner, V for a partitioning by the Hypervertices, E for a partitioning by the Hyperedges.* CC *stands for* CONNECTEDCOMPONENT, LP *for* LABELPROPAGATION, PR *for* PAGERANK *and* SSSP *for* SINGLESOURCESHORTESTPATH.

Comparing the different phases. As presumed, Phase 2 always takes longer than Phase 1 when tested with more than one node. The reason is most likely that, for more than one node, Phase 1 only sends messages to local partition elements while Phase 2 sends messages to local and remote partition elements. Phase 3 is always skipped and therefore 0 when running an algorithm on a single node. When running on several nodes, then Phase 3's running times seem to be proportional to Column 7 in Table 5.2-5.9. Phase 3 is also mostly much faster than Phase 2.

Comparing the different partition strategies. We now compare the running times for the partitions created by the random partitioner when partitioning by the hypervertices with the running times for the Hype partitioner. We see that Hype is much slower in Phase 2. We believe that this is a result of the huge size of the uniques' out-neighborhoods in some of the partitions (see Column 4 in Table 5.4 and 5.9).
We could not find a pattern when comparing other partition strategies.

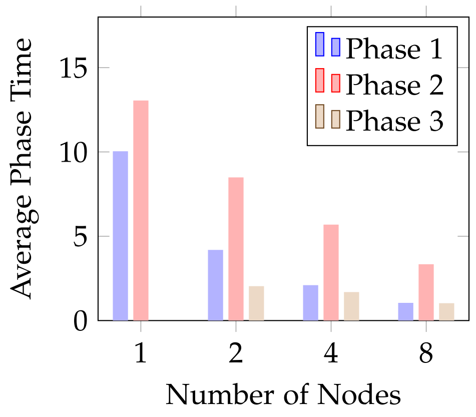
5.6 Memory Usage

Figure 5.16, 5.17 and 5.18 list the memory usage for running the algorithms on the com-Orkut, Friendster and LiveJournal hypergraph. Each figure provides one plot for every tested combination of algorithm and partition strategy. Each plot displays the maximal memory usage among the nodes when running a test with one, two, four or eight nodes.

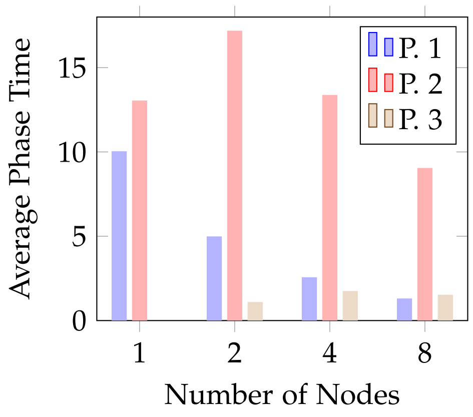
Comparing the different hypergraphs. As expected, the maximal memory usage grows with the size of the processed hypergraphs. It is between 0.5 and 8.1 GB for the smallest hypergraph, Friendster, and between 0.9 and 18 GB for the biggest hypergraph, com-Orkut. As presumed, increasing the number of nodes decreases the maximal amount of memory needed for a node.

Comparing the different algorithms. The most memory intensive algorithm by far is LabelPropagation. This is no surprise, since the status of the partition elements is a map and not a single value, as it is for the other algorithms (see Section 4.5). After LabelPropagation, the frontier based SingleSourceShortestPath algorithm is often the second most resource intensive one, probably because it tracks the active partition elements explicitly (see Section 4.8.1).

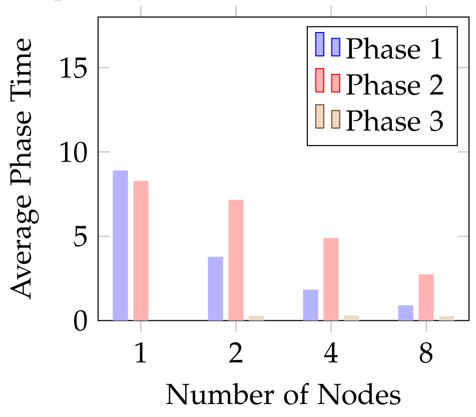
Comparing the different partition strategies. For more than one node, the Hype partitions always require the most memory. Again, this is probably due to Hype's unbalanced distribution of the replicas (see Section 5.3). We cannot see a clear pattern



(a) Partitioned by the *hypervertices* with the *random* partitioner.

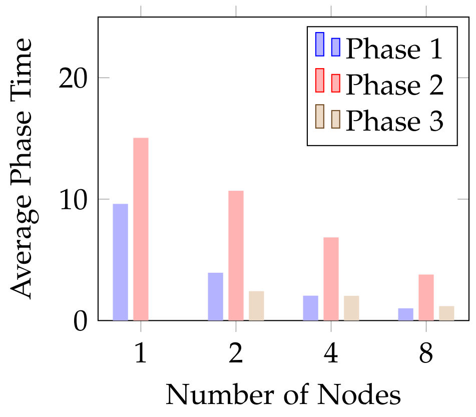


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

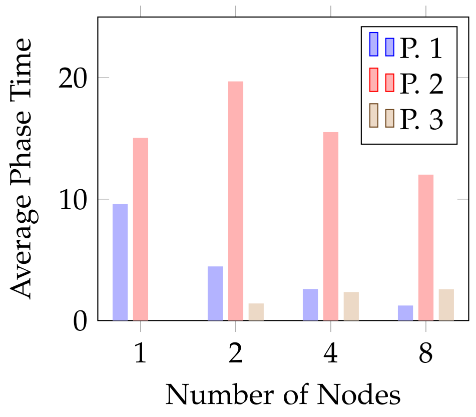


(c) Partitioned by the *hyperedges* with the *random* partitioner.

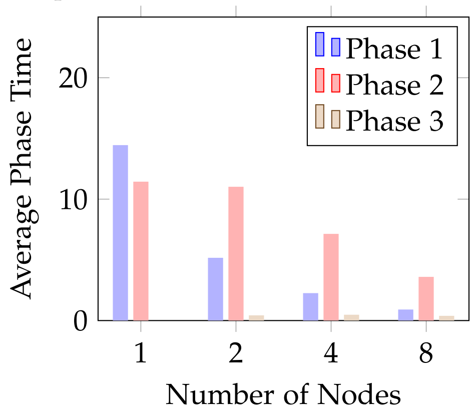
Figure 5.4: Average Phase Running Time for CONNECTEDCOMPONENT on *com-Orkut*. Phase 4 is negligible and therefore omitted. *P. stands for Phase.*



(a) Partitioned by the *hypervertices* with the *random* partitioner.

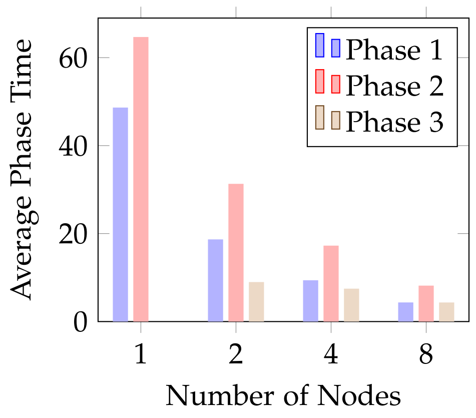


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

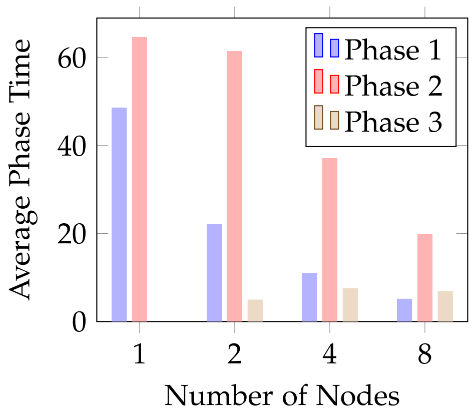


(c) Partitioned by the *hyperedges* with the *random* partitioner.

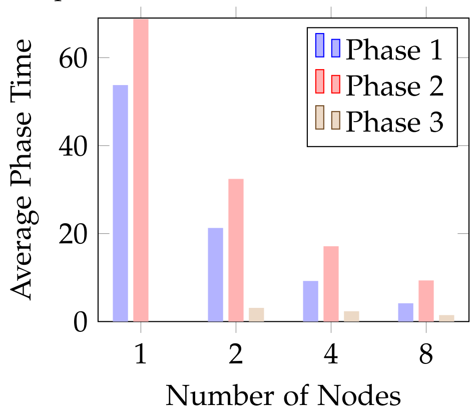
Figure 5.5: Average Phase Running Time for PAGERANK on *com-Orkut*. Phase 4 is negligible and therefore omitted. *P. stands for Phase.*



(a) Partitioned by the *hypervertices* with the *random* partitioner.

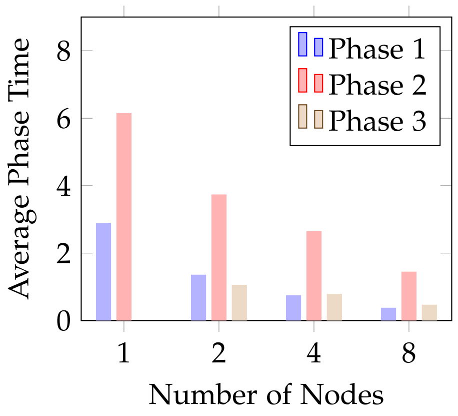


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

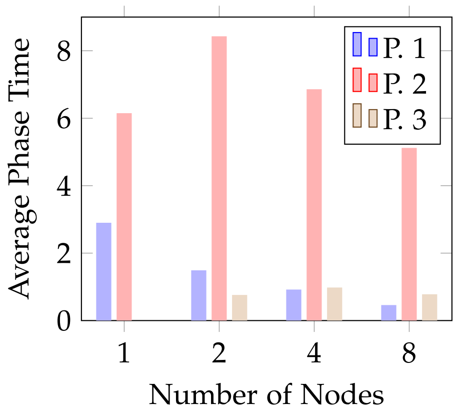


(c) Partitioned by the *hyperedges* with the *random* partitioner.

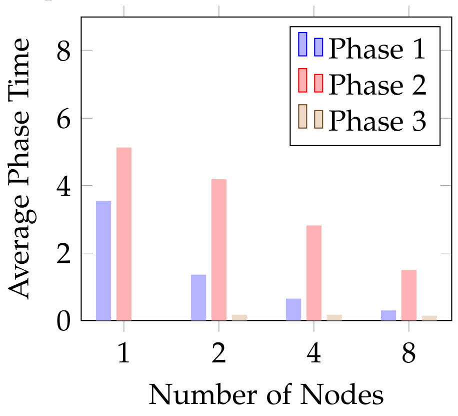
Figure 5.6: Average Phase Running Time for LabelPropagation on *com-Orkut*. Phase 4 is negligible and therefore omitted.



(a) Partitioned by the *hypervertices* with the *random* partitioner.

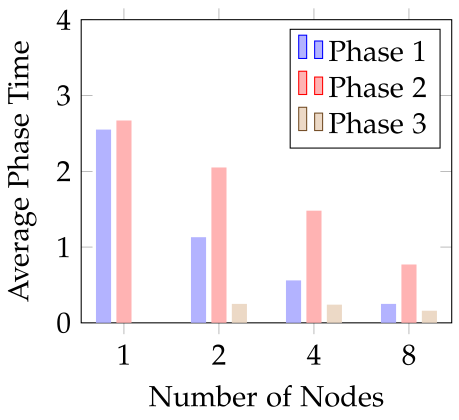


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

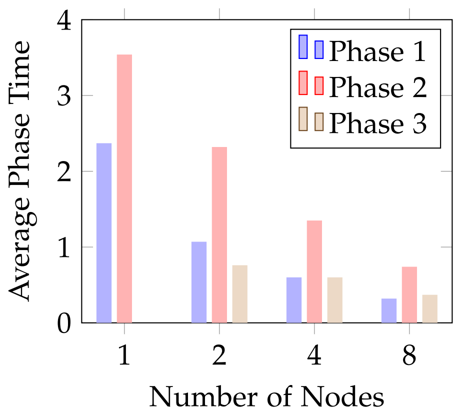


(c) Partitioned by the *hyperedges* with the *random* partitioner.

Figure 5.7: Average Phase Running Time for SingleSourceShortestPath on *com-Orkut*. Phase 4 is negligible and therefore omitted. *P. stands for Phase.*

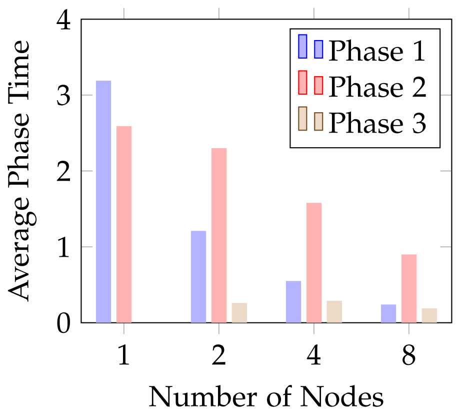


(a) Partitioned by the *hypervertices* with the *random* partitioner.

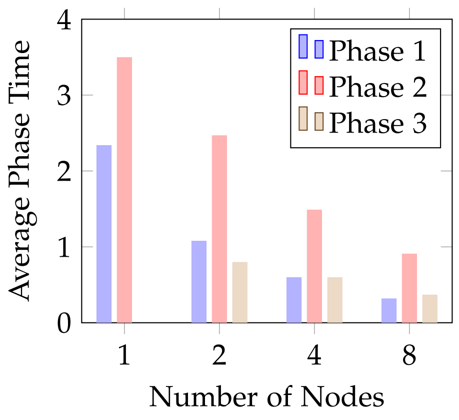


(b) Partitioned by the *hyperedges* with the *random* partitioner.

Figure 5.8: Average Phase Running Time for ConnectedComponent on *Friendster*. Phase 4 is negligible and therefore omitted.

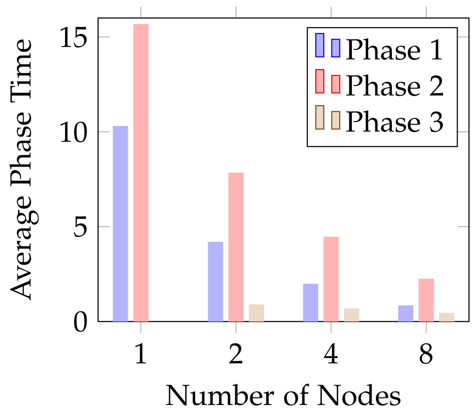


(a) Partitioned by the *hypervertices* with the *random* partitioner.

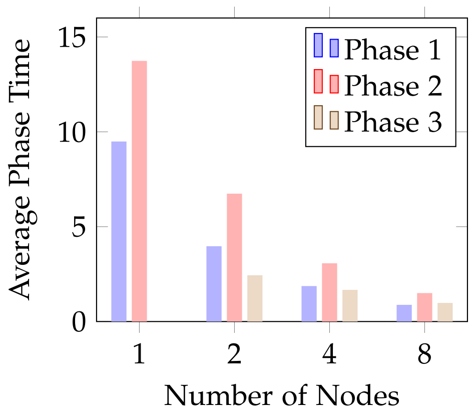


(b) Partitioned by the *hyperedges* with the *random* partitioner.

Figure 5.9: Average Phase Running Time for PageRank on *Friendster*. Phase 4 is negligible and therefore omitted.

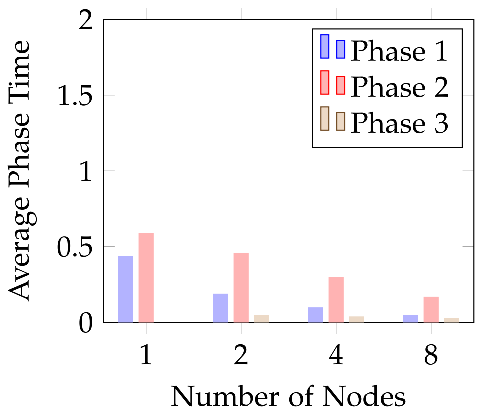


(a) Partitioned by the *hypervertices* with the *random* partitioner.

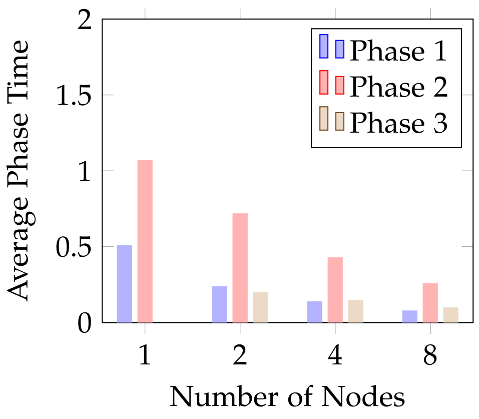


(b) Partitioned by the *hyperedges* with the *random* partitioner.

Figure 5.10: Average Phase Running Time for LabelPropagation on *Friendster*. Phase 4 is negligible and therefore omitted.

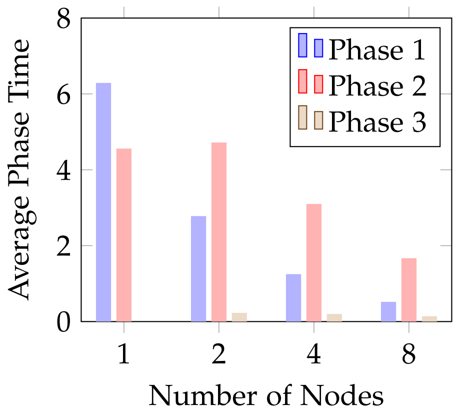


(a) Partitioned by the *hypervertices* with the *random* partitioner.

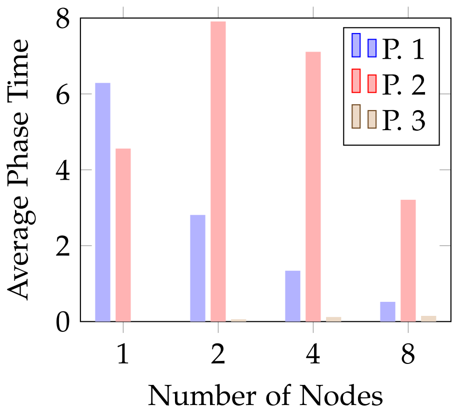


(b) Partitioned by the *hyperedges* with the *random* partitioner.

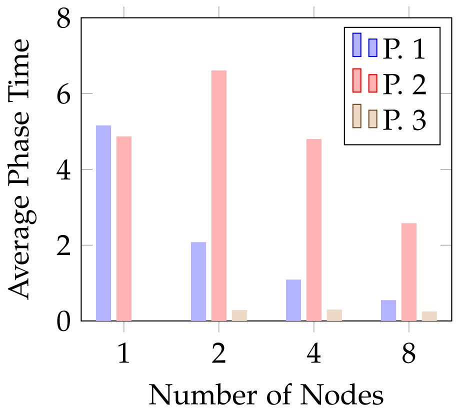
Figure 5.11: Average Phase Running Time for SingleSourceShortestPath on *Friendster*. Phase 4 is negligible and therefore omitted.



(a) Partitioned by the *hypervertices* with the *random* partitioner.

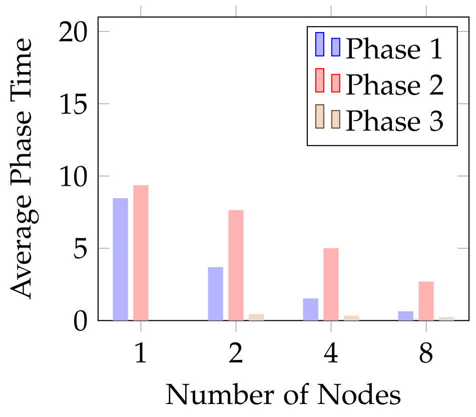


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

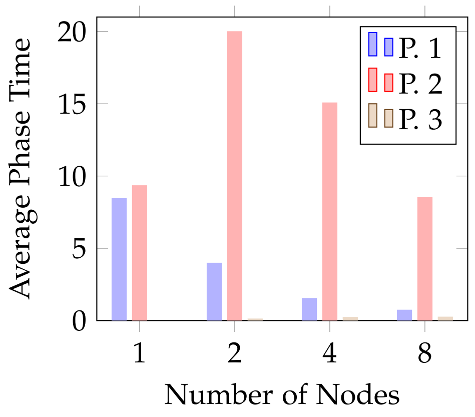


(c) Partitioned by the *hyperedges* with the *random* partitioner.

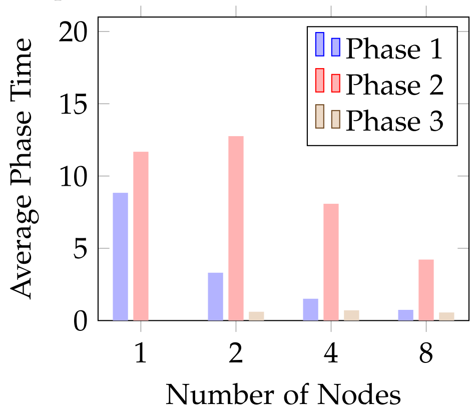
Figure 5.12: Average Phase Running Time for ConnectedComponent on *LiveJournal*. Phase 4 is negligible and therefore omitted. *P. stands for Phase.*



(a) Partitioned by the *hypervertices* with the *random* partitioner.

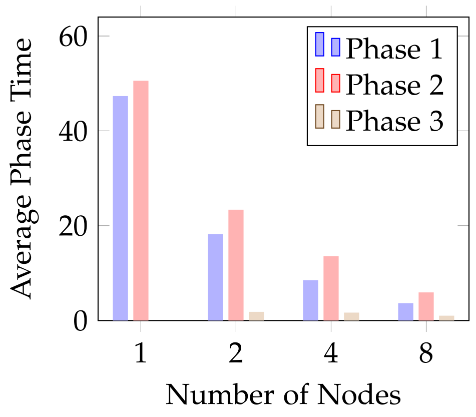


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

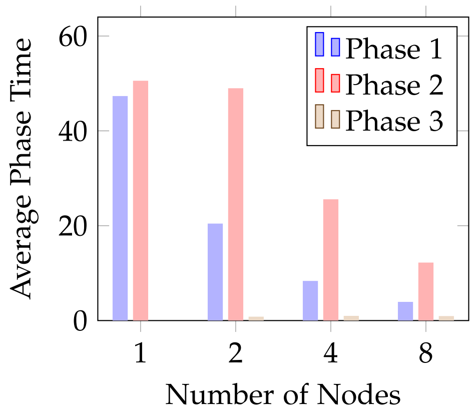


(c) Partitioned by the *hyperedges* with the *random* partitioner.

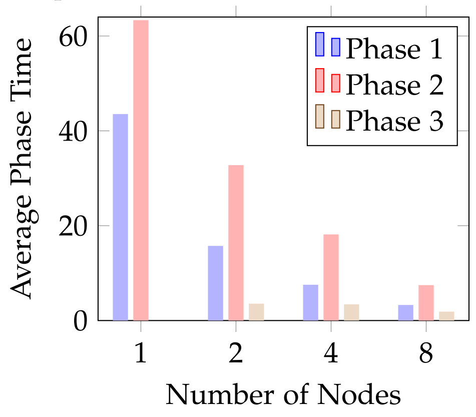
Figure 5.13: Average Phase Running Time for PageRank on *LiveJournal*. Phase 4 is negligible and therefore omitted. *P. stands for Phase.*



(a) Partitioned by the *hypervertices* with the *random* partitioner.

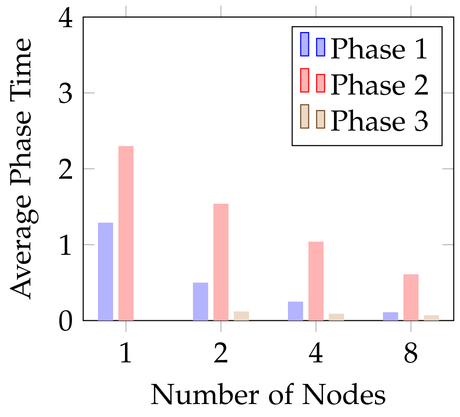


(b) Partitioned by the *hypervertices* with the *Hype* partitioner.

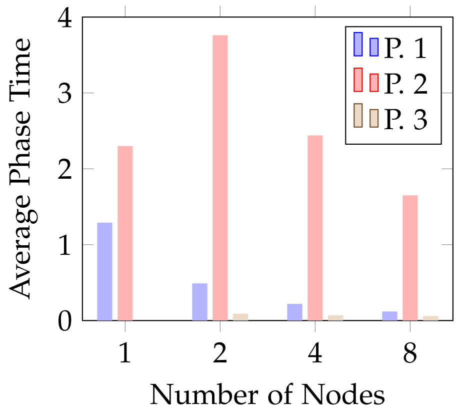


(c) Partitioned by the *hyperedges* with the *random* partitioner.

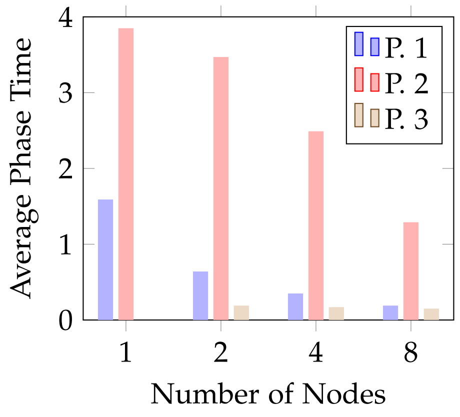
Figure 5.14: Average Phase Running Time for LabelPropagation on *LiveJournal*. Phase 4 is negligible and therefore omitted.



(a) Partitioned by the *hypervertices* with the *random* partitioner.



(b) Partitioned by the *hypervertices* with the *Hype* partitioner.



(c) Partitioned by the *hyperedges* with the *random* partitioner.

Figure 5.15: Average Phase Running Time for SingleSourceShortestPath on *LiveJournal*. Phase 4 is negligible and therefore omitted. *P. stands for Phase.*

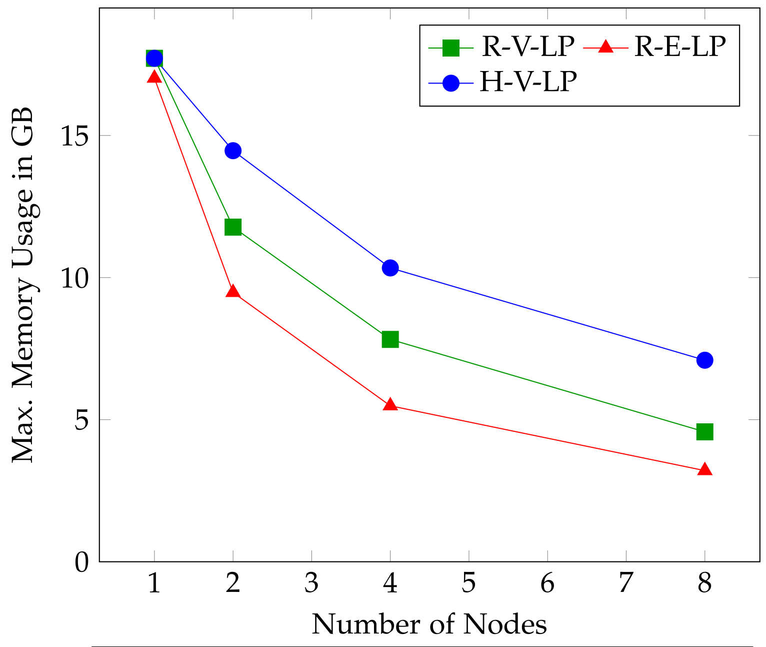
when comparing the random partitions partitioned by the hypervertices with the random partitions partitioned by the hyperedges.

5.7 Comparison with other Tools

Comparing to MESH. Just as the Hygra Paper, we were not able to setup the MESH or HyperX framework on our cluster [Shu20]. However, the MESH paper reports an average superstep running time of about a minute for PageRank on LiveJournal. Their setup consisted of 8 6-core nodes with 64 GB RAM and 750GB free disk space each. We, on the other hand, just needed less than four seconds with 8 2-core nodes when running the same algorithm on the same hypergraph (see Figure 5.3).

Comparing to HyperX. The HyperX paper did not implement the PageRank algorithm but provides numbers for testing LabelPropagation on the Friendster data set. They reported an average superstep running time of about 14.0 seconds on an 8 node cluster with 4 cores and 16 GB memory per node. This is also much more than the 3.6 seconds that we evaluated for the same algorithm and hypergraph (see Figure 5.2), even though our nodes have half the number of cores [Jia+18].

Comparing to Hygra. Fortunately, we were able to install and test Hygra on one of our nodes. An average superstep on the LiveJournal hypergraph took ~4.6 and ~3.1 seconds for the ConnectedComponent and PageRank algorithm. This is much faster than Dyper which took ~11.0 and ~17.9 seconds, respectively (see Figure 5.3). The reason is probably that Hygra is optimized for a single machine [Shu20] while Dyper is not.



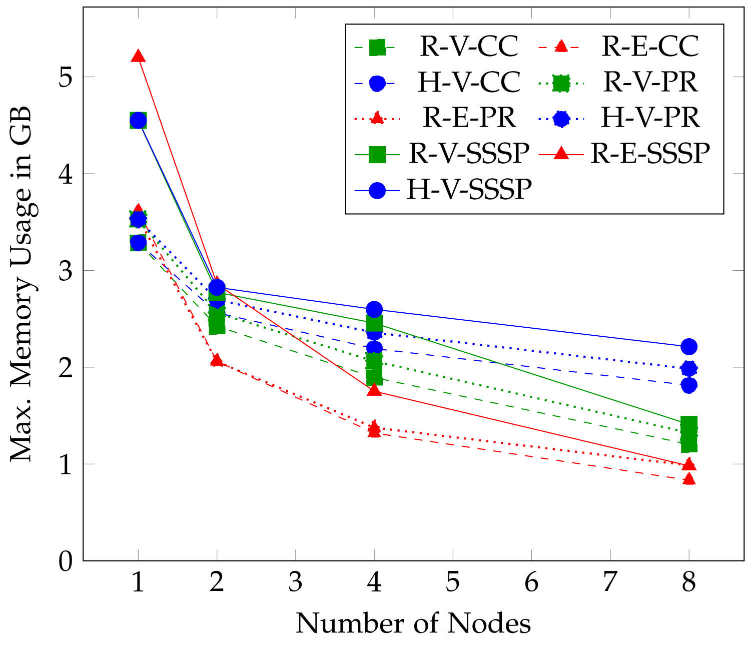
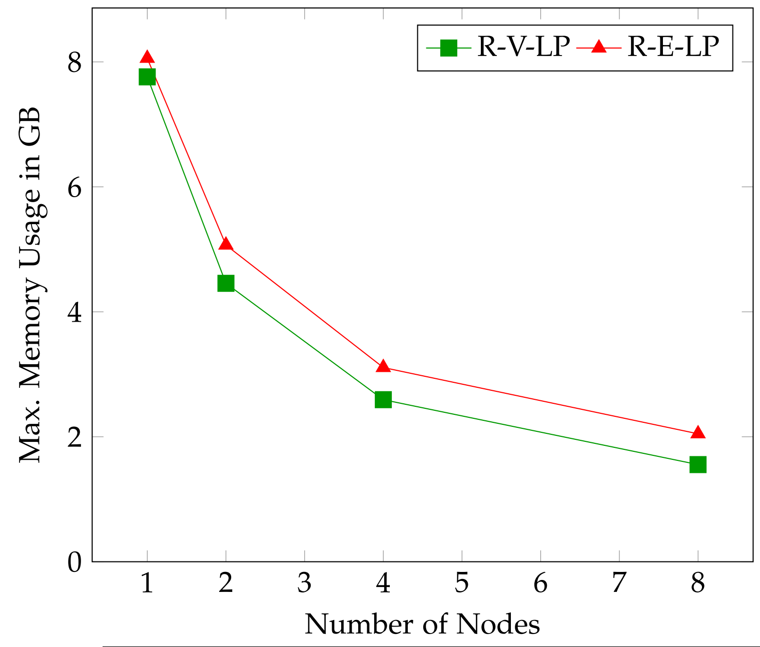


Figure 5.16: Max. memory usage for the *com-Orkut* hypergraph. *R stands for the Random partitioner, H for the Hype partitioner, V for a partitioning by the Hypervertices, E for a partitioning by the Hyperedges.* CC *stands for* ConnectedComponent, LP *for* LabelPropagation, PR *for* PageRank *and* SSSP *for* SingleSourceShortestPath.



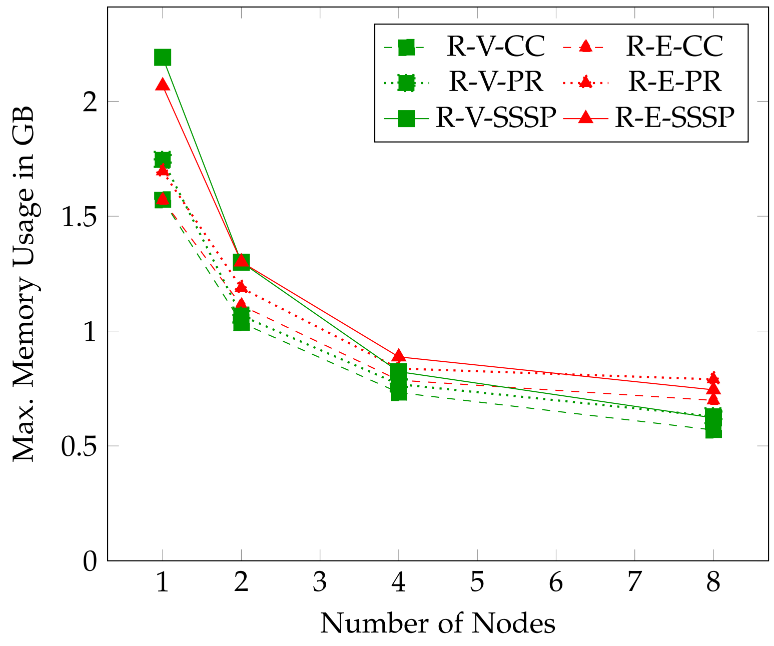
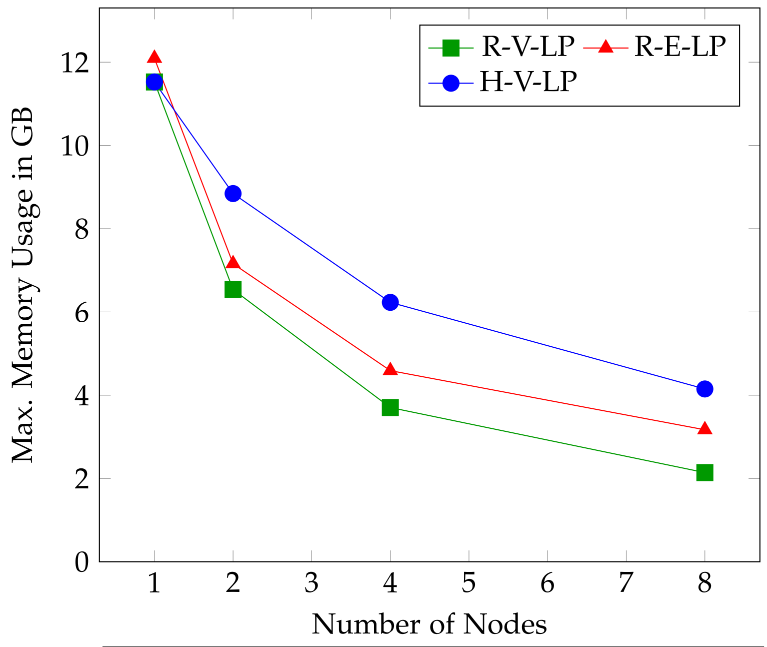


Figure 5.17: Max. memory usage for the *Friendster* hypergraph. Unfortunately, we were not able to partition the Friendster hypergraph with Hype. *R stands for the Random partitioner, V for a partitioning by the Hypervertices, E for a partitioning by the Hyperedges.* CC *stands for* CONNECTEDCOMPONENT, LP *for* LABELPROPAGATION, PR *for* PAGERANK *and* SSSP *for* SINGLESOURCESHORTESTPATH.



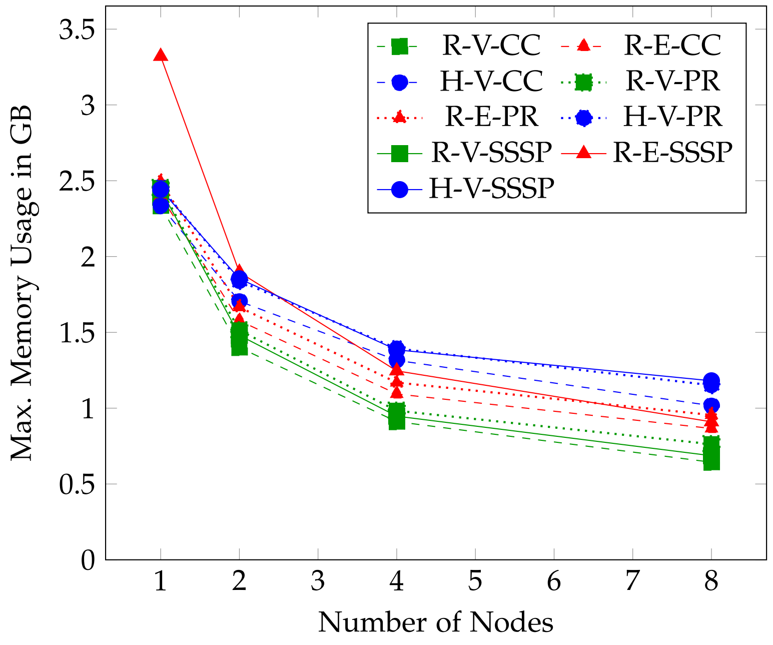


Figure 5.18: Max. memory usage for the *LiveJournal* hypergraph. *R stands for the Random partitioner, H for the Hype partitioner, V for a partitioning by the Hypervertices, E for a partitioning by the Hyperedges.* CC *stands for* ConnectedComponent, LP *for* LabelPropagation, PR *for* PageRank *and* SSSP *for* SingleSourceShortestPath.

6 Conclusion

Until now, user faced a difficult dilemma when processing hypergraphs: Either choose the fast Hygra framework and be constrained to hypergraphs that fit on a single machine [Shu20]. Or take the much slower MESH or HyperX and be very limited by the computing power you have [Hei+19; Jia+18].
Our thesis solves this dilemma. With Dyper, user now have a distributed *and* fast hypergraph processing system. Besides being efficient, Dyper comes with important features: It supports a `combine` function and arbitrary partition strategies: Users do not only have the choice between partitioning their hypergraph by the set of hypervertices or hyperedges but can also provide a custom partitioning.
In our evaluation, we showed that Dyper outperforms MESH and HyperX by a ratio on 15:1 and 4:1 for the algorithms PageRank and LabelPropagation, respectively. This is even more astonishing when we consider that it ran on much fewer cores. However, on a single machine, Hygra still seems to be the best choice. Running on the same node, it was almost four times faster than Dyper.
For the future, we could not only tweak the performance of our program but also add further features to it. For example, our program currently only supports messages and statuses with a fixed size. Relaxing this restriction would open our framework for more algorithms, making it even more useful...

List of Figures

List of Tables

Bibliography

[Ber84] C. Berge. *Hypergraphs: combinatorics of finite sets*. Vol. 45. Elsevier, 1984.

[BF97] A. L. Blum and M. L. Furst. "Fast planning through planning graph analysis." In: *Artificial Intelligence* 90.1 (1997), pp. 281–300. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/S0004-3702(96)00047-1`.

[Bha+12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. "Graph-based analysis and prediction for software evolution." In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 419–429. DOI: `10.1109/ICSE.2012.6227173`.

[Bla] J. Blandy. *Comparison of Rust async and Linux thread context switch time.* `https://github.com/jimblandy/context-switch`. (Accessed on 10/06/2023).

[BP98] S. Brin and L. Page. "The anatomy of a large-scale hypertextual Web search engine." In: *Computer Networks and ISDN Systems* 30.1 (1998). Proceedings of the Seventh International World Wide Web Conference, pp. 107–117. ISSN: 0169-7552. DOI: `https://doi.org/10.1016/S0169-7552(98)00110-X`.

[dash] *dashmap - Rust*. `https://docs.rs/dashmap/latest/dashmap/`. (Accessed on 10/06/2023).

[DB14] A. Ducournau and A. Bretto. "Random walks in directed hypergraphs and application to semi-supervised image segmentation." In: *Computer Vision and Image Understanding* 120 (2014), pp. 91–102.

[DY08] L. Ding and A. Yilmaz. "Image segmentation as learning on hypergraphs." In: *2008 Seventh International Conference on Machine Learning and Applications*. IEEE. 2008, pp. 247–252.

[GJS76] M. Garey, D. Johnson, and L. Stockmeyer. "Some simplified NP-complete graph problems." In: *Theoretical Computer Science* 1.3 (1976), pp. 237–267. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(76)90059-1`.

[Guz19] S. Guze. "Graph Theory Approach to the Vulnerability of Transportation Networks." In: *Algorithms* 12.12 (2019). ISSN: 1999-4893. DOI: `10.3390/a12120270`.

[Hei+19] B. Heintz, R. Hong, S. Singh, G. Khandelwal, C. Tesdahl, and A. Chandra. "MESH: A Flexible Distributed Hypergraph Processing System." In: *CoRR* abs/1904.00549 (2019). arXiv: `1904.00549`.

[Jia+18] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang. "HyperX: A scalable hypergraph framework." In: *IEEE Transactions on Knowledge and Data Engineering* 31.5 (2018), pp. 909–922.

[Kar] G. Karypis. *METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering*. `https://github.com/KarypisLab/METIS`. (Accessed on 10/14/2023).

[KN] S. Klabnik and C. Nichols. *The Rust Programming Language*. `https://doc.rust-lang.org/book`. (Accessed on 09/19/2023).

[Kun] J. Kunegis. *The KONECT Project*. `konect.cc/`. (Accessed on 09/19/2023).

[LKa] J. Leskovec and A. Krevl. *SNAP: Network datasets: Friendster social network*. `http://snap.stanford.edu/data/com-Friendster.html`. (Accessed on 09/11/2023).

[LKb] J. Leskovec and A. Krevl. *SNAP: Network datasets: LiveJournal social network*. `https://snap.stanford.edu/data/soc-LiveJournal1.html`. (Accessed on 09/11/2023).

[LKc] J. Leskovec and A. Krevl. *SNAP: Network datasets: Orkut social network*. `http://snap.stanford.edu/data/com-Orkut.html`. (Accessed on 09/11/2023).

[LK14] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[Mal+10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: a system for large-scale graph processing." In: *Proceedings of the 2010 international conference on Management of data*. New York, NY, USA, 2010, pp. 135–146.

[May+18] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel. "HYPE: Massive Hypergraph Partitioning with Neighborhood Expansion." In: *CoRR* abs/1810.11319 (2018). arXiv: `1810.11319`.

[MK14] N. D. Matsakis and F. S. Klock. "The Rust Language." In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 103–104. ISBN: 9781450332170. DOI: `10.1145/2663171.2663188`.

[nist1] *strongly connected graph*. `https://xlinux.nist.gov/dads/HTML/stronglyConnectedGraph.html`. (Accessed on 09/10/2023).

[nist2] *hypergraph*. https://xlinux.nist.gov/dads/HTML/hypergraph.html. (Accessed on 09/10/2023).

[PLJ23] I. Plauska, A. Liutkevičius, and A. Janavičiūtė. "Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller." In: *Electronics* 12.1 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12010143.

[Raj21] S. Raju. *Main Reasons For The Failure Of Orkut | StartupTalky*. https://startuptalky.com/why-orkut-failed/. (Accessed on 09/19/2023). Apr. 2021.

[RAK07] U. N. Raghavan, R. Albert, and S. Kumara. "Near linear time algorithm to detect community structures in large-scale networks." In: *Phys. Rev. E* 76 (3 Sept. 2007), p. 036106. DOI: 10.1103/PhysRevE.76.036106.

[rustc] *Releases · rust-lang/rust*. https://github.com/rust-lang/rust/releases. (Accessed on 09/19/2023).

[Shu20] J. Shun. "Practical Parallel Hypergraph Algorithms." In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '20. San Diego, California: Association for Computing Machinery, 2020, pp. 232–249. ISBN: 9781450368186. DOI: 10.1145/3332466.3374527.

[Sta07] C. J. Stam. "Graph theoretical analysis of complex networks in the brain." In: *Nonlinear Biomedical Physics* 1 (2007). ISSN: 1999-4893. DOI: 10.1186/1753-4631-1-3.

[std] *std - Rust*. https://doc.rust-lang.org/std/. (Accessed on 09/19/2023).

[tea20] teambizzbucket. *Why Did Friendster Shutdown? Here is the complete analysis! | BizzBucket*. https://bizzbucket.co/friendster-shutdown-analysis/?expand_article=1. (Accessed on 09/19/2023). Sept. 2020.

[TL10] L. Tang and H. Liu. "Graph Mining Applications to Social Network Analysis." In: *Managing and Mining Graph Data*. Ed. by C. C. Aggarwal and H. Wang. Boston, MA: Springer US, 2010, pp. 487–513. ISBN: 978-1-4419-6045-0. DOI: 10.1007/978-1-4419-6045-0_16.

[toklib] *tokio - Rust*. https://docs.rs/tokio/latest/tokio/. (Accessed on 10/06/2023).

[toktut] *Tutorial | Tokio - An asynchronous Rust runtime*. https://tokio.rs/tokio/tutorial. (Accessed on 10/06/2023).

[Tru93] R. J. Trudeau. *Introduction to graph theory*. Dover Publications, 1993.

[ZHS06] D. Zhou, J. Huang, and B. Schölkopf. "Learning with hypergraphs: Clustering, classification, and embedding." In: *Advances in neural information processing systems* 19 (2006).