**Diplomarbeit**

# A flexible and portable approach for communication in distributed computing systems

Erik Zenker

03. December 2014

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur für Rechnerarchitektur

Betreuender Hochschullehrer: Prof. Dr. Wolfgang E. Nagel
Betreuender Mitarbeiter: Dr. Guido Juckeland

**TECHNISCHE UNIVERSITÄT DRESDEN**

Fakultät Informatik Institut für Technische Informatik, Professur für Rechnerarchitektur

# AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

*Name, Vorname des Studenten:* Zenker, Erik

*Immatrikulations-Nr.:* 3421679

*Thema:* **Entwicklung einer generischen Kommunikationsschicht für Multi-GPGPU Simulationen am Beispiel PIConGPU**

*Zielstellung:*

Lastverteilung, Fehlertoleranz und Kollektive Operationen sind heutzutage wichtige Mechanismen im Bereich des Hochleistungsrechnens. Deswegen ist es nötig eine möglichst unabhängige Kommunikationsstruktur zur Verfügung zu stellen, auf welcher diese drei Techniken aufbauen können.

Besonders bei rechenintensiven Multi-GPGPU Particle-in-Cell (PIC) Programmcodes ist eine generische und flexible Kommunikationsschicht ein grundlegender Bestandteil. Diese Schicht, zwischen Applikation und Kommunikationskanal, soll unabhängig von der zu Grunde liegenden Kommunikations-Middleware (z.B.: MPI, ZMQ) sein und somit eine einfache Ersetzung ermöglichen.

Ziel dieser Arbeit ist es, die Kommunikations-Topologie von PIConGPU zu erfassen, daraus eine generische Kommunikationsschicht zu entwerfen und insbesondere für MPI zu implementieren. Dabei ist es die Aufgabe dieser Kommunikationsschicht, Daten zwischen Kommunikationsknoten auszutauschen und somit, in Hinblick auf Lastverteilung und Fehlertoleranz, das Verschieben ganzer Arbeitsbereiche auf andere Rechner oder Geräte zu ermöglichen.

*Betreuender Hochschullehrer:* Prof. Dr. Wolfgang E. Nagel

*Betreuer:* Dr. Michael Bussmann,
Dr. Guido Juckeland, Andy Georgi

*Beginn am:* 03.05.2014

*Einzureichen am:* 03.11.2014

Wolfgang Nagel

Unterschrift des betreuenden Hochschullehrers

## Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 8. Oktober 2018

Erik Zenker

# Contents

# 1 Introduction

The domain of high performance computing is subject to constant development. The main focus of this development is the increase of computing power [**ref:performance_development**]. More computing power enables high performance applications to solve more complex problems or allows for an increase of the problem size. Current state of the art computing systems reached the area of one petaflop, that are $10^{15}$ floating point operations per second (petaflop/s), in the year 2008 [**ref:ibm_roadrunner**]. But only a small number of applications worldwide have taken advantage of these so called petascale systems.

One application that has shown scalability on petascale systems, is PIConGPU [**ref:picongpu_scale**]. It is a particle in cell physics simulation that was running on the Titan supercomputer at the Oak Ridge National Laboratory in the United States [**ref:titan**], reaching 7.1 petaflop/s peak performance by utilizing 18.000 Graphics Processing Units (GPUs). However, applications capable of fully utilizing a petascale system can often be expected to desire systems with even more computational power.

Therefore, current computing systems will no longer adequately cover the future needs of complex, computation intensive scientific and industrial applications. To provide more computing power to these applications, a rapid progress in the development of computers and algorithms is necessary. The next milestone in the development of supercomputers are exascale systems, capable of at least one exaflop/s, which represents a thousandfold increase over the first petascale system.

On the one hand, not all questions about the construction of such a system are solved. On the other hand, it is certain that exascale will increase the amount and complexity of computers to a new level [**ref:cresta**], amplifying the challenge of reliability, programmability and usability of such systems.

Considering an exascale system, it is not sufficient to simply port the applications previously running on petascale systems. The sole, massive increase in size of the computing system will lead to a decrease in the mean time to failure (MTTF) and an increased importance of data locality. Furthermore, the emergence of accelerator hardware leads to computing systems that are more heterogeneous and hierarchical, as well as increasingly complex to program and use [**ref:accel**]. Therefore, applications running on high performance computers have to be adapted to utilize the upcoming generation of supercomputers. Current applications lack techniques that form the basis for load balancing and fault tolerance. Thus, there is a need for a smart description concept for high performance applications that models upcoming exascale computing systems with respect to these techniques.

Many applications exploit the parallel power of clusters by interconnecting their parallel entities via network. These applications can be enhanced by a smart communication approach. This approach will enhance already existing communication libraries by a

further layer, which is necessary for the upcoming supercomputers. This additional layer is separated into three components: an abstract communication layer, a model of the communication topology, and a mapping from the topology onto the abstract communication layer. These mappings can be changed at any time during the application execution. In connection with the exchangeable communication library, it forms a flexible and portable communication approach for the upcoming supercomputer generation. A prototype, that combines these three components in a single system, was implemented and deployed in simulation applications. The message passing interface (MPI) was used within this prototype as communication back-end underneath the communication abstraction layer. It maps common communication processes onto equivalent MPI methods.

The following Chapter 2 first motivates the design of the system and subsequently provides a discussion about of the system components. Selected implementation details of the system is presented in Chapter 3. The developed prototype is evaluated and compared in contrast to an MPI implementation in Chapter 4. Chapter **??** provides ideas to enhance the system in future work. Finally, Chapter **??** summarizes the developed system and Chapter **??** provides an outlook for future development.

# 2 Design

## 2.1 Motivation

These days, complex simulation applications no longer fit on a single computing device like a desktop computer. Therefore, parallel computers are necessary instruments in scientific computing where time to solution is important and a large amount of memory is required [**ref:hpcc1**]. To exploit the performance of a parallel computer, the application needs to be distributed onto the computing environment and exchange data via communication over a network.

For the following assume, the mentioned parallel computer is a compute cluster. In principle, a compute cluster is a distributed memory architecture and forms a homogeneous network of similar compute entities, hereafter referred to as nodes. Furthermore assume, all cluster nodes are equipped with the same hardware and a node provides only one type of computing hardware.

Simulation applications are prepared for this kind of homogeneous clusters by domain decomposition. Domain decomposition is a very popular method to separate the simulation domain of an application into smaller chunks of work, called subdomains, which are often solvable independently of each other. Subdomains exchange simulation specific data through predefined paths. This enables an application to exploit the tremendous performance gain of parallel computers. The decomposed simulation domain $\Phi$ is used to map its subdomains to the set of hardware topologies $\Gamma$ through a mapping $\omega$:

$$\Phi \xrightarrow{\omega} \Gamma$$

Figure 2.1 shows the domain decomposition of a simulation domain and a static one-to-one mapping of its subdomains to the nodes of the compute cluster.

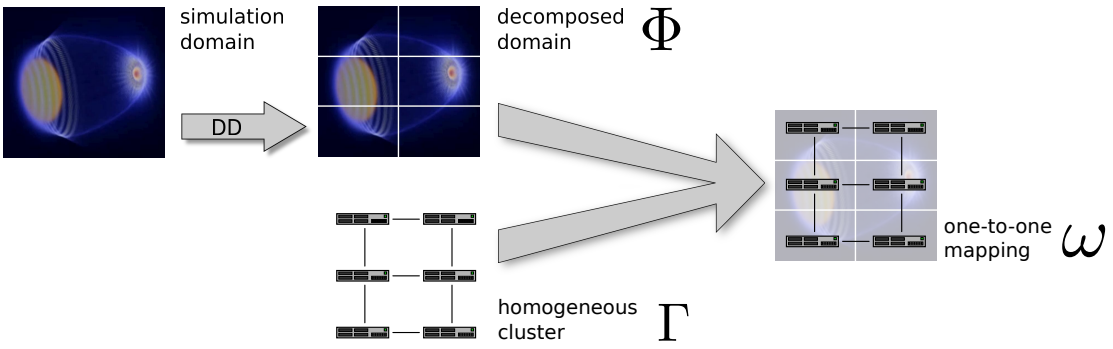


Figure 2.1: The simulation domain is decomposed in subdomains (DD). These subdomains are mapped one-to-one to the homogeneous cluster through $\omega$. This kind of static mapping is sufficient for most applications on cluster systems.

This static mapping is sufficient for most simulations that are executed on clusters. It works, as long as both hardware and simulation stay in this fixed relationship. Thus, no change of the mapping $\omega$ is necessary as long as a simulation will always be executed on the same cluster, with the same network topology, and the same algorithms describing the simulation model.

However, it is not possible to map all modern simulations in this model of a homogeneous world. Some simulations show horizontal and vertical heterogeneous behavior. Horizontal heterogeneity in a simulation denotes that subdomains are not optimal mapped onto the hardware topology due to unbalanced load or failures of nodes. Even though there is an optimal mapping $\omega$ for a time step $t_0$, it does not imply that the same mapping is optimal or even possible for time step $t_1$. Thus, a simulation behaves horizontal heterogeneous if $\omega(t_0) \neq \omega(t_1)$.

Especially for load balancing the implications are that the workload of subdomains can vary during the time of simulation. The simulation may run into an unbalanced state of workload distribution after a time period $\Delta t$. That unbalanced state neither saturates all available computing resources nor does it minimize the overall simulation time. Figure 2.2 shows the workload distribution of the simulation domain through a heat map. The workload distribution changes after a certain time period $\Delta t$. The simulation is decomposed again and mapped through a new $\omega$ to the hardware topology to reach a balanced workload distribution.


Figure 2.2: Horizontal heterogeneity by unbalanced load distribution after a time period $\Delta t$ presented through a heat map. The simulation domain is decomposed again and mapped with a new $\omega$ to the hardware topology. After the remapping the workload is distributed equally.

The failure of a node during the execution of a simulation that implies the termination of subdomain calculations on this node is another problem which leads to heterogeneous behavior. The simulation can be continued if it is possible to map the subdomain of the failed node through a new $\omega$ to another node. Figure 2.3 shows this failure of a node. The subdomain of the failed node is mapped with a new $\omega$.

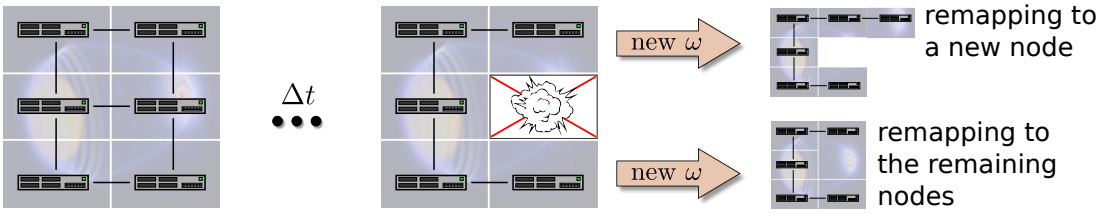


Figure 2.3: Horizontal heterogeneity by the failure of a cluster node after a time $\Delta t$. The subdomain of the failed node is mapped through a new $\omega$ to another node. The simulation can be continued after this remapping.

The mapping $\omega$ is not sufficient to describe vertical heterogeneous effects of simulations. Therefore, the set of communication topologies $\Sigma$ is introduced and $\omega$ is split into the mappings $\sigma$ and $\gamma$, as the following formula shows:

$$\omega : \ \gamma \circ \sigma : \ \Phi \xrightarrow{\sigma} \Sigma \xrightarrow{\gamma} \Gamma$$

Vertical heterogeneity on a simulation is defined by the existence of multiple hardware topologies which requires multiple communication topologies. Each hardware topology might need a sophisticated modeling of their communication mechanism. Figure 2.4 shows the modeling of the all-to-all and next neighbor communication topologies of a simulation by graphs through $\sigma_0$ and $\sigma_1$. Furthermore, these graphs are mapped to the hardware topologies of an accelerator device through $\gamma_0$ and a cluster through $\gamma_1$.

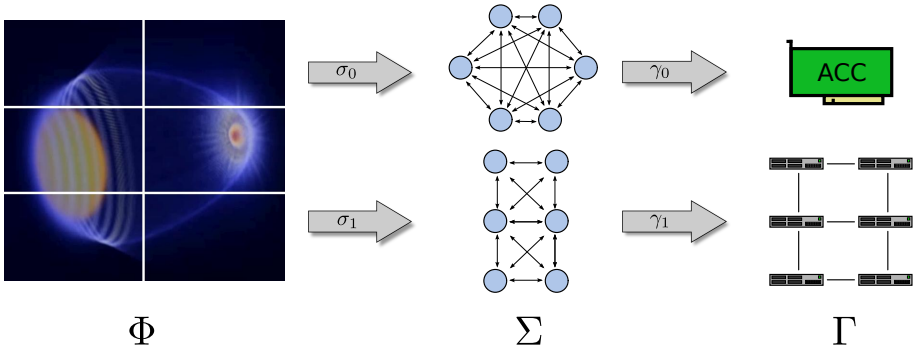


Figure 2.4: Vertical heterogeneity by two communication topologies in a single simulation. The mappings $\sigma_0$ and $\sigma_1$ model different communication topologies. The topologies are mapped by $\gamma_0$ to an accelerator and by $\gamma_1$ to a cluster.

A real world example for such heterogeneous behavior of simulations is the PIConGPU simulation. PIConGPU is a relativistic particle-in-cell (PIC) application designed for simulating laser-plasma interactions [**ref:picongpu**]. The simulation domain is decomposed into subdomains forming a three-dimensional grid of cells. Each grid cell is mapped to a single Graphics Processing Unit (GPU). While magnetic and electric fields of a cell only influence their directly neighboring fields, particles interact globally with each other. The former necessitates next neighbor and the latter all-to-all communication algorithms. Furthermore, during simulation, particles can move to other subdomains, which leads to an unbalanced distribution of particles over the entire simulation domain.

On the hardware side, PIConGPU is developed for the execution on cluster systems with NVIDIA accelerator hardware. CPUs are responsible for data offloading and communication over the network, some of the available accelerators compute the interaction of fields and particles, while others generate a visualization of the simulated data. Thus, vertical and horizontal heterogeneity, both in simulation domains and on the hardware topology, are already present in real world simulation applications.

This work will neither cover the implementation of a communication library nor will it cover load balancing or fault tolerance algorithms. But all these problems are solvable if the implicit mapping of the domain decomposed simulation to the hardware topology is replaced by separate mappings $\sigma$ and $\gamma$.

Thus, this work provides an abstract description which enables to map the simulation algorithm to communication topologies and in turn to map the communication topologies to the hardware topologies by two mappings. These mappings are exchangeable during the run-time of the simulation. The most general approach to describe communication topologies is utilized: a graph. This graph is mapped at run-time to an abstract description of hardware topologies.

An intermediate layer between simulation application and the software stack of the computing system is implemented. This layer allows for the description of the simulation domain by a graph and an explicit mapping of this graph to an abstract communication layer, while the underlying communication library is easily exchangeable. With such a

setup, the mappings can be adjusted to heterogeneous behavior of simulation application and computing hardware.

The following sections will present the design of the implemented system. At first, the requirements for such a system are discussed. It is followed by a description of the overall design. Finally, the design of each component is discussed in detail.

## 2.2 Requirements

The previous Section 2.1 described the state-of-the-art of simulations on cluster systems and provided a motivation to solve the problems of modern simulations. It was determined that an intermediate layer between application and communication library is required. This layer should offer the possibility to describe the communication processes of the simulation in a very general manner. The construction of this intermediate layer involves three steps. First, existing communication libraries should be abstracted by a general communication interface. Thus, communication is not restricted to a single communication library and, therefore, incraeses the simulations portability. Second, the communication topology of the simulation should be modeled explicitly, so that the modeled communication topology can be mapped explicitly onto a communication abstraction layer. Third, the combination of modeled communication topology and the abstraction from communication libraries form an approach to communicate virtually on basis of the described communication topology. In summary, the discussion on the intermediate layer results in the following three requirements of the referred problems of modern simulations:

1. Abstract data exchange method
2. Modeling of the communication topology
3. Mapping of communiation topology onto the abstract communication interface

## 2.3 Design Overview

Physical networks such as Ethernet, Infiniband or Myrinet are the foundation of common communication libraries such as the Message Passing Interface (MPI). MPI is a standardized and portable message-passing specification, defined by the Message Passing Interface Forum [**ref:mpi_specification**]. It has become the de facto standard for communication in cluster systems. An MPI program is a set of autonomous processes, executing the same program, in a single program multiple data style (SPMD, a subcategory of MIMD [**Flynn:1972:COE:1952456.1952459**]). Each process rests in its own address space and communicates via MPI communication primitives with other processes in the same MPI program.

State-of-the-art simulation applications are usually implemented directly on these existing communication libraries. The developed system introduces an intermediate layer between application and communication library. This layer fulfills all requirements set up in Section 2.2. Figure 2.5 shows the components of this intermediate layer

and which requirements they fulfill. On top of the existing communication libraries the communication abstraction layer (CAL) uses library specific adapters to provide a general interface for the upper layer. A graph is used to model the communication topology of the simulation domain and a graph-based virtual overlay network (GVON) maps this graph onto the hardware topology of the communication abstraction layer.

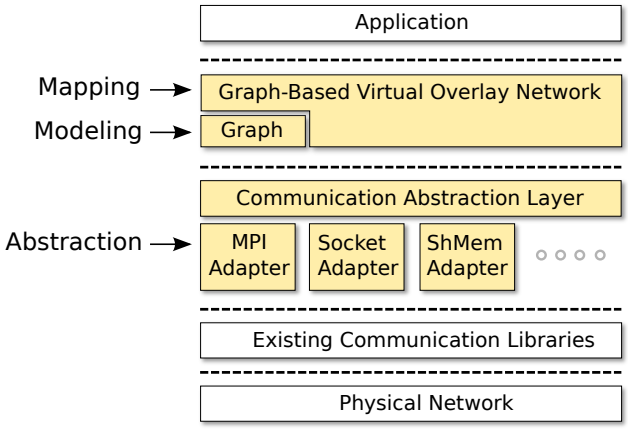


Figure 2.5: Design of the developed system in layers. An intermediate layer between application and communication library is introduced. This layer provides the communication abstraction layer (CAL), the graph description of the communication topology, and the graph-based virtual overlay network (GVON). These components fulfill the discussed requirements in Section 2.2.

The following sections discuss each component in detail: Section 2.4 presents the CAL; Section 2.5 explains the graph interface; Section 2.6 follows up with the GVON. Component interfaces will be described in some cases by pseudo-code which is related to the C++ programming language. It is assumed, that the reader has a basic knowledge of programming and function interfaces.

## 2.4 Abstraction from Existing Communication Libraries

Simulation applications can be interconnected by a vast variety of communication libraries. Most simulations utilize exactly one possible library, but this restricts the simulation to computing systems which only support the selected library (Figure 2.6).

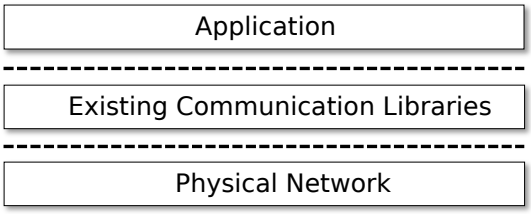


Figure 2.6: The simulation application is implemented directly on top of a communication library. The application needs to directly interface with this library. The exchange of the library and keeping the same interface is hardly possible.

Assuming, that a simulation is distributed on several computers and uses a communication library based on TCP/IP sockets. The execution of this simulation on a single machine is an interesting use case for testing. It results in no problem when the simulation is executed on this local machine and communicates locally by sockets. But, it might be more efficient to use shared memory to exchange data between subdomains, instead of TCP/IP messages.

However, exchanging the communication interface in the example above usually requires a lot of reprogramming of the communication logic. Therefore, it is not a sufficient solution for the problem. Rather, varying communication libraries should be addressable by the same communication interface to make the application portable for varying computing environments. The possibility to exchange the underlying layer without changing the interface to this layer is a precondition for future applications in distributed computing. Thus, the abstraction from this existing communication libraries is a fundamental property for a portable application.

The challenge is to provide a very general interface, that provides the possibility to address varying communication libraries through it. This interface should be as common as possible and should be easily deployable to existing applications by replacing the actual communication library without changing the fundamental understanding of the way to communicate within this application. The following section describes the design of such an abstract communication interface.

### 2.4.1 Communication Abstraction Layer

The *Communication Abstraction Layer*, short *CAL*, is a general communication interface on top of an existing communication library. These libraries are accessible through adapters, which provides the possibility to access varying communication libraries. By exchanging the adapter, the application based on the interface can be ported to varying communication libraries without changing interface function calls. The CAL is configured with a certain adapter at compile-time. Thus, by using this flexible approach, no run-time overhead should occur. The evaluation of the developed system in Section 4 will show the run-time overhead with respect to MPI.

The CAL provides an abstract interface for common communication operations. The adapter needs to implement the interface demanded by the CAL. The communication operations are performed by existing communication libraries wrapped by an adapter.

Instances participating on communication in general are called peers in the following. The interface of the CAL provides basic point-to-point communication operations (Section 2.4.3), collective operations (Section 2.4.5) and operations for the grouping of peers (Section 2.4.4). Figure 2.7 shows the communication abstraction layer on top of existing communication libraries. The CAL with a specific adapter could already replace a certain communication library in a simulation application, but it is not meant to be the level of abstraction the simulation programmer will interact with. The design of further layers of abstraction will be discussed in Sections 2.5 and 2.6.

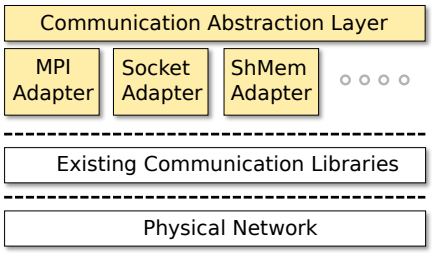


Figure 2.7: The CAL on top of existing communication layers. Varying communication libraries can be addressed through the CAL interface if an according adapter for this library is implemented.

Using the CAL instead of a concrete communication library has the advantage, that only the adapter needs to be replaced when changing the communication environment, for example when migrating to another compute architecture. Although the application has to be recompiled, the CAL interface stays the same.

### 2.4.2 Addressing of Peers

Each communication library provides a particular approach to address peers. Internet socket based systems address their peers by IP addresses, MPI based systems address their processes by ranks, and shared memory based systems are utilizing memory addresses. These diverse approaches to address peers in a network need to be translated to an unified address space to provide a single interface for the CAL user. Therefore, the CAL provides a virtual address, short *vAddr*, which is an unique identifier for its peer. Based on this virtual address, peers are able to address each other through the CAL interface.

The translation of virtual addresses to the adapter specific real address is resolved by the adapter itself. Thus, the adapter defines how the participants of its network are mapped onto the given virtual address space of the CAL. This mapping is hidden from the CAL, so that the CAL only needs to handle virtual addresses and the adapter handles its particular address space.

### 2.4.3 Point-to-Point Communication Interface

In the first place the CAL provides point-to-point communication methods. These are basic methods to exchange abitrary data between two peers, which are available in blocking and non-blocking variants.

A non-blocking communication method returns an *event* object that represents the state of the executed communication method. The state of the *event* is either *ready* or *not ready*. An *event* provides functions to request the current state and to wait until the method has finished. The CAL only provides the event interface. Adapters implement the event interface for their particular communication library since each libary treats non-blocking communication differently [**ref:mpi_specification**, **ref:boost_asio**].

A point-to-point communication method is called with a triplet of arguments: the virtual address of sender or receiver, the message description tag, and the actual data object to exchange. The design of this interface was influenced by existing communication libraries that all share a common interface [**ref:boost_mpi**, **ref:boost_asio**, **ref:zmq**]. The message description tag is the header of the message. It helps to distinguish between messages of the same sending peer. The data object must provide methods to retrieve the data pointer and the size of the data. The data pointer must point to a contiguous memory area. The following lists the point-to-point communication methods of the CAL interface:

- void **send** (destinationVAddr, tag, data)
- void **recv** (sourceVAddr, tag, data)
- Event **asyncSend** (destinationVAddr, tag, data)
- Event **asyncSecv** (sourceVAddr, tag, data)

### 2.4.4 Grouping of Peers

Besides the communication methods, grouping of peers is the most important function the CAL provides through its interface. By default, all peers of the CAL are members of the global group of peers. A group of peers will be called *context*. Accordingly, the global group of peers is called *global context*. A context is valid for all contained peers. In turn, it is invalid for all peers not contained.

A context stores the set of virtual addresses of its members , the number of members, and whether the context is valid for a particular peer. It is the base for communication algorithms with more than two participating peers.

The CAL always provides the global context from which a peer can retrieve its global virtual address. A new context can be created from a subset of peers of an already existing context. Hence, a new context can be created according to the requirements of the communication algorithm. This new context provides its own virtual address space, thus, the virtual address of a peer is context dependent.

Each adapter can handle contexts differently. The CAL only defines the context interface, but each adapter needs to implement the context interface for its particular communication library. The adapter must respect that a virtual address is context

dependent. Therefore, the mapping of a virtual address to a real address has to be adapted to the currently used context.

### 2.4.5 Collective Operation Interface

A *collective operation*, short *collective*, is a communication pattern that is executed simultaneously by all peers of a context. For example, collective are utilized to collect, share and reduce data. Each collective can be implemented by a sequence of peer to peer operations, but communication libraries, especially the ones for high performance calculations such as MPI, often provide optimized collective support.

A collective is always executed with respect to a context and the result is either received by a single peer, the root, or by all peers. While a large number of collectives is known to the parallel computation commmunity, the CAL only provides the most common ones with respect to the analysis of the PIConGPU source code. However, other collectives are implementable in future work. The following lists a subset of the provided collective operations of the CAL and their description:

- void **gather** (rootVAddr, context, send, recv)
  Collection of data elements from every peer in the context. Each peer contributes the same number of data elements. The collected data is received by the root peer.
- void **scatter** (rootVAddr, context, send, recv)
  Distribution of data elements from the root peer to all other peers of the context. The data is divided into chunks of same size, depending on the number of peers. One chunk is send to each peer.
- void **reduce** (root, context, binOperation, send, recv)
  Reduction of data elements from every peer of a context by a binary function that takes arguments of an arbitrary type and returns a values of the same type.
- void **broadcast** (root, context, data)
  Distributes data elements from the root peer to all other peers of the context. Every peer receives the same data elements.
- void **syncronize** (context)
  This operation synchronizes the control flow of all peers of the context.
- void **createContext** (peers, context)
  Creation of a new context from a subset of peers of an already present context.

All peers of the listed collectives send or receive the same number of elements. But some algorithms require an exchange of varying number of elements. Therefore, the collectives gather and scatter are also available in a variant with varying number of elements per peer. These collectives get an additional receive count argument, that describes how many data elements each peer has contributed to the collective. Furthermore are the operations gather, scatter, and reduce available in an all receive variant. All senders of the collective will receive the collective result. Therefore, the specification of a root peer can be omitted for these variants.

## 2.5 Modeling of the Application Domain by a Graph

A simulation model is usually implemented through a particular algorithm. Porting this algorithm onto a parallel computing system requires a decomposition of the simulation domain into a set of subdomains. Communication is necessary if these subdomains need to interact with each other. Whereby, the communication topology of the simulation is determined by the algorithm.

The most general method to model a topology is a *graph* [**ref:graph**]. Furthermore, the explicit modeling of the communication topology by a graph has the advantage that the reuse of this topology reduces the implementation effort for future simulation applications. A graph consists of edges and vertices. Edges are described as a pair of vertices.

Vertices of the graph used in further descriptions are connected by directed edges. Loops and multiple edges between two vertices are allowed (Figure 2.8). Together this results in a directed multigraph with loops, which is called a quiver [**ref:quiver**].

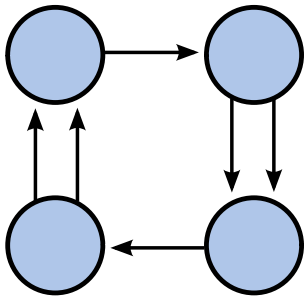

Figure 2.8: The graph interface describes quivers. This image depicts a directed cycle multi graph with four vertices. Graphs can be used to describe the communication topology of a domain decomposed applications.

Changing the simulation algorithm implies a different communication topology and in turn implies a different graph. The graph has the advantage, that the communication topology is modeled explicitly. It is not necessary to rewrite the simulation application as long as the application is based on the topology information provided by the graph. Thus, communication processes can by implemented on top of the graph description. The presented graph is addressable by the following interface:

- VertexList **getVertices** ()
  Returns the list of all vertices of the graph.
- Vertex **getVertex** (index)
  Returns a particular vertex by vertex *index*.
- VertexList **getAdjacentVertices** (vertex)
  Returns a list of all adjacent vertices of *vertex*.

- EdgeList **getOutEdges** (vertex)
  Returns a list of all outgoing edges from the *vertex* paired with their target vertex.
- EdgeList **getInEdges** (vertex)
  Return a list of all incoming edges to the *vertex* paired with their source vertex.
- Graph **createSubGraph** (vertices)
  Returns a subgraph, that only contains the defined *vertices* from the supergraph.

### 2.5.1 Properties

A graph viewed in isolation is only a representation of connected vertices, not providing further information. Interpreting the graph as a C++ object container offers the possibility to insert objects into the graphs vertices and edges.

An object that resides in these containers will be called property and is introduced as a concept to annotate a graph. A property usually provides subdomain information and can be bound to vertices and edges (Figure 2.9). Each vertex and edge of the same graph share the same type of property. A graph annotated with properties is significantly more meaningful and therefore helps to keep the connection between the simulation domain and communication topology.

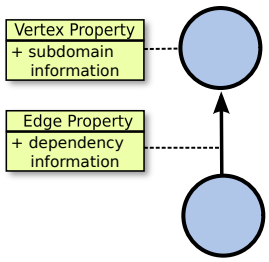


Figure 2.9: A graph annotated with both vertex and edge properties. The vertex property primarily describes a subdomain. The edge property describes the dependencies between subdomains.

The field of application of properties can by diverse. It highly depends on the graph and its area of utilization. A property can also be used to create a connection between a pair of graphs, resulting a hierarchical description.

### 2.5.2 Modeling Game of Life as a Graph

To give an example on how to model a specific simulation application, Figure 2.11 shows the modeling of the Game of Life (GoL) [**ref:gol**] domain by a graph. The GoL simulates the evolution of a set of cells for an arbitrary amount of time steps. The cells are arranged in a two-dimensional grid (Figure 2.10). A cell has a state which is either alive or not alive and the state of the next time step is calculated by rules. The common set of rules determines the state of a cell for the next time step using current state information of the neighboring cells. Assume for the following descriptions there is only one rule: “a

dead cell with exactly three living cells in the neighborhood will be reborn on the next time step".

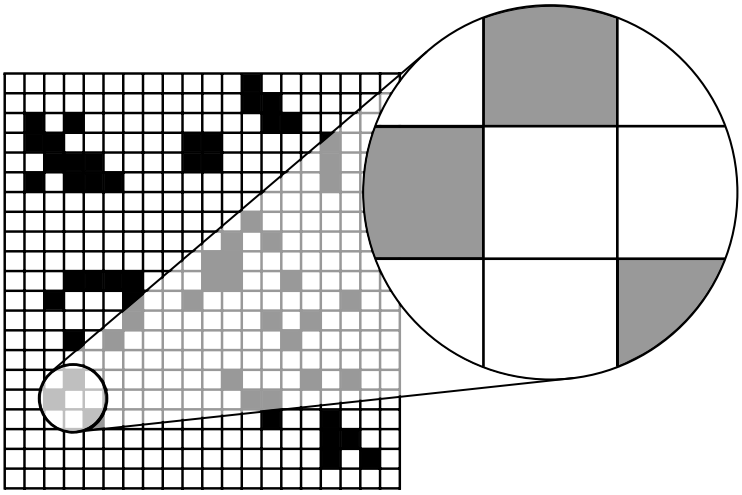

Figure 2.10: Image detail of a GoL domain showing 9 neighboring cells. The GoL domain is a rectangular grid of cells. Cells have a state, while living cells are colored dark, dead cells are white. The state of a cell for the next time step is determined by a set of rules.

The GoL simulation is modeled as follows: every cell is represented by a vertex and neighboring cells are connected by an edge. Each vertex has the property *Cell*, which contains the state of the cell. Figure 2.11 shows a visualized image detail of the modeled GoL domain. To determine the next state of a cell, the number of living adjacent cells have to be counted. The cell will be alive on the next time step, if exactly three adjacent cells are alive.

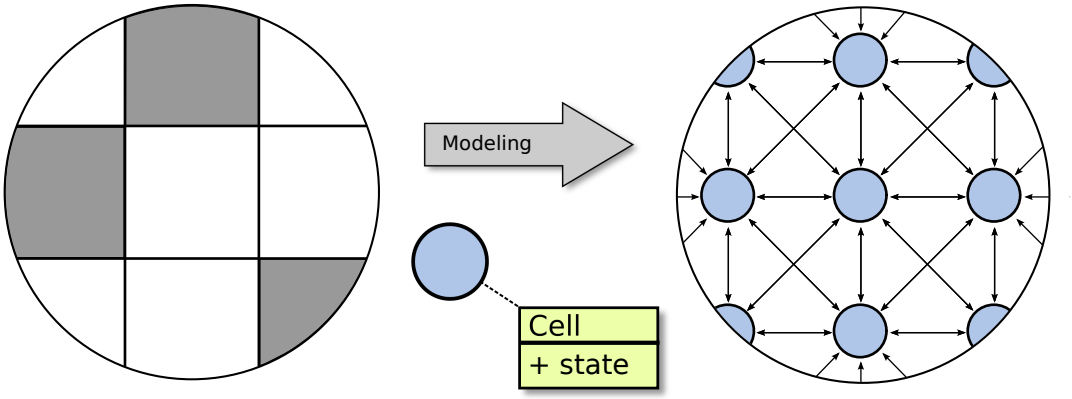


Figure 2.11: Image detail of a Game of Life domain (on the left) was modeled as a graph (on the right). Each Vertex is described by the vertex property *Cell*, containing the state information of the cell.

The rule is now changed slightly: “a cell is alive on the next time step when at least one diagonal neighbor is alive”. A changed rule implies a change in the GoL algorithm. This changes the GoL communication topology and therefore the modeled graph. Figure 2.12 models the GoL domain with the changed rule. Vertices are connected with its diagonal located vertices. Nevertheless, to determine the next state of a cell the number of living adjacent cells have to be counted.

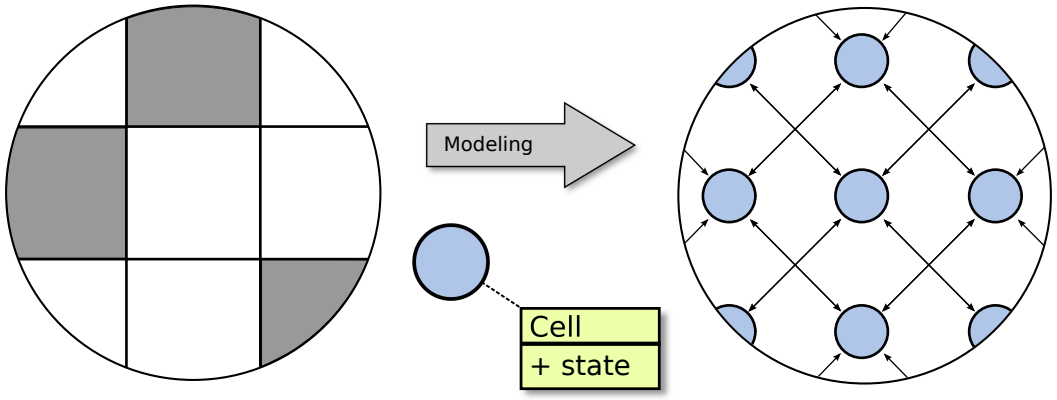


Figure 2.12: Image detail of a Game of Life domain (on the left) modeled with a different rule (on the right). Only diagonally located vertices are connected.

### 2.5.3 Partitioning the GoL Graph

The graph in Figure 2.11 depicts the fine granular model of the GoL domain, where each cell is represented by a vertex. While it is the smallest possible domain decomposition, it might not be very efficient when a single cell is calculated by a single process. One process can compute considerably more cells.

Figure 2.13 shows the creation of a partitioned graph by combining multiple vertices to partitions. Partitions are connected by edges when at least one cell at the border of a partition is the neighbor cell of a border cell from another partition. The creation of a partioned graph with a minimum of connections is the topic of graph partitioning algorithms.

While the fine granular GoL graph represents the GoL cells in detail, the partitioned graph represents dependencies of partitions. The GoL graph and the partitioned graph can coexist and can be connected by their properties. Thus, the properties of the partioned graph changed with respect to the GoL graph, such that partitions store the partitioned cells and the partition connections refer to cells in neighboring partitions.

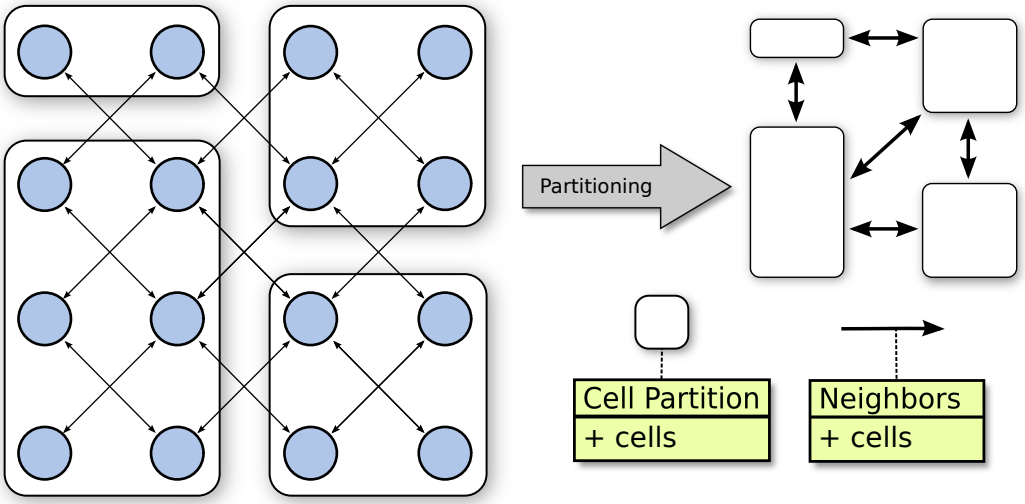


Figure 2.13: The GoL graph is partitioned by combining multiple cells. The properties of the partioned graph have changed with respect to the GoL graph. The vertex property contains the cells of a partition, while the edge property contains information about border cells in neighboring partitions.

The partitioned graph can be used as foundation to distribute GoL to multiple nodes of a cluster. Each partition can be calculated by its own process. The processes calculate a time step of their local GoL graph and communicate the states of their border cells to adjacent bundles.

A different approach to model the GoL domain has been introduced. It used the same description language, the graph. Fine granular modeled domains can be partioned by common graph partition algorithms to obtain more performance by clustering vertices. An automated transformation from a graph description into a more efficient graph description is an interesting topic, but not covered by this work.

### 2.5.4 Modeling a N-Body Simulation as a Graph

The N-body simulation computes particles in a two- or three-dimensional space influenced by physical forces. The force considered for this simulation is the gravity among each particle. A particle is described by its mass $m$, location $\overrightarrow{r}$ and velocity $\overrightarrow{v}$. The gravitational force between two particles $i$ and $j$ can be calculated in a two-body model with the gravitationlal constant $G$ by the Equation 2.1:

$$\overrightarrow{F_{i,j}} = Gm_i m_j \cdot \frac{\overrightarrow{r_j} - \overrightarrow{r_i}}{|\overrightarrow{r_j} - \overrightarrow{r_i}|^3} \tag{2.1}$$

Equation 2.2 describes the gravitational force of particle $i$ to all other N-1 particles as the sum of each particular force $\overrightarrow{F_{i,j}}$.

$$\overrightarrow{F_{i,n}} = \sum_{\substack{j=0 \\ j \neq i}}^{N-1} \overrightarrow{F_{i,j}} \tag{2.2}$$

Therefore, the force affecting particle $i$ is calculated by considering velocity, mass and location of all other particles. Modeling the particle dependencies results in an all-to-all communication pattern with a complexity of $O(N^2)$. Figure 2.14 models the N-body domain with $N = 6$ as a fully connected graph. Particles are represented as pentagons in a two-dimensional space. The size of the particles symbolizes its mass and the arrow its velocity and direction. In the model, a vertex represents a particle and all particles are connected via an edge. A vertex has the property *Particle* that includes the particle mass, location and velocity.

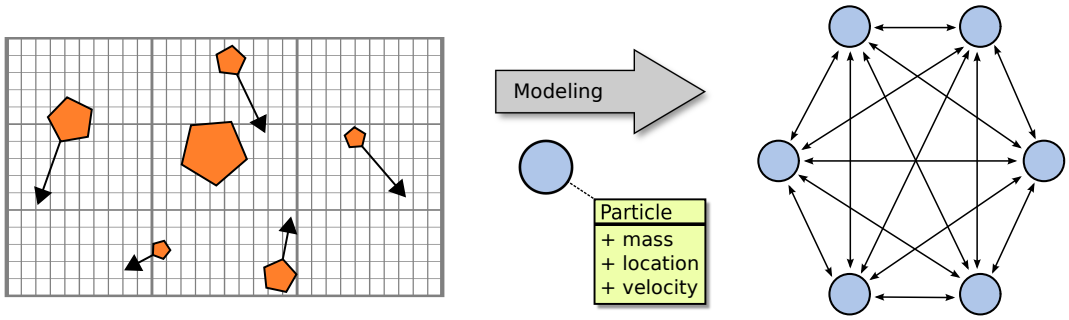


Figure 2.14: Modeling of an N-body domain with $N = 6$. A particle is represented by a vertex and each vertex is connected to all other vertices by edges. Each particle is connected with each other particle forming an all-to-all topology.

A gravitational force on the particle $i$ results in a change of its velocity and location after a fixed amount of time $\Delta t$. The following equations describe these changes.

$$\overrightarrow{a} = \frac{\overrightarrow{F_{i,n}}}{m} \tag{2.3}$$

$$\overrightarrow{v} = \overrightarrow{a} \cdot t + \overrightarrow{v_0} \tag{2.4}$$

$$\overrightarrow{r} = \frac{\overrightarrow{a}}{2} \cdot t^2 + \overrightarrow{v_0} \cdot t + \overrightarrow{r_0} \tag{2.5}$$

## 2.6 Graph-Based Virtual Overlay Network

The previous sections described the tools to communicate between peers and to model a communication topology. The combination of these tools establish a virtual communication layer within the simulation domain. This layer is created by an explicit mapping of the communication topology onto the CAL. Therefore, an application domain and its

according communication topology modeled by a graph can be used to perform communication processes between subdomains. This section introduces a *graph-based virtual overlay network*, short *GVON*, that is based on the graphs and the CAL.

The graph provides a simulation domain specific communication topology. It is used as a blueprint for the virtual network topology. Communication operations are provided by the CAL as base for the overlay network. Figure 2.15 shows the GVON as a layer between the graph and the CAL on one side and the application on the other side.

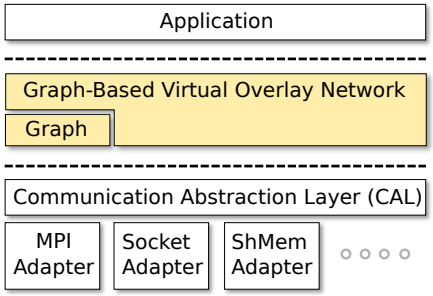


Figure 2.15: The GVON provides communication functionality based on the CAL, but uses the communication topology modeled by the introduced graph.

All peers that want to take part in the communication of the GVON need to know exactly the same graph. In order to ensure that, the graph can be constructed in parallel by all peers, loaded from the same file of a distributed file system, or could even be delivered by a master peer. Furthermore, peers need to use the same adapter configuring its CAL, otherwise communication is not possible.

### 2.6.1 Mapping of a Graph to the CAL

A mapping called the *vertex map* connects the graph and the CAL by a mapping of vertices to peers. A vertex map is valid for a certain graph. Furthermore, the GVON provides a mapping of graphs to contexts (Section 2.4.4), called the *graph map*. This mapping is necessary since the GVON will also be used to map collective operations on a graph to the CAL. The vertex and graph map are the basis for the mapping of communication processes to the communication abstraction layer. The mapping of vertices to peers and graphs to contexts is a joint process of all peers that want to participate in the communication based on a particular graph. This mapping process is divided into two phases.

#### First Phase: Distribute

The first phase distributes vertices of a graph to the peers, whereby every vertex is assigned to a peer. Figure 2.16 shows three peers and a graph of four vertices that are mapped to the peers. It is possible that more than one vertex is mapped to a peer. There are a variety of methods to distribute the vertices. It could be done randomized, round robin, consecutive, or dictated by some master peer. The distribution behavior is defined by the user of the GVON and might be object for further optimization.

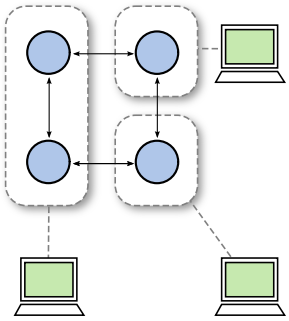

Figure 2.16: All vertices of a graph are distributed onto three peers. The vertices are not distributed evenly, thus, one peer is host for two vertices.

#### Second Phase: Announce

In the second phase, the peers announce their mapped vertices to all other peers. A peer that announces vertices is called host and the announced vertices are called hosted vertices. Each host receives a list of its hosted vertices from every other host. This list is used to update the vertex map and to create a new context only including the hosts. The new context will be bounded to the graph of the hosted vertices in the graph map. A host is responsible for the communication of its hosted vertices. So to speak, it is also possible that a host is responsible for all vertices of a graph and finally communicates with itself the entire time. Depending on the used adapter this might not even be a problem, since communication of a host with itself can be performed in shared memory and does not necessarily include network latency.

### 2.6.2 Communication within the GVON

From the perspective of an overlay network, a vertex is interpreted as a virtual peer: edges between adjacent vertices indicate that the virtual peers are able to communicate with each other. An application, implemented on top of the GVON, has the transparent view of exchanging messages between vertices. The GVON is taking care, that the messages reach the correct vertex host. Finally, this is the level of abstraction an application interacts with.

The GVON provides similar functionality like the presented communication operations in the CAL in Section 2.4.3 and 2.4.5, but on graph-basis. Therefore, both point-to-point and collective operations are provided.

#### GVON Point-to-Point Methods

A point-to-point operations within the GVON involves exchanging data between adjacent vertices over an existing edge. These operations exist in blocking and non-blocking variants. While non-blocking operations return an *event* known from the CAL, nothing is returned by blocking operations. The GVON provides the following interface for point-to-point operations:

- void **send** (graph, destinationVertex, edge, data)
- void **recv** (graph, sourceVertex, edge, data)
- Event **asyncSend** ( graph, destinationVertex, edge, data)
- Event **asyncRecv** ( graph, sourceVertex, edge, data)

The GVON has the task to resolve both the context of a graph and the host of the source or destination vertex. These information are queried from the vertex and graph map. When this information is resolved the programm flow is handed over to the CAL. Finally, the CAL is responsible for the data exchange.

### GVON Collectives

Operations between all vertices of a graph can be performed as collective operations (Section 2.4.5). The collective has to be performed for all hosted vertices of a host. Otherwise the overall execution of the operation is blocked by the CAL. The operation is absolutely transparent for each vertex. the result of the collective is received by a root vertex or by all vertices of the graph.

A collective operation is first executed locally for all hosted vertices of a host. Afterwards, it is handled by the CAL and transmitted to the receiver(s). Figure 2.17 shows a gather operation on the same graph and mapping of Figure 2.16. The gather operation first collects the data of hosted vertices of a host locally, then uses the gather operation provided by the CAL. A similar approach is used for the reduce operation, whereby data is either collected or reduced locally to a single value, depending on the commutativity of the reduce operation.

The execution of the collective operation can be done sequentially or in parallel. Both variants have their specific implementation and usage specific problems. While sequential execution could lead to dead lock behavior when a host forgets to execute the collective for at least one hosted vertex, a parallel execution needs to ensure that the GVON is implemented in a thread safe manner.

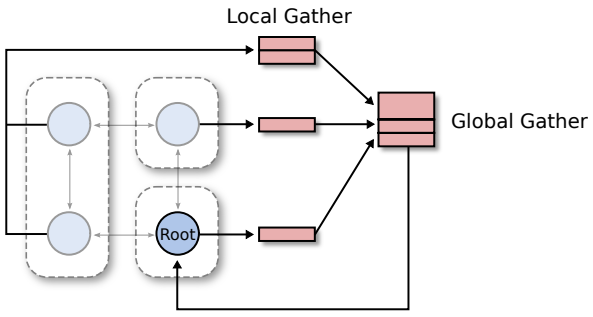


Figure 2.17: Gather operation of the GVON. Data is first locally and then globally gathered. Finally, the gathered data is transmitted to the root vertex. The local gather is executed by the GVON and the global gather by the CAL.

### 2.6.3 Remapping of Vertices

Since the communication topology is separated from the communication library, it is possible to move a hosted vertex to another host at run-time. This run-time behavior is an interesting fact with respect to load balancing and fault tolerance.

Figure 2.18 shows the remapping of hosted vertices of Figure 2.16. The mapping is modified, such that the hosted vertices are distributed only to two hosts. One host is responsible for three vertices and the other for one vertex. The third peer is not part of this graph anymore.

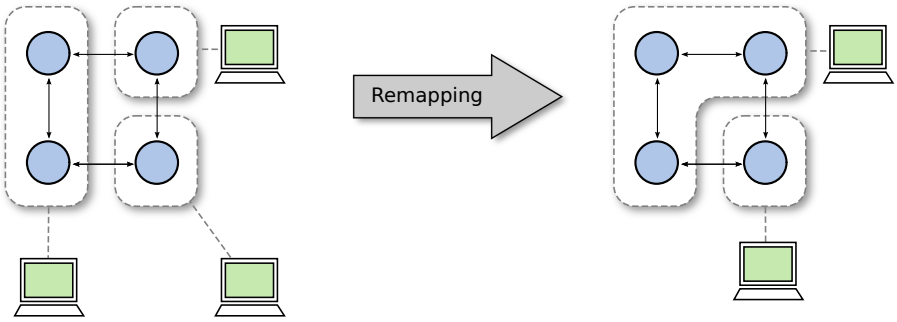


Figure 2.18: Redistribution of vertices to peers. The number of hosts of the graph is reduced from three to two hosts.

Load balancing can be an issue when the communication topology is changing during simulation execution. For example, the GoL simulation could add a new rule that leads to a different communication topology for that might exist a better vertex mapping. A traditional field of application for remapping would be unbalanced workload in the simulation application. Such an unbalanced state can be recognized by monitoring and eliminated by remapping the workload in a fair way.

Another issue is the failure of single cluster nodes and therefore also a failure of hosts located on these nodes. Assuming that the data of hosts were stored in a checkpoint, the hosted vertices of the failed host can be adopted by another host, also at run-time. The process of remapping is also divided in two phases: distribution and announcement and therefore very similar to the mapping process of Section 2.6.1.

## Design Summary

This chapter provided an abstract description which enables to map the simulation algorithm to communication topologies and in turn to map the communication topologies onto the hardware topologies by two separate mappings $\sigma$ and $\gamma$ mentioned in Section 2.1.

These mappings were establisehd as an intermediate layer between the simulation application and the communication topologie. This layer was built from three components: the CAL (Section 2.4.1) that provides an abstract communication interface for communcation libraries, graphs (Section 2.5) that model the communication topologies,

and the GVON (Section 2.6) that provides communication operations based on the graphs through mappings of vertices to peers at run-time.

# 3 Implementation

While chapter 2 presented the design of the developed system will this chapter pick up some design details and focus on their implementation. It is not intended to provide a full documentation of the developed source code. Instead, it is a guide in understanding the system that provides a basis for further development. Therefore, documentation should be retrieved from the source code or doxygen [**ref:doxygen**].

The development of the system started from the analyzed requirements of the PIConGPU source code. Since PIConGPU is implemented in C/C++ and utilizes the Boost libraries [**ref:boost**], the C++ programming language was chosen for this implementation. Additional libraries were selected with respect to similar design paradigms and an easy integration into a C++ environment. Furthermore, the implementation utilizes Standard Template Library (STL) methods as often as possible to guarantee comfortable integration into existing applications. To fully utilize the rich feature set of the C++ programming language, features of the new C++11 standard were also used in this implementation [**ref:c++11**].

The objective was to develop the designed system as a fully functional prototype. The prototype is utilized to implement a simple example simulation such as GoL and N-body to proof functionality.

Particular C++ implementation details are highlighted by small source code snippets. Hence, a basic knowledge of the C++ programming language and templates is assumed.

This chapter is divided into three parts. Each part describes the implementation of one component of the designed intermediate layer. Starting with the CAL in Section 3.1, that discusses first interface related issues and second the implementation of an MPI adapter. Afterwards, A graph implementation based on the Boost Graph Library is discussed in Section 3.3. Finally, the implementation of the GVON interface is described in Section 3.4 and utilized in Section 3.5 and 3.6 to implement the GoL and N-body simulations.

## 3.1 The CAL by Policy Based Design

The CAL was introduced in Section 2.4.1 as a portable and flexible communication abstraction based on an particular adapter. While the CAL defines the adapter interface, the adapter implements communication operations based on an existing communication library. It was required, that the adapter should be exchangeable and configurable at compile-time. These requirements meet the properties of a policy based design [**ref:policy_based_design**].

Policy based design is a programming paradigm especially for the C++ programming language based on the concept of policy and host classes. A policy is a class or class

template interface, which provides inner type definitions, member functions, and or member variables. An implementation of a policy is called policy class and is inherited by or contained within a host class. The advantage of policy based design is that the functionality of the policy is bound to its host class at compile-time, providing no run-time overhead. The interface of the policy is strictly defined by the host class. Not adhering to this interface leads to compile-time errors. In summary, the host class defines the interface of the policy, while the policy class implements arbitrary functionality against the determined host interface. It is also called the compile-time variant of the strategy pattern in [**ref:policy_strategy**].

Applying the policy based design on the design of the CAL induces an allocation of roles: the CAL is the host class and the adapter the communication policy. In the following will communication policy and adapter be used as equal terms. Figure 3.1 shows the modeling of the CAL in an UML diagram. The CAL class is configurable with an adapter class as its template argument. At the same time it inherits the properties of the adapter class in protected mode. In addition to communication and context related methods, the CAL class also provides class interface definitions of the context and event class. They inherit their functionality from corresponding classes in the adapter class, also in protected mode. Thus, an adapter has to implement an inner context and an inner event class. Section 3.2 will discuss the implementation of these adapter-specific context and event class implementations in detail.

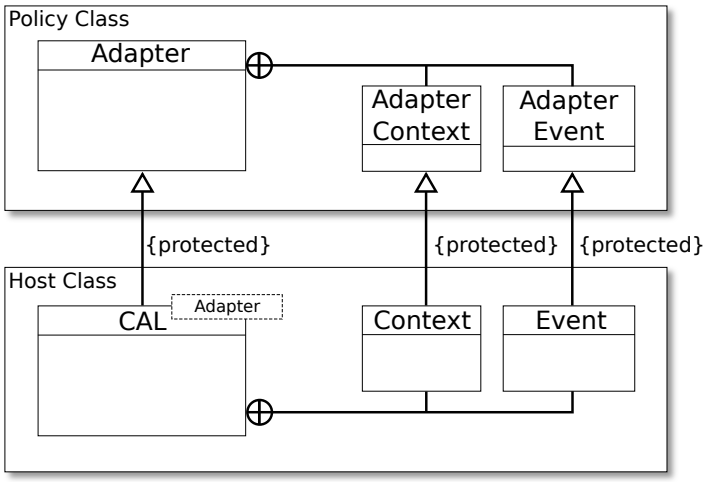


Figure 3.1: The CAL implemented through policy based design. The adapter is configured by a template argument at compile-time. The CAL defines the context, event and communication interface. The adapter implements the determined interface of the CAL and the adapter-specific context and event classes.

The CAL provides all communication methods discussed in Section 2.4.3 and 2.4.5 and context creation methods discussed in the following Section 2.4.4. The communication policy implements all methods required by the adapter definition that are discussed in Section 3.1.1.

### 3.1.1 Communication Policy interface

An adapter needs to implement the policy interface determined by the CAL. The interface was inspired by the sophisticated MPI language binding for C [**ref:mpi_c_binding**]. Therefore, it is implemented on a rather low level of abstraction to provide the most possible flexibility. A subset of adapter methods was chosen to illustrate the interface. A full listing of all methods should be retrieved from the source code documentation.

- Point-to-Point Operation

```
// Adapter Method
template <typename T_Data, T_Context>
Event asyncSendData(const T_Data* data,
                    const unsigned sendCount
                    const VAddr destination,
                    const T_Context context,
                    const unsigned tag);
```

- Collective Operation

```
// Adapter Method
template <typename T_Data, T_Context>
void gather(const T_Data* sendData,
            const unsigned sendCount,
            const T_Data* recvData,
            const unsigned recvCount,
            const VAddr root,
            const T_Context context);
```

- Grouping

```
// Adapter Method
template <typename T_Context>
T_Context createContext(const std::vector<VAddr> vAddrs,
                        const T_Context oldContext);
```

The adapter interface only provides a subset of operations known from MPI or other communication libraries. Most of these operations were necessary during the implementation of the system or used in example applications. Extending the adapter interface with more operations is left as future work.

### 3.1.2 Mapping to Virtual Addresses

Chapter 2 established the requirement for an unified address space to uniquely address peers in a context. The design decision was focused on simplicity. Therefore, the address space for the virtual addresses is in the range of natural numbers from zero to the number of peers in a context minus one. A adapter needs to map its own address space into the virtual address space. An example for such a mapping is provided in Section 3.2.

### 3.1.3 CAL Data Object Interface

All communication methods are able to transmit data objects of an arbitrary data type defined by the template argument `T_Data`. The data object needs to provide a pointer to the memory location where the data is stored and the number of data elements to transmit. The method `data()` must return the memory pointer and the method `size()` the number of elements of the data. C++ Containers of the STL such as `std::vector` and `std::array` [**ref:vector**, **ref:array**] already provide these methods. Therefore, data can be easily encapsulated into these containers. The following listing shows the implementation of the `asyncSend()` method of the CAL. It shows the usage of the discussed data object interface. The object `sendData` with data type `T_Data` must provide both functions required by the data object interface. Utilizing `sendData.data()` and `sendData.size()` fulfills the interface requirements of the communication policy stated in Section 3.1.1.

```
1  // Communication Abstraction Layer Method
2  template <typename T_Data>
3  Event asyncSend(const VAddr destVAddr,
4                  const Tag tag,
5                  const Context context,
6                  const T_Data& sendData){
7
8    return Event(
9      CommunicationPolicy::asyncSendData(
10       sendData.data(),
11       sendData.size(),
12       destVAddr,
13       context,
14       tag
15
16     )
17
18   );
19
20 }
```

Listing 3.1: Data objects of the template data type `T_Data` must provide the methods `data()` and `size()` which need to offer the memory pointer and the number of elements.

### 3.1.4 Reduce Operation Interface

The CAL reduce operation uses a binary operator to reduce a set of objects of several peers. The C++ STL already provides a handfull of binary functions in the *functional* header such as `std::plus`, `std::multiplies` or `std::minus` [**ref:functional**]. Functions that are not provided by the STL can be easily created with a C++ struct or class that overloads the parenthesis operator. The following listing shows such a struct for the maximum operator.

```
template<typename T_Data>
struct maximum : public std::binary_function<T_Data, T_Data, T_Data> {

  // Parenthesis operator overloaded with binary operator for maximum
  const T_Data& operator()(const T_Data& x, const T_Data& y) const {
    return x < y? y : x;

  }

};
```

Listing 3.2: Implementation of the binary operator maximum. These kind of binary operators can be used for the CAL reduce operation. Some binary operators are predefined in the C++ STL functional header.

An adapter that implements the reduce operation has to ensure that it can handle binary operators. There may exist cases in which the C++ binary operator has to be transformed to an adapter-specific one. In these cases, the tranformation logic needs to be implemented inside the adapter class. Section 3.2.1 discusses the transformation of binary operators by the MPI adapter.

## 3.2 The MPI Adapter as Reference Adapter Implementation

A reference adapter is presented to provide an insight into the development of an adapter and to provide hints on parts of the adapter development that should be considered carefully. Furthermore, the description provides a guideline for the implementation of additional adapter classes.

The reference adapter implementation is based on MPI as existing communication library. The adapter acts as a translation or mapping from the CAL interface to the MPI interface. MPI was chosen because it already provides a lot of functionality required by the CAL interface without any further effort. Additionally, it is available on wide range of computing systems and can be used for free by open source implementations. This section will utilize very MPI specific vocabulary. It is assumed that the reader has a basic knowledge on MPI terminology.

The instantiation of a CAL object configured to use the MPI adapter leads to an initialization of the MPI adapter. This initialization step creates the initial global context by grouping all peers of `MPI_COMM_WORLD`. Furthermore, a mapping from MPI ranks to virtual addresses for the global context is created in the `vAddrMap`. With respect to the definition of virtual addresses in Section 3.1.2, the initial `vAddrMap` contains a one-to-one mapping from virtual addresses in the global context to MPI ranks in the `MPI_COMM_WORLD`. The global context and each derived context are mapped to MPI Communicators through a map data structure called the `contextMap`.

The MPI C language binding is used inside the adapter to address the message passing interface. It provides the most general interface to MPI and can be integrated into a C++ environment. This MPI interface is very similar to the adapter interface. Therefore,

the implementation of the adapter interface through MPI routines is not very complex. Nevertheless, the following listing shows that the `asyncSendData` method does not only forward the function call to the MPI communication method `MPI_Issend`. But it also translates the virtual address `destination` of the peer to the rank of the MPI process through the `vAddrMap` at line 10. Furthermore, it performs a translation of the context to the MPI communicator in line 13 through the `contextMap`. Finally, it converts the data type `T_Data` to an MPI data type in line 16. The conversion of data types will be discussed in detail in Section 3.2.2.

```
1  // MPI Adapter Method
2  template <typename T_Data, typename T_Context>
3  Event asyncSendData(const T_data* const data,
4                      const unsigned count,
5                      const Vaddr destination,
6                      const T_Context context,
7                      const unsigned msgType){
8
9    // Translation of vaddr to rank
10   int destRank      = vAddrMap[context][destVaddr];
11
12   // Translation of context to MPI Communicator
13   MPI_Comm comm     = contextMap[context];
14
15   // Conversion from T_Data to MPI data type
16   MPI_Datatype type = ToMPIDatatype<T_Data>::type;
17
18   // MPI specific send operation
19   MPI_Request request;
20   MPI_Issend(const_cast<T_Data*>(data),
21              count,
22              type,
23              destRank,
24              msgType,
25              comm,
26              &request);
27
28   // Create and return event
29   return Event(request);
30
31 }
```

Listing 3.3: Communication method for non-blocking sending of data within the MPI adapter class. The virtual address of the destination is translated to its MPI rank. The context is translated to an MPI communicator and the template data type `T_Data` is translated to an MPI data type. The communication operation is finally performed by MPI.

The remaining adapter methods are implemented in a similar manner, except for the reduce operation. Since the CAL interface accepts arbitrary binary operators for reducing data, the MPI adapter must be able to handle these operators. The following Section 3.2.1 covers the conversion of C++ binary operators to operators that are utilized in MPI.

### 3.2.1 Transformation of Binary Operators

MPI provides built-in support for binary operators by two approaches. The simplest approach is the direct usage of predefined binary MPI operators [**ref:mpi_bin_op**]. But, since the system user should not utilize MPI operators directly, the CAL could wrap all available MPI operators in a struct of static const expressions like shown in Listing 3.4. The CAL could forward these structs to the user of the system and the user could select from these binary operators. For example, a reduction by accumulation would use the `BinaryOperations::SUM` operator.

```
struct BinaryOperations {
  static constexpr BinaryOperation MAX = MPI_MAX;
  static constexpr BinaryOperation MIN = MPI_MIN;
  static constexpr BinaryOperation SUM = MPI_SUM;
  static constexpr BinaryOperation PROD = MPI_PROD;

};
```

Listing 3.4: A subset of binary operators derived by transforming MPI operations to static const expressions.

The downside of this approach is, that only twelve predefined operators exist, which restricts the application to this limited set of operators. MPI provides the possibility to create abitrary binary operators with the `MPI_Op_create` function. The source code for transforming abitrary binary operators to binary MPI operators was taken from the boost::mpi project [**ref:boost_mpi**] and was adapted for the needs of the MPI adapter. The following listing shows the transformation of the binary C++ operator `op` from type `T_BinaryOperation` with result data type `T_Data`. After the transformation, the MPI operator can be utilized by all MPI routines that are using binary operators through `MPIOperator.getOperator()`.

```
// Binary C++ operator
T_BinaryOperator op;

// Transformation of binary operator
toMPIOperator<T_Data, T_BinaryOperator> MPIOperator(op);

// Retrieve binary MPI operator
MPIOperator.getOperator()
```

The second approach is substantially more flexible as the first one. Thus, it was selected to implement binary operators for the MPI adapter.

### 3.2.2 Data Type Conversion

C++ data types need to be transformed to MPI data types within the MPI adapter. On the one hand, MPI predefines primitive data types, but on the other hand also provides the possibility to define data structures based upon sequences of the MPI primitive data types. These user defined data structures are called derived data types.

Usually, primitive data types are contiguous, which means that they form a solid block of memory with no gaps in between. However, derived data types allow you to specify non-contiguous data in a comfortable manner and to treat it as though it was contiguous. Therefore, more complex data types can be transformed into derived data types. The exchange of complex data types was not necessary in the development of the system based on examples of Section 3.5.2. Therefore, the implementation was focused on the transformation of primitive data types. Nevertheless, a transformation is available in the boost::mpi [**ref:boost__mpi**] implementation. Thus switching to boost::mpi or adopting their source code would solve this problem without any further effort.

The primitive C++ data types can be mapped directly to primitive MPI data types. Based on the fact that MPI data types are defined as C macros, a C++ data type needs to be converted to an integer number. The conversion is implemented by the concept of type traits [**ref:type__trait**]. Type traits are classes that obtain characteristics of types in the form of compile-time constant values.

The type trait is responsible for the transformation of primitive C++ data types to primitive MPI data types. In Listing 3.5 a template struct defines the default behavior of the type trait. The struct defines a static const expression `type`, which is set to a fixed MPI data type. The template argument `T_Data` is the primitive C++ data type that should be transformed. The default struct will transform arbitrary data types `T_Data` to the `MPI_INT` data type. The MPI data type of `T_Data` can be retrieved by `ToMPIDatatype<T_Data>::type`.

```
1  template<typename T_Data>
2  struct ToMpiDatatype {
3
4    static constexpr MPI_Datatype type = MPI_INT;
5
6  };
```

Listing 3.5: A templated struct defines the default behavior of the type trait. It will transform arbitrary data types `T_Data` to the `MPI_INT` data type if no other type trait is defined

The template in Listing 3.6 is an explicit specialization of the default template in Listing 3.5. In this case, a specialization for the C++ data type `char`, which will be transformed to the `MPI_CHAR` data type. Thus, each transformation of a primitive C++ data type needs to be defined by its own specialization template.

```
template<>
struct ToMPIDatatypes<char> {

  static constexpr MPI_Datatype type = MPI_CHAR;

};
```

Listing 3.6: A specialization of the default type trait for the C++ data type `char`. It transforms the C++ data type `char` to the `MPI_CHAR` data type of MPI.

The communication operations within the adapter use the type traits to transform the data types of the input and output data by querying the type trait. The following listing provides an example for the conversion of the C++ data type `char`.

```
MPI_Datatype type = ToMPIDataType<char>::type;
```

### 3.2.3 MPI Adapter Specific Event Implementation

The MPI adapter-specific `Event` utilizes the MPI functions `MPI_Wait` and `MPI_Test` to test non-blocking operations for termination. In MPI, a non-blocking function returns a request object, which is the handle to verify if a the function terminated. The following listing shows the `Event` implementation of the MPI adapter. The `wait()` method is implemented by using `MPI_Wait` on the request. The `ready()` method uses `MPI_Test` to check if the non-blocking function terminated.

```
class Event {
  public:

  Event(MPI_Request request) : request(request){

  }

  ~Event(){

  }

  void wait(){
    MPI_Status status;
    MPI_Wait(&request, &status);

  }

  bool ready(){
    int flag = 0;
    MPI_Status status;
    MPI_Test(&request, &flag, &status);
    return bool(flag);

  }

  private:

  MPI_Request request;

};
```

Listing 3.7: MPI adapter-specific `Event` implementation. Non-blocking MPI operations can be handled by an `MPI_Request` object. The `wait()` method waits until the communication operation terminated with `MPI_Wait`. The `ready()` method checks if the communication operation terminated with `MPI_Test`.

## 3.3 Graph Based on the Boost Graph Library

Section 2.5 introduced a graph interface. This graph interface is implemented on basis of an existing graph library as back-end. A wide range of graph libraries such as [**ref:lemon**, **ref:boost_bgl**, **ref:igraph**, **ref:ogdf**] were considered as back-end libraries. However, since, the Boost project is closely related to the STL, the *Boost Graph Library* [**ref:boost_bgl**] (BGL) was selected as graph back-end library. While the BGL provides a powerful and complex interface with wide range of graph algorithms, just a small subset of this functionality is really necessary.

Therefore, the BGL is wrapped inside a graph class providing common graph functionality which simplifies the BGL interface. The graph class provides all methods discussed in design Section 2.5, but the methods are internally handled by the BGL.

### 3.3.1 Vertices Annotated by Properties

A graph in the BGL can be annotated by property objects. These objects are used to describe vertices and edges with simulation-specific information, as it was discussed in Section 2.5. Vertices and edges are provided with properties at compile-time through a template parameter. The BGL refers to vertices and edges by indices of integer numbers, whereby properties of this vertices can be queried from so called property maps. The graph connects these indices with its properties to create a vertex and edge description.

The implementation of a property is a struct or a class with arbitrary content. The simplest property is a struct only containing an `id`, as presented in Listing 3.8. This `id` is necessary since it is used to create an internal mapping of the property to the vertex index of the BGL. The `SimpleProperty` is predefined in the graph header. It is used as default property when the graph is not configured by a certain property and it can be utilized to create custom properties by inheritance.

```
1  struct SimpleProperty {
2
3      SimpleProperty() : id(0) {
4
5      }
6
7      SimpleProperty(ID id) : id(id) {
8
9      }
10
11     // Vertex id refers to BGL vertex index
12     unsigned id;
13
14 };
```

Listing 3.8: Predefined property within the graph class which only provides an id member variable. This property can be used a skeleton for further property implementations.

Requesting the vertices of the graph returns a vector with all vertices of the graph. However, that vector contains all vertex properties in the context of the BGL. Each of these properties represents a mapping to the internal BGL graph. This mapping is transparent to the system user. Hence, the user does not have to care about it.

### 3.3.2 Creation of a Graph

A graph is created by a vector of edge descriptors and a vector of corresponding vertices. An edge descriptor is the tuple of source vertex, destination vertex and the connecting edge. The vector of edge descriptors can also be generated by a graph generator. Listing 3.9 demonstrates the creation of a graph from a hypercube topology with three dimensions.

```
// Definition of EdgeDescriptor
typedef std::tuple<Vertex, Vertex, Edge> EDesc;
const unsigned nDimensions = 3;

// Create vertex and edge vectors
std::vector<Vertex> vertices;
std::vector<EDesc> edges = Topology::hypercube<Graph>(nDimensions, vertices);

// Create graph by edges and vertices
Graph graph(edges, vertices);
```

Listing 3.9: Generation of a three-dimensional hypercube graph.

Figure 3.2 depicts a set of implemented generators of commonly used graph topologies. This set of generators covers fully connected topology, star topology, n-dimensional grid topology, and n-dimensional hypercube topology. These generators are parametrizable by the number of vertices and/or dimension. Communicaton topologies are usually implemented from the scratch, but these topology generator functions have the advantage that they can be reused in different applications with same communication topology.

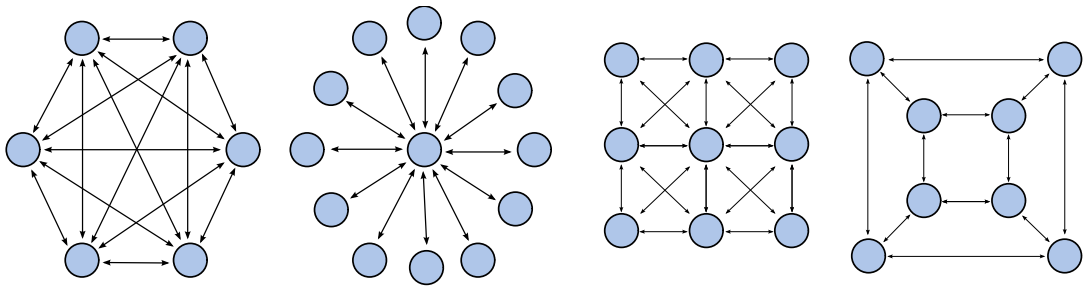

Figure 3.2: Set of already implemented graph generators. Topology descriptions can be reused in different projects.

### 3.3.3 Creation of a Subgraph

The creation of a subgraph plays an important role for collective operations on a subset of graph vertices by the GVON (Section 2.6.2). A subgraph is created from a graph by selecting a subset of its vertices. Subgraph creation is utilized as an equivalent to the creation of sub contexts in the CAL. Collective operations on this subgraph only consider vertices within this subgraph. The BGL provides already built-in support for subgraphs. The following listing demonstrates the creation of a subgraph.

```
// Creation of a subgraph
Graph& subGraph = graph.createSubGraph(subGraphVertices);
```

A created subgraph is linked to its supergraph, whereby each supergraph holds a list of its subgraphs. The connection of super- and subgraph is important for the context creation for collective operations in Section 3.4, because the announcement of a subgraph

considers first of all the context of the supergraph. Figure 3.3 depicts the creation of a two-dimensional hypercube subgraph from a three-dimensional hypercube graph.

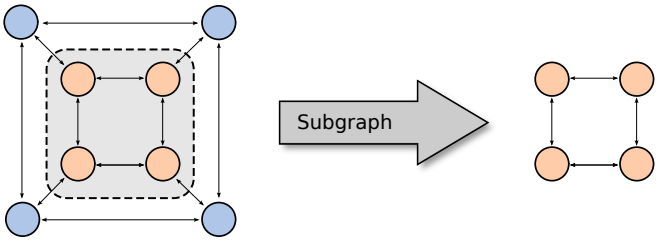


Figure 3.3: Inner vertices of a three-dimensional hypercube graph are transformed to a two-dimensional hypercube subgraph. Both graphs stay in connection by the BGL.

## 3.4 GVON Implementation

The GVON was introduced as a combination of the graph and CAL which provides communication operations on basis of the graph. This combination is the connection of graph and CAL through mappings. These mappings are core components of the GVON and are constructed by the announcement process. The announcement process implementation is discussed in Section 3.4.1. In the following, each map provided by the GVON is described:

The `vertexMap` is a mapping from a vertex to the virtual address of its host. This map is used for all methods that need to resolve the host of a vertex.

```
1 std::map<Graph, std::map<Vertex, VAddr>> vertexMap;
```

The `graphMap` is a mapping from a graph to its related context. This context is created exclusively for this graph in the announcement process. This map is important, since only the hosts of a graph are able to communicate on basis of this graph.

```
1 std::map<Graph, Context> graphMap;
```

Listing 3.10 shows the `asyncSend` method of the GVON. The vertex is translated to its host's virtual address in line 9. The graph is translated to its context in line 12.

```
// Graph-Based Virtual Overlay Network Operation
template <typename T_Data>
Event asyncSend(Graph& graph,
                const Vertex destVertex,
                const Edge edge,
                const T_Data& data){

  // Retrieve host virtual address of destVertex in graph
  VAddr destVAddr = vertexMap[graph][destVertex];

  // Retrieve context of the graph
  Context context = contextMap[graph];

  // Use CAL to send data
  return cal.asyncSend(destVAddr, edge.id, context, data);

}
```

Listing 3.10: The `vertexMap` is used to translate vertices to virtual addresses and the `graphMap` is used to translate graphs to contexts. The communication operation is performed by the the CAL.

Collective operations also utilize the GVON mappings, but their implementations are far more complex. These collective implementations will be discussed in detail in Section 3.4.2.

### 3.4.1 Announcement Process of Graphs

Peers that want to host vertices need to announce these vertices. Therefore, the GVON Interface provides an announcement method for the construction of two mappings. First, a mapping from vertices to peers of a context in the `vertexMap`. Second, a mapping from graphs to new created contexts in the `graphMap`. The method takes a pair of a graph and a vector of vertices as input arguments. It is a collective operation of all the peers that want to take part on the communication of this graph.

These peers need to create an exclusive context for themselves, which only contains hosts of the graph that will be announced. To that end, the most general context of the hosts, possibly including more peers, has to be determined. The overall most general context is the global context of all peers in the network, but in some cases there exists also a context with less peers. This is either the context of the graph or the context of the supergraph, if these graphs were already announced.

This most general context is used to gather the number of hosted vertices of each host and to create a new context that only contains peers which host at least one vertex. The graph is mapped to this new context in the `graphMap`.

Furthermore, this new context is used for announcing the hosted vertices of each host through an `allGather` operation, such that each host can update its `vertexMap` for this context.

### 3.4.2 Collective Operations on Graphs

Since hosts manage potentially more than a single vertex, a host has to perform the collective operation at first locally for its hosted vertices. This means, that the data of each hosted vertex has to be collected locally before the collective operation of the CAL can be performed. The fact that data objects can have arbitrary data types increases the difficulty of the implementation of these collective operations. In the following, the collector of a GVON collective is described:

The collector is a templated `static` object from type `std::vector<T_Data>`. The data of the hosted vertices is collected in a collector object until all hosted vertices of this host have terminated their collective operation. Such a collector will be created for each data type `T_Data`.

```
// Type dependent instanciation of the collector
static std::vector<T_Data> collector;
```

Since in some cases the resulted data is received by a root vertex, the host of the root vertex stores the receive object pointer.

```
// Receive object pointer of root host
static T_Data* rootRecvData;
```

Each call of a collective operation is collecting the vertex data of the calling host in the collector. In the case a host is calling the operation with the root vertex, the receive pointer of the root vertex is stored.

```
// Collection of vertex data
collector.push_back(vertexData);
```

If the last data object is collected, the operation is executed locally for each host and its result is forwarded to the CAL interface. The CAL executes the collective operation among all other hosts of the graph and returns the result. For collectives such as `gather`, the resulting data is reordered such that the GVON returns the data in ascending vertex id order. Finally, the result is written to the receive pointer of the root vertex.

## 3.5 Implementing a Game of Life

The Game of Life simulation was implemented to show the developed system in a real world scenario. The set of implemented rules based on [**ref:gol_rules**] is listed in the following:

1. Any live cell with fewer than two live neighbors dies, caused by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.

3. Any live cell with more than three live neighbors dies, caused by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, caused by reproduction.

### 3.5.1 Configuration and Initialization of Game of Life

The following source code describes necessary steps to configure and initialize the system before starting the communication of the GoL. The source code uses the presented MPI adapter, GoL graph, and Cell property. It demonstrates the general program flow and can be the foundation for other simulation applications.

1. **Configuration**

   a) Configure the CAL

   The target system is a cluster providing MPI as communication library. Therefore, the CAL is configured by the reference MPI adapter described in Section 3.2. The adapter provides type definitions for the virtual address, event and context that are necessary for later usage:

```
1 // Configure CAL
2 typedef CommunicationPolicy::MPI                  Mpi;
3 typedef CommunicationAbstractionLayer<Mpi>        MpiCAL;
4 typedef typename MpiCAL::VAddr                    VAddr;
5 typedef typename MpiCAL::Event                    Event;
```

Listing 3.11:

   b) Configure the graph

   The graph is configured with the vertex property `Cell` (Listing 3.13) and the edge property `SimpleProperty`. It provides type definitions for `Vertex` and `Edge`:

```
1 // Configure graph
2 typedef Graph<Cell, SimpleProperty>               GoLGraph;
3 typedef typename GoLGraph::Vertex                 Vertex;
4 typedef typename GoLGraph::Edge                   Edge;
```

Listing 3.12:

   The `Cell` property contains the state information of a cell, and inherits the vertex id from `SimpleProperty`. The following listing shows the source code of the `Cell` property.

```
// Cell property
struct Cell : public SimpleProperty {

 Cell() : SimpleProperty(0), isAlive{{0}}, aliveNeighbors(0){

 }

 // Initialization of the cell
 Cell(ID id) : SimpleProperty(id), isAlive{{0}}), aliveNeighbors(0){
   unsigned random = rand() % 10000;
   if(random < 3125){
     isAlive[0] = 1;

   }

 }

 // State of the cell
 std::array<unsigned, 1> isAlive;

 // Number of alive neighbors
 unsigned aliveNeighbors;

};
```

Listing 3.13:

c) Configure the GVON

The GVON is configured by the previously configured `GoLGraph` and `MpiCAL` (Listings 3.12 3.11).

```
// Configure GVON
typedef VirtualOverlayNetwork<GoLGraph, MpiCAL> GVON;
```

2. **Initialization**

a) Create the graph object

The graph is generated by the predefined graph generator for grids. The generator creates a two dimensional grid, where cells are connected to their vertical, horizontal and diagonal neighbors (Figure 2.11).

```
// STL namespace
using namespace std;

// Type definitions
typedef typename GoLGraph::EdgeDescriptor EDesc;

// Generate GoL graph
vector<Vertex> vertices;
vector<EDesc> edges = Topology::grid<GoLGraph>(n, vertices);
GoLGraph graph (edges, vertices);
```

b) Create the CAL and GVON objects

```
// Instantiate CAL and GVON
MpiCAL cal;
GVON gvon(cal);
```

c) Distribute the vertices

The graph vertices are distributed in round-robin fashion, which is by far not the optimal vertex distribution method, but this is not the focus of this work.

```
// Distribution of vertices by round-robin
Context context = cal.getGlobalContext();
vector<Vertex> hostedVertices = Dist::roundrobin(context, graph);
```

Figure 3.4 shows two alternative vertex distributions methods. In the first variant, hosted vertices are connected within the same host, since this reduces communication operations with other peers over a network. In the second variant, every vertex is hosted by exactly one peer, representing a case where the graph represents exactly the communicating processes.

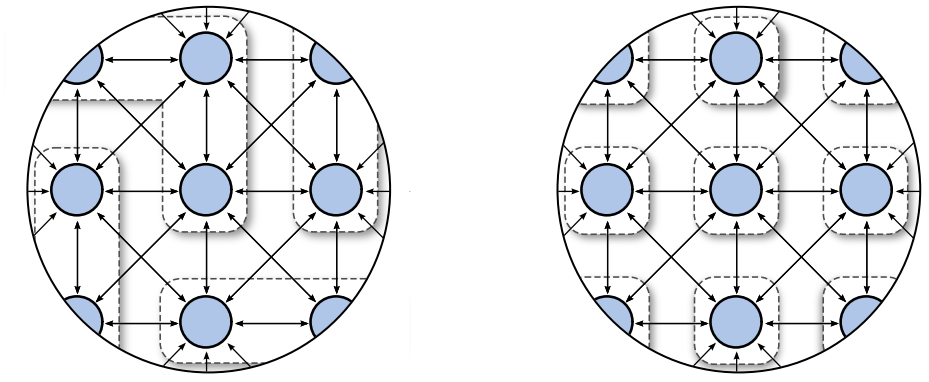

Figure 3.4: Image detail of one section of the GoL world. Possible distributions of modeled GoL graph. On the left-hand side, a peer hosts a connected set of vertices. On the right-hand side, every peer hosts exactly one vertex.

Possible distributions range from one host that hosts all vertices to the number of hosts where every host is responsible for a single vertex. Expanding this range for an execution of an application with more peers than available vertices offers an interesting case for fault tolerance and load balancing. Additional peers could be used as backup peers, but this is left open for future work. After distribution vertices, every peer announces its vertices to the GVON.

d) Announcement of the hosted vertices

```
1 // Announcement of hosted vertices
2 gvon.announce(graph, hostedVertices);
```

### 3.5.2 Communication of Game of Life

Since the utilized GoL rules only require next-neighbor communication, the GoL domain is modeled as a two-dimensional grid with diagonal connections as shown in Figure 2.11. Therefore, a GoL cell is represented by a vertex and neighboring cells are connected by edges. A cell has to communicate with its neighboring cells to retrieve the neighboring cells states and to calculate its own state for the next time-step.

In the following, the implementation of a single time-step of the GoL is described. The communication between vertices of the GoL graph is handled by the GVON. A host knows all its hosted vertices whose communication it needs to manage. In the case of GoL, a host has to exchange the state of its hosted vertices with all neighboring vertices.

First, each host sends the state of its cells to all neighboring cells (Listing 3.14). The host retrieves the target vertex of each outgoing edge for a hosted vertex from the graph. After that, the state of the corresponding cell is transmitted to the target vertices sequentially in non-blocking mode. Events of the send operation are collected and checked later for termination.

```
// Send state to neighbor cells
for(Vertex myVertex : hostedVertices){
  for(std::pair<Vertex, Edge> outEdge : graph.getOutEdges(myVertex)){

    // Retrieve target vertex
    Vertex targetVertex = outEdge.first;
    Edge   edge         = outEdge.second;

    // Send cell state to neighboring cell
    events.push_back(
      gvon.asyncSend(graph, targetVertex, edge, myVertex.isAlive)
    );

  }

}
```

Listing 3.14: A host sends the cell state of its hosted vertices to all neighboring vertices. The information of neighboring vertices is retrieved from the graph. The cell states are send in the non-blocking variant.

Figure 3.5 shows the sending process of a host for each of its hosted vertices. The host queries the graph for outgoing edges for each of its hosted vertices. The cell state of a hosted vertex needs to be send to the target vertices of its outgoing edges.

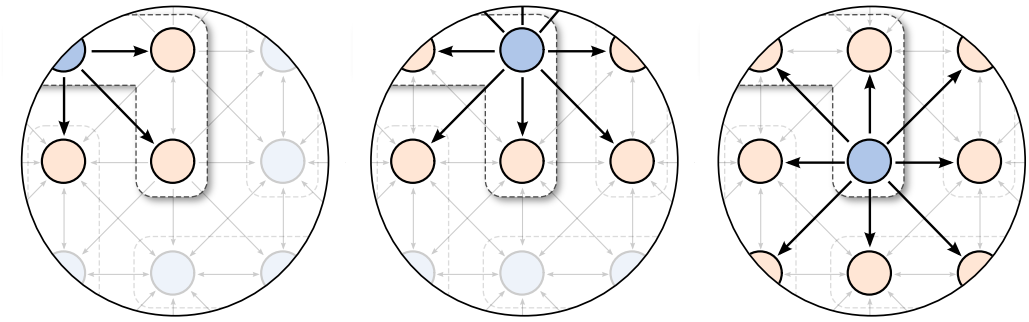

Figure 3.5: Image detail of one section of the GoL world. A host sends the cell state of its hosted vertices to neighboring vertices. This can also imply that a host has to communicate with itself.

As a second step, each host receives state information of neighboring cells for each of its hosted vertices (Listing 3.15). The host queries the source vertex of incoming edges of its hosted vertices from the graph. Afterwards, it receives the state information from this neighboring source vertex. The receive operation is used in blocking mode to ensure that all peers are synchronized afterwards.

```
// Recv state from neighbor cells
for(Vertex myVertex : hostedVertices){
  for(std::pair<Vertex, Edge> inEdge : graph.getInEdges(myVertex)){

    // Retrieve source vertex
    Vertex sourceVertex = inEdge.first;
    Edge   edge         = inEdge.second;

    // Receive cell state of the neighboring cell
    gvon.recv(graph, sourceVertex, edge, sourceVertex.isAlive);

    // Update number of living neighbors
    if(sourceVertex.isAlive[0]) {
      myVertex.aliveNeighbors++;

  }

}
```

Listing 3.15: A host receives the cell state of neighboring vertices of its hosted vertices. The information of neighboring vertices is retrieved from the graph. The cell states are received in the blocking variant to synchronize the hosts.

Once all send and receive operations terminated, each host updates the cell states of its hosted vertices according to the introduced rules. Finally, the state information of all cells is gathered by the root vertex (vertex with id equal to zero) which prints it to the console for visualization (Listing 3.16). The next time-steps will repeat the previous communication steps until the application is aborted or a fixed number of time-steps is reached.

```
// Gather state by root vertex
for(Vertex myVertex: hostedVertices){
  gvon.gather(root, myVertex, graph, myVertex.isAlive, golGameField);

}
```

Listing 3.16: The cell states of all vertices of the GoL graph is gathered by a host that is responsible for the root vertex. This host could visualize the current state of the GoL domain.

## 3.6 Implementing a N-Body Simulation

The implementation of the N-body simulation, discussed in Section 2.5.4 is very similar to the GoL implementation with respect to the GVON. Its implementation is also split into three phases: configuration, initialization and communication. The slight differences in implementation details are described in the following enumeration:

1. **Configuration**

   The graph is annotated by the property `Particle`, containing information about the particle's mass, velocity, and location.

2. **Initialization**

   Since the chosen N-body algorithm is based on an all-to-all communication pattern, the communication dependencies are modeled as a fully connected graph.

3. **Communication**

   Every host sends particle information of its hosted vertices to adjacent vertices. In return, it receives particle information from all adjacent vertices. Since a vertex is adjacent to all other vertices, it sends to and receives from all other vertices.

After each communication step, a host updates the particle state of its hosted vertices. The communication phase is repeated for an arbitrary but fixed amount of time steps or until the simulation execution is canceled.

The implemented N-body simulation is completely different from the presented GoL implementation, but uses similar communication concepts. Therefore, the source code responsible for communication in the GoL simulation can be reused to implement this N-body simulation.

## 3.7 Redistribution of Vertices

The GVON provides explicit mappings from vertices to peers and from graphs to contexts. These mappings can be modified dynamically at the run-time of the simulation application. To demonstrate that behavior of the system, an occupation scenario, where a vertex is changing its host is implemented. This is possible, since hosted vertices are not bound statically to their hosts.

The scenario is the following: a host occupies or steals a vertex from another host and henceforth hosts this vertex. This process is called occupation, because the change of the host is dictated by the so-called master host.

The occupation process starts by determining the master host of a context. The master is the result of a random number modulo the size of the context. To find a consensus random number in the context, every host generates its own random number and the consensus random number is calculated by accumulating all random numbers with a reduce operation.

Once the master is determined, it defines an occupy vertex of the graph. This vertex needs to be hosted from another host, such that the master does not occupy a vertex from itself.

The occupy vertex is broadcasted to the set of hosts in the context and every host in the context checks whether this vertex is contained in its set of hosted vertices. The

host which manages this vertex must release it, while the master host adds the vertex to its hosted vertices.

Although, the hosts know the master and the vertex that was occupied, the distribution and announcement processes are strictly separated. Therefore, the hosts need to reannounce their hosted vertices of the graph, although nothing changed for the most peers. Figure 3.6 demonstrates the occupation of a vertex. The occupied vertex does not need to be adjacent to hosted vertices of the master host.

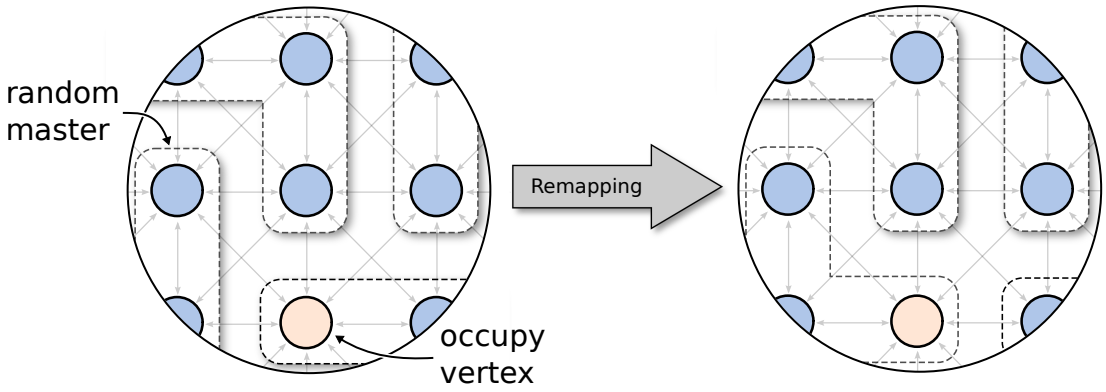


Figure 3.6: Remapping of vertices to peers. A random master host is determined by a consensus random number generation. The master dictates the vertex which will be occupied. Hosted vertices of the graph are reannounced.

Even though this occupation scenario is a simple example, it shows remapping of vertices at run-time and, therefore, the potential to establish load balancing and fault tolerance techniques on top of the system. Ideas on load balancing and fault tolerance are discussed in Section **??**.

# 4 Evaluation

This chapter evaluates the implementation of the developed system with respect to performance and flexibility. The evaluation has emphasis on determining the overhead of the CAL and GVON configured with the MPI adapter in comparison to plain MPI.

It is expected that the overhead is negligible, because of compile-time optimization. Therefore, there will be no difference if MPI is exchanged by the GVON. Such that, communication processes will be as fast as the underlying adapter implementation.

The evaluation is divided in several benchmarks and every benchmark provides several experiments. Synthetic benchmarks will evaluate single communication methods. These benchmarks meassure the pure run-time differences without the influence of simulation computations or communication overlapping. Additionally, the flexibility benchmark evaluates the behavior of the system when vertices are redistributed at run-time. Finally, real world simulation benchmarks will compare the implemented simulations in comparison to equivalent MPI implementations. For these experiments the amount of hosts, number of messages to send, number of elements to send, and usage of network will vary.

The benchmark system is the HPC system of the Helmholz Zentrum Dresden Rossendorf (HZDR)[**ref:hzdr_cluster**]. A part of the HZDR system is the hypnos linux cluster (Ubuntu 12.04.4 LTS) consisting of two head nodes and more than 150 compute nodes. The so-called laser nodes combine four AMD Interlagos Opterons on four sockets on a single mainboard resulting in 64 CPUs per node. The Interlagos Opterons 6276 are two Valencia chips on a single die, whereby two cores share a 64 KB L1 Cache and a two MB L2 Cache. Laser nodes are interconnected by an Infiniband network. These nodes are utilized for all synthetic, real world simulation, and flexibility benchmarks.

The nodes provide their software environment using the module system. The software is compiled by g++ 4.8.2 with OpenMPI 1.8.0.

## 4.1 Synthetic Point-to-Point Benchmark

This benchmark compares the run-time of fundamental point-to-point communication operations of MPI with CAL and GVON operations. The run-time overhead of the CAL and GVON is determined with respect to the plain usage of MPI. The experiemnt only contains source code which is necessary for the implementation of point-to-point communication. Therefore, it is free from any application dependent logic which could influence the meassurements of the communication operations.

In particular, two peers are exchanging data, whereby one peer is the sender and the other is the receiver. The run-time is measured for sending and receiving of $n$ messages with $m$ elements. A single element is set fixed to a 64 Bit integer. This experiment will

be called send/recv operation in the further benchmark description. Every send/recv operation is executed a thousand times and averaged afterwards to reduce variations in the run-time measurement. An experiment configuration has the following parameters:

- Number of consecutive send/recv operations $n$
- Number of elements per send/recv operation $m$
- Communicating with or without network

The following experiments vary a single parameter while the others stay fixed. This should evaluate the impact of this parameter with respect to the equivalent MPI implementation.

### Increase the Number of Consecutive Send/Recv Operations $n$

This experiment increases the number of consecutive send/recv operations $n$ from one to ten thousand by a step size of ten. The number of elements per send/recv operation is restricted to a single element. The run-time for these consecutive send/recv operations is measured and averaged to obtain the run-time for a single operation. The experiment is limited on a single node. Therefore, no network latency is included in the results. Figure **??** shows the averaged run-time and the according run-time ratio with respect to MPI.

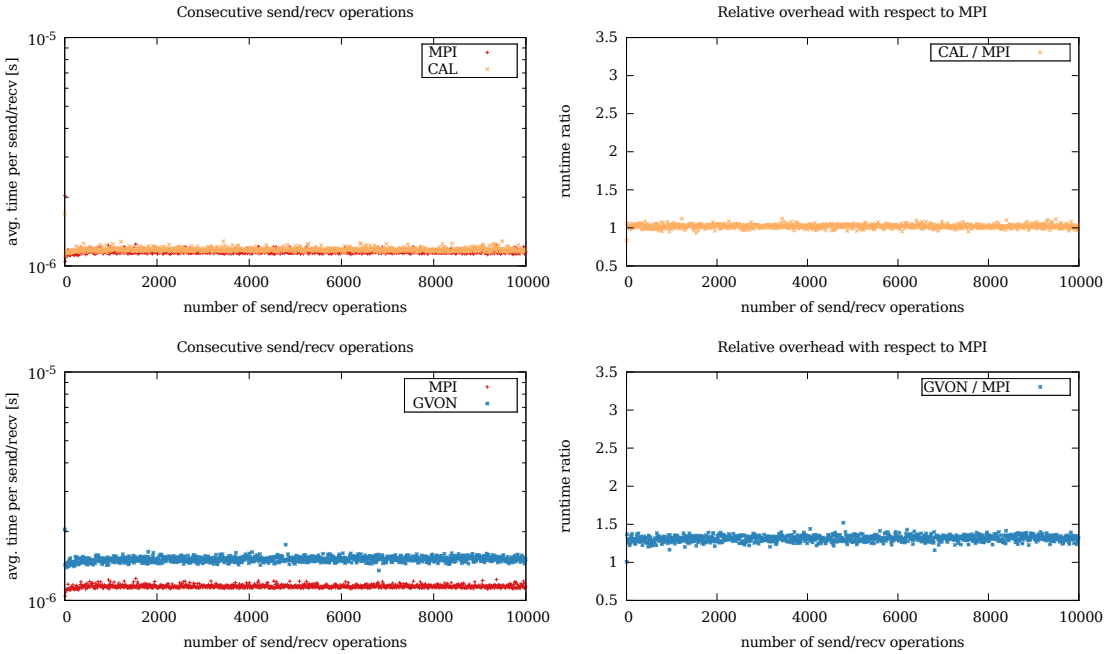


Figure 4.1: Average run-time of a single send/recv operation and the according relative overhead of CAL and GVON with respect to MPI. The number of consecutive operations is increased from one to ten thousand by a step size of ten. The CAL and GVON show a constant average overhead of around two percent and thirty percent.

The relative overhead of the CAL and GVON stabilizes with increasing number of send/recv operations. The CAL adds a constant average overhead of around two percent per send/recv operation with respect to MPI through virtual address and context translations. The GVON adds a constant average overhead of around thirty percent per send/recv operation. Since the communication topology of the GVON is modeled by a graph, containing two nodes connected by a directed edge, the look-up of this graph adds a constant overhead to each send/recv operation. Instead, the CAL and MPI are sending to and receiving from fixed peer addresses without the need for a look-up. However, because the graph is not changing during the experiment, it is sufficient to perform the graph look-up only once. Experiments performed with only one graph look-up will be named GVON OGL. The one look-up variant should reduce the overhead to the similar level of the CAL. Figure **??** shows the results of the experiment that only performs one look-up per $n$ send/recv operation.

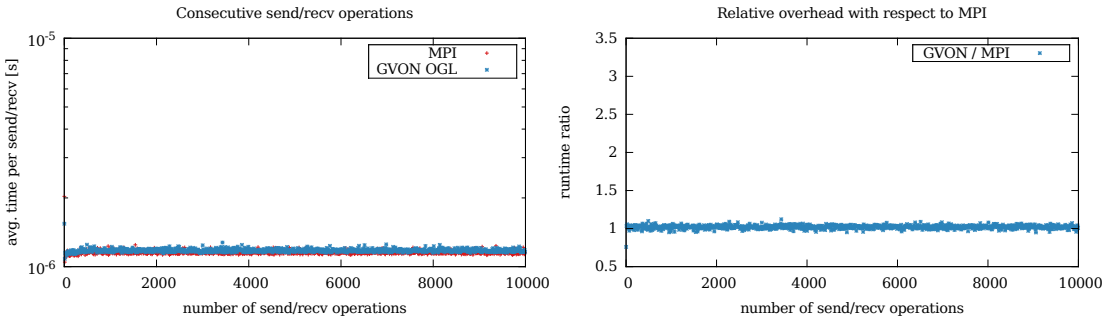


Figure 4.2: The number of graph look-ups is reduced to one look-up for $n$ send/recv operations. This reduces the GVON overhead to a similar level of the CAL.

Changing the experiment to a one look-up variant reduced the GVON overhead to the same level of the CAL (around two percent). Nevertheless, vertices and graphs need to be translated to virtual addresses and contexts, resulting in a small overhead with respect to the CAL is still present. It shows that the graph implementation needs to be optimized with regard to the look-up of incoming and outgoing edges in future work.

### Increase the Number of Elements per Send/Recv Operation $m$

This experiment increases the number of elements $m$ per send/recv operation from one to ten thousand by a step size of ten. This evaluates the behavior of CAL and GVON if more than a single element is sendet. Furthermore, this benchmark agrees more with real world simulation since these simulations usually send more than a single element.

A single experiment obtains the average run-time over thousand operations. This experiment is limited to execution on a single node. Thus, no network latency is included in the results. Figure **??** shows the averaged run-time of a send/recv operation for an increasing number element and the run-time ration with respect to MPI.

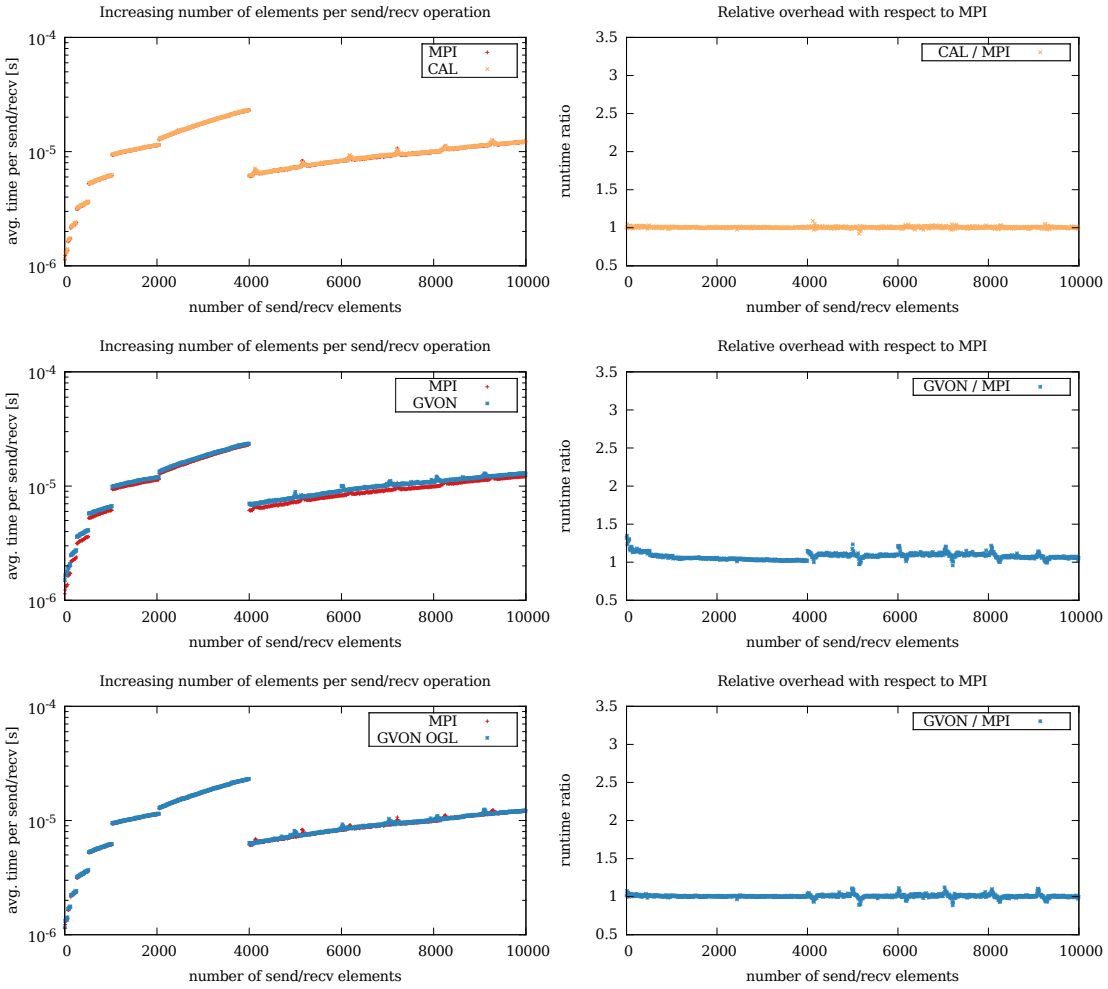


Figure 4.3: Increasing the number of elements per send/recv operations reduces the relative overhead with respect to MPI. The overhead drops to a negligible level (less than one percent).

With increasing number of elements per send/recv operations, the run-time of CAL and GVON implementations converge with the run-time of the MPI implementation. The average overhead of the CAL is negligible (less than one percent) and the average overhead of the GVON is reduced to around seven percent per send/recv operation. The last pair of figures shows, that changing the GVON experiment to a one graph look-up variant drops the GVON run-time overhead to a negligible level.

### Including Network

The synthetic benchmark with increased number of elements per send/recv operation presented above were performed only locally on a single node. Therefore, the run-time measurements were not influenced by network latency. communication over the network where added to agree more with real world simulations. the previous experiment setup

is changed to a configuration with two nodes, whereby the peers are not located on the same node. It is expected that the relative overhead of CAL and GVON decreases even more in contrast to the previous experiments, while the Figure **??** shows the average run-time of GVON and CAL with respect to MPI for increasing send/recv operations including network latency.

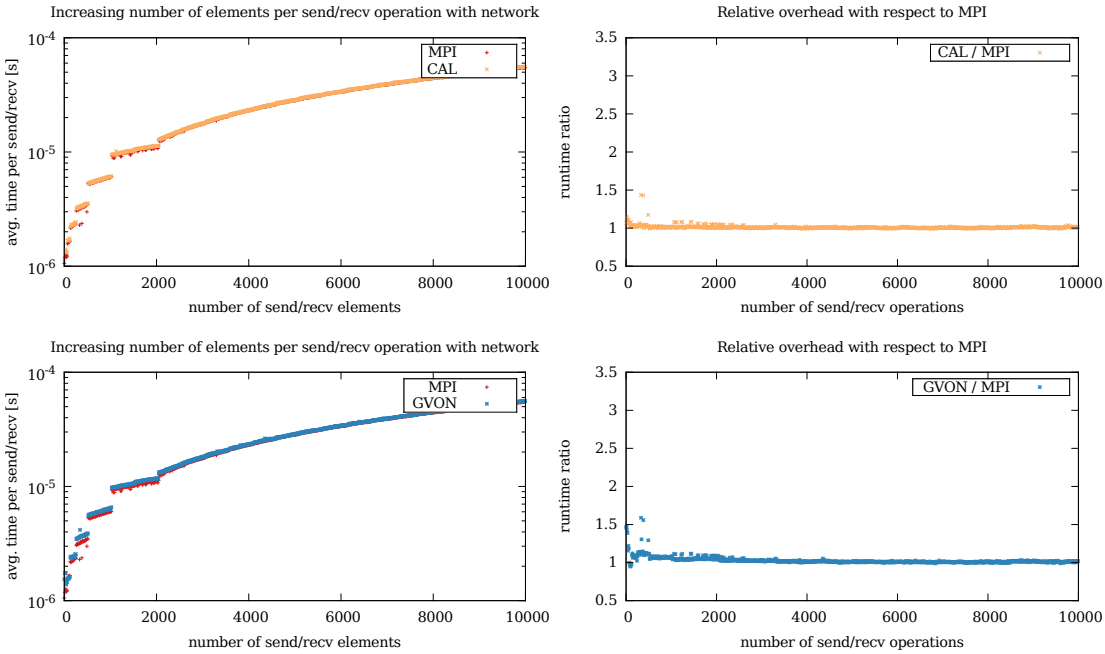


Figure 4.4: Increasing the number of elements per send/recv operations with the influence of network latency. An overhead with respecet to MPI is hardly recognizeable.

This experiment resembles a real world simulation more closely, because this simulation usually send more than a single element per communicate operation and communicates over a network. The CAL and GVON provide a constant overhead that does not scale with the size of the application and is in an acceptable range (a few percent). Therefore, the GVON can be considered for the deployment in real world simulations.

## 4.2 Synthetic Collective Benchmark

This benchmark compares the run-time of MPI collectives with equivalent collectives of the CAL and GVON. These operations are evaluated additionally to point-to-point operations because the GVON performs local data collection first, which might be a source of overhead. To determine the run-time for a single collective execution, the measurement is averaged over a thousand executions. Since the overhead with respect to MPI should be determined, only the gather and reduce collectives were chosen for comparison. Other collective operations behave in an analogous manner. These experiments will be executed

on a setup with 8 laser nodes, while the number of peers per nodes will be increased from one CPU to a maximum of 64 CPUs. This will lead to maximum number of 512 CPUs.

### Gather Collective

This experiment evaluates the gather operation of MPI in comparison to the CAL and GVON gather operations. The GVON gather operation is first performed locally for all hosted vertices of a host. Furthermore, a reordering of the gather results in vertex index order is performed. An overhead with respect to MPI is expected. Figure **??** shows the average run-time of a gather operation for increasing number of peers including network latency.

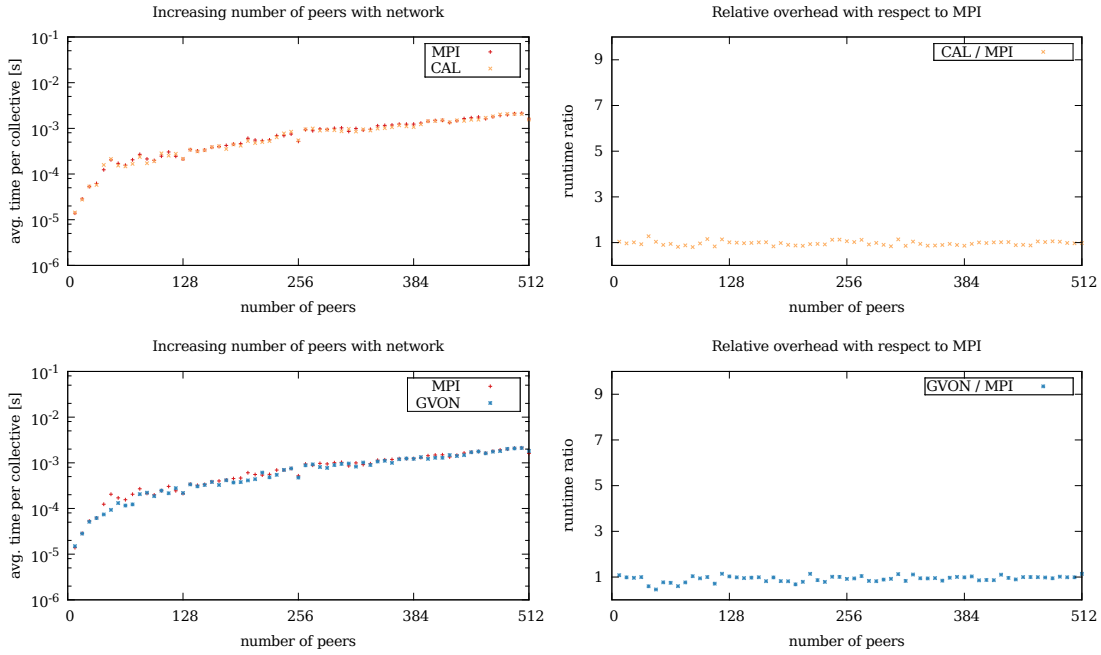


Figure 4.5

Differently than expected, the run-time overhead of the CAL and GVON is negligible or not even present. Collective operations of the GVON do not require a look-up of graph information. Thus, a source of overhead mentioned in the previous benchmarks is not present on this experiment.

### Reduce Collective

This experiment evaluates the reduce operation of MPI in comparison to the CAL and GVON reduce operations. The reduce operation of the GVON performs a local reduce first. Figure **??** shows the average run-time of a reduce collective for increasing number of peers with and without network.

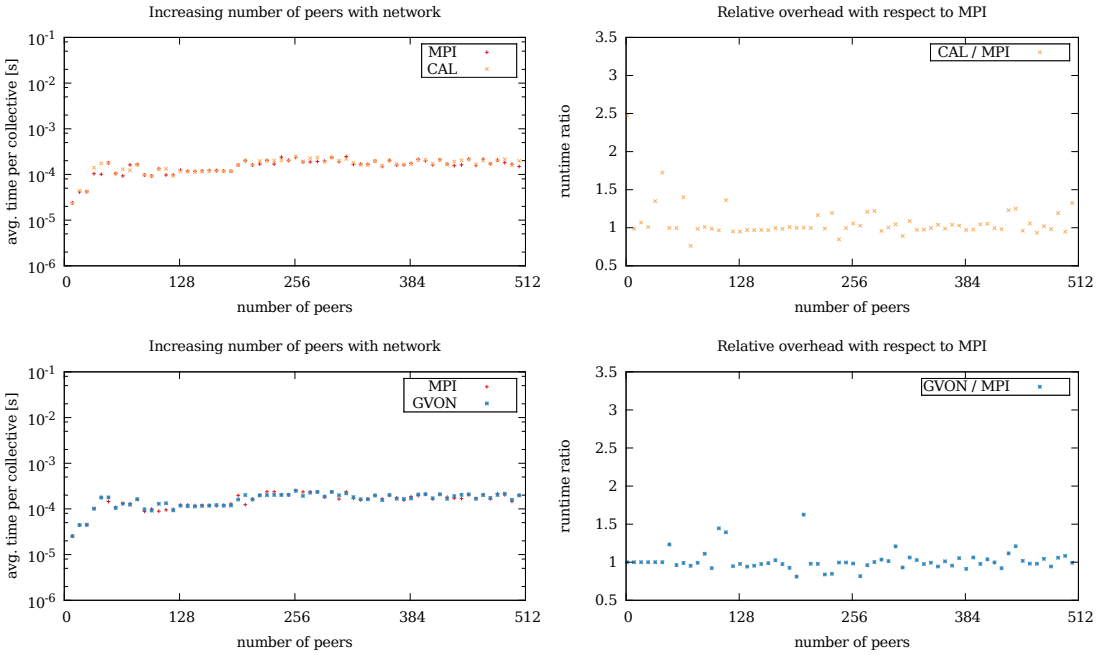


Figure 4.6

This experiment behaves similar to the gather experiment with the exception of some outliers. These outliers are normal in such a network-based environment and therefore need not to be discussed or analyzed. But even with the additional abstraction and local collection of data the GVON collective operations show negligible overhead with respect to MPI.

## 4.3 Flexibility Benchmark

This benchmark analyzes the impact of redistribution of hosted vertices to a varying set of hosts at run-time. It demonstrates the flexibility of the system by remapping with respect to load balancing and fault tolerance. A simulation based on the GVON is executed by a number of hosts $h$. After a period of time steps $t$, $h$ is varied.

The Peers are distributed equally over a set of laser nodes and every peer hosts approximately the same number of vertices.

### 4.3.1 GoL Flexibility Benchmark

The GoL simulation described in Section 3.5 forms the basis of this experiment. The simulation is started with 256 peers and the game field contains 8000 cells. The vertices of the modeled GoL graph are distributed to the peers using a round robin algorithm. The number of hosts is decreased by one every ten time steps until the number of hosts is equal to one. From this point the experiment increases the number of hosts by one every ten time steps until the number of peers is equal to 256. Figure **??** shows the average

run-time of a single GoL time step whereby the number of hosts are varied. The time for redistribution and reannouncement of the vertices are included in the measurements, so that the run-time increases slightly with respect to an implementation without remapping. This increase in the run-time is acceptable since remapping is usually not performed every time step.

Figure 4.7: Average time per time step with a varying number of hosts. First, the number of hosts is decreased from 256 to one. Subsequently, the number of hosts is increased from one to 256.

Reducing the number of hosts implies that the remaining hosts are responsible for more hosted vertices. Therefore, the run-time should increase with decreasing number of hosts. But differently than expected, the average run-time decreases with the decreasing number of hosts at the beginning of the experiment. Therefore, less peers can also imply more data locality which further implies a faster access to the state of neighboring cells.

The average time per time step reaches its minimum at 64 hosts. This point defines a distribution of vertices where the hosts are saturated with vertices. Decreasing the number of hosts even more only increases the run-time of a time step. The run-time maximum is reached when the whole simulation is calculated by a single host. Increasing the number of hosts from a single host back to 256 hosts shows symmetrical behavior. The remapping features of the GVON can be used to optimize the distribution of the GoL graph. This can be utilized to address both problems of load balancing and fault tolerance.

## 4.4 Real World Simulation Benchmark

The previous benchmarks evaluated the developed system in a very synthetic and unreal fashion. But, real world simulations usually perform calculations in between their communication operations. Furthermore, it is usual to overlap calculations with non-blocking communication operations. Experiments evaluating the Game of Life and N-body simulations were chosen as examples for such real world simulations. These experiments compare the implementations on top of the GVON from Section 3 to equivalent MPI implementations. Equivalent refers to the same amount of communication operations and that the same functions are used to calculate or update the simulation states. These simulations were selected because they base on different communication topologies. While the GoL only implements next neighbor communication, models the N-Body simulation an all-to-all communication pattern. A setup with 8 laser nodes is used. The number of peers per node is incremented for each measurement from one peer to a maximum of 64 peers. The peers are equally distributed to the nodes. Which leads to maximum number of 512 peers.

### 4.4.1 Game of Life Benchmark

This experiment compares the GVON implementation of the Game of Life from Section 3.5 with an equivalent MPI implementation. The GoL domain is a rectangular field and the state of every cell is calculated exclusively by one peer. The average run-time for a time step for an increasing number of cells is measured. The run-time is measured with the influence of network latency. Figure **??** shows the average run-time of a time step dependent on the number of cells.

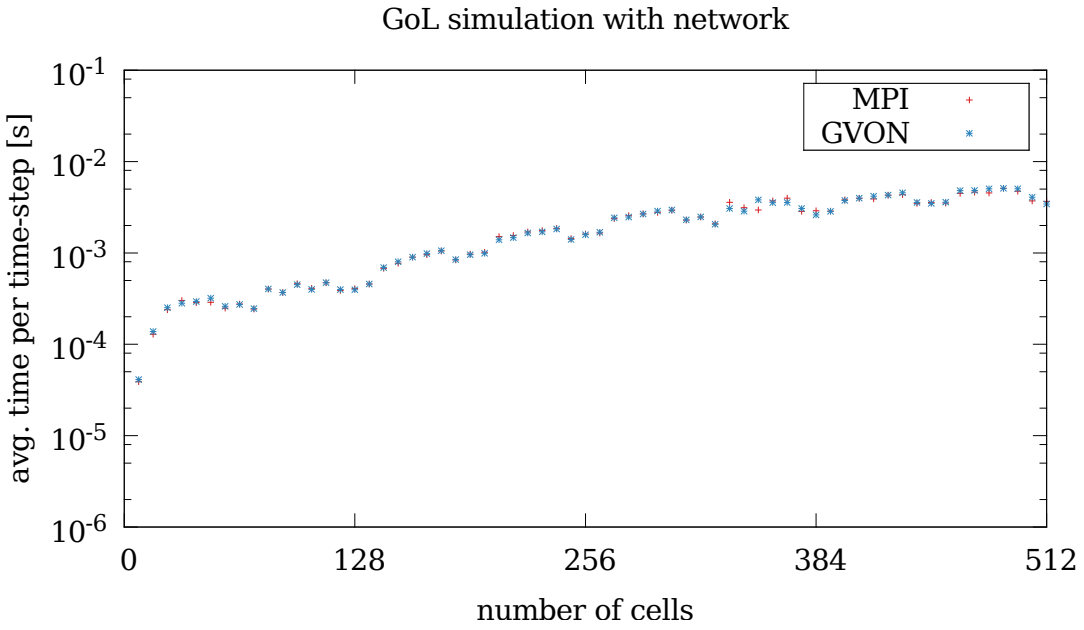


Figure 4.8: Average run-time of GoL simulation for increasing number of cells.The run-time measurement of the GVON implementation strongly agrees with the MPI implementation.

The GVON implementation shows no overhead in comparison to the MPI implementation. This behavior was expected, because communication operations are overlapped with next state calculations. Thus, the measurements have demonstrated that the GVON can be deployed into applications that are based on next neighbor communication without the expectations of run-time overhead.

### 4.4.2 N-Body Benchmark

This benchmark evaluates the implemented N-body simulation with increasing number of bodies. The N-Body simulation is in contrast to the GoL simulation based on an all-to-all communication pattern. An implementation based on MPI is compared to an implementation based on the GVON. Thereby, each body is managed by a peer. Therefore, the number of peers increases with the number of bodies. Figure **??** shows the averaged run-time of a time step with dependence to the number of bodies.

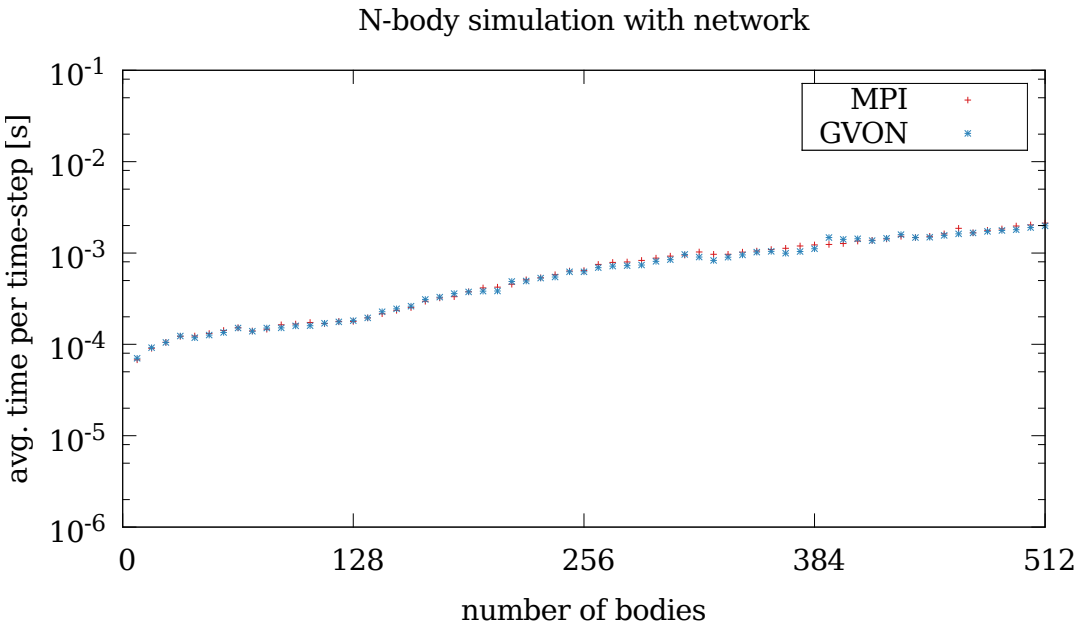


Figure 4.9: Average run-time of N-body simulation for increasing number of bodies. The simulation is executed with network. The run-time measurement of the GVON implementation strongly agrees with the MPI implementation.

Analogous to the GoL simulation shows this experiment no overhead in comparison to the MPI implementation. Both real world simulations have shown that the usage of the GVON communication approach does not add overhead to the execution of communication methods. Thus, the GVON interface is suitable for simulations based on an all-to-all and next neighbor communication pattern.

## Evaluation Summary

The evaluation of the benchmarks presented above has demonstrated the lightweight of the developed system. It was possible to perform a remapping of vertices to peers at run-time which is an important feature to solve load balancing and fault tolerance problems. The deployment into real world simulations have shown almost exact agreement with the run-time based on equivalent MPI implementations. The abstraction from an existing communication library provides negligible overhead, but flexibility to be able to perform remappings of vertices to peers at run-time. Only some extremly unreal scenarios have shown small run-time overhead. The next step would be the evaluation within a simulation such as PIConGPU.

# 5 Future Work

This chapter will first introduce ideas for enhancing the CAL by further adapters. Subsequently, ideas for load balancing and fault tolerance are presented. The discussed ideas should be a guide for further development and call attention to the possibilities of the GVON interface.

## 5.1 Implementation of Further Adapters

The prototype implementation described in Section 3 utilized MPI as communication back-end. While MPI is probably available on every computing system and implements probably every important communication protocol, other interesting communication libraries could be used as communication back-end. This section will discuss adapters based on different communication libraries other than MPI.

### Internet Socket Based Adapter

Distributed applications, that are not deployed on clusters but on the network of the internet, usually utilize communication libraries different from MPI. Two interesting libraries are ZMQ [**ref:zmq**] and Boost Asio [**ref:boost_asio**], which are both based on the TCP/IP and UDP/IP protocol stack. They can be used to interconnect applications over the internet. Therefore, computations could be based on the internet as communication network. Implementing an adapter for the CAL that interfaces with ZMQ or Boost Asio would open an application that is designed around the the GVON interface to the world of distributed computing over the internet.

Projects for distributed computing such as Folding@home [**ref:folding_at_home**] and SETI@home [**ref:seti_at_home**], utilize the computing power of computers distributed all over the world connected via the internet. These applications distribute a complex problem to a large number of peers. It would be interesting to move an application from a cluster to a distributed environment just by changing the CAL adapter (Figure **??**).

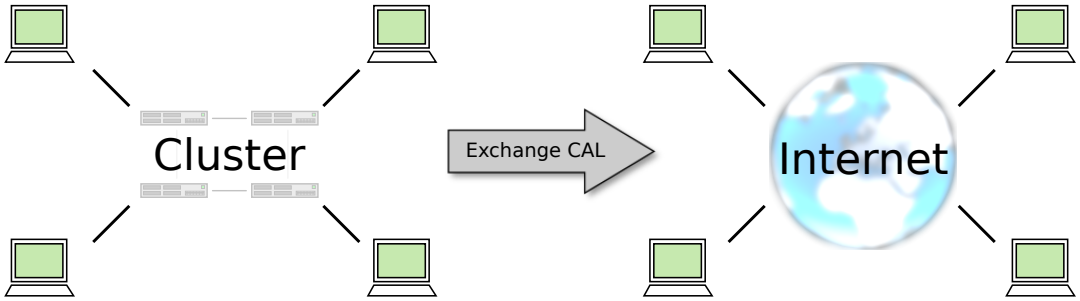


Figure 5.1: By exchanging the MPI adapter through an adapter based on internet sockets, an application initially designed for a cluster would also be executable as a distributed application over the internet.

Another use case is the connection of cluster nodes over the internet. This would construct a grid computing environment based on the GVON interface. Such an application could communicate via two CAL objects, one local CAL configured by MPI and a global CAL configured by internet sockets. This approach could extend the computational power of a cluster by further compute units located in other clusters.

### Accelerator Based Adapter

In contrast to the sender and receiver model of the communication over a network such as the internet or a local area network, the communication with and in between accelerator devices is mostly modelled in a one-sided fashion. Therefore, the accelerator needs to be managed from a host CPU. This host needs to transform the one-sided communication into a two-sided communication that is supported by the developed system. Therefore, communication processes, such as send and receive, between accelerators are managed by their hosts.

A CAL adapter could model the offloading mechanism of accelerators for CUDA [**ref:cuda**] and OpenCL [**ref:opencl**] as communication processes.

In the context of the developed system, the pair of host and device forms a peer of the CAL. Assume two computers A and B, each equipped with an accelerator, are connected via a network. Transmitting data from one accelerator A to other accelerator B (Figure **??**) is split into the following steps:

1. A sends data to B (send)
   a) Copy data from accelerator A's memory to host A's memory
   b) Send data to host B
2. B receives data from A (recv)
   a) Receive data from host A
   b) Copy data from host B's memory to accelerator B's memory

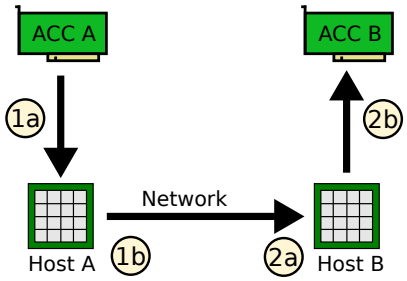


Figure 5.2: Exchanging data between accelerators. The data needs to be temporarily stored on the host's memories until the data is transmitted over a network connection.

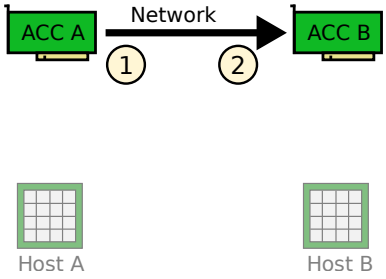


Figure 5.3: Direct exchange of data between accelerators over a network. The CPU memory is bypassed. NVIDIA provides such a technique called GPUDirect.

The adapter design for accelerators mentioned above is required to take the detour above the host CPU. However, since CUDA version 4.0 and 5.0 there exist techniques that allow to circumvent this detour. CUDA 4.0 introduced the Unified Virtual Addressing (UVA). It creates a uniform address space for the host and all devices on a single node. Utilizing UVA removes the need for explicit copies from and to accelerators.
The introduction of further GPUDirect features with CUDA 5.0 offers first the possibility for direct exchange of data between accelerators on the same node by peer-to-peer techniques. Furthermore, data can be exchanged between accelerators connected via a network by remote direct memory access. Both approaches bypass the CPU and exchange data directly over the PCI bus or network controller (Figure **??**).

## 5.2 Deployment in Real World Simulations

Section 3.5 and 3.6 have demonstrated that implementations of Game of Life and N-body simulations on basis of the GVON are possible. The performance evaluation in Section **??** has shown that there exists no significant overhead with respect to equivalent MPI implementations. Therefore, deploying the developed system into real wold simulations will be the next step.

### Deployment in PIConGPU

In the beginning of this work, PIConGPU was the first simulation whose communication processes were analyzed. However, the system was not developed for the sole objective of the utilization in PIConGPU. Therefore, it was developed independently from PIConGPU. The next step for evaluating the developed system is the deployment of the GVON as library into PIConGPU. This will evaluate how well the developed system can be integrated into existing projects.

The PIConGPU communication processes are based on a three-dimensional grid topology with diagonal connections. It resembles the communication patterns used in the GoL simulation in Section 2.5.2. Thus, a replacement of communication related parts in the PIConGPU source code seems likely.

PIConGPU is interesting with respect to an accelerator-based adapter mentioned above, because calculations in PIConGPU are offloaded to GPUs. Therefore, bypassing the CPU in case of accelerator to accelerator communication would increase the data transfer performance.

### Deployment in HASEonGPU

HASEonGPU is another project developed at the Helmholz Zentrum Dresden Rossendorf (HZDR). HASEonGPU is a Monte-Carlo simulation of photons in a laser gain medias. Whereby, the amplified spontaneous emission (ASE) of photons is calculated for a set sample points in the simulated volume of the gain media. The ASE value of each sample point can be calculated independently from each other. Thus, a distribution onto a cluster is possible. The code for communication was never analyzed, but it might be interesting if the developed communication approach would fit on HASEonGPU.

The communication scheme works as follows: a master peer distributes the set of sample points onto available worker peers. This forms a communication based on a star topology, where the master peer is placed as star center. In a scenario with high workload, e.g. lots of sample points, the master peer could be the communication bottleneck.

However, utilizing the GVON would provide the possibility to exchange this topology. The star topology could be replaced by a more general tree topology. The master peer could delegate the distribution task to sub-master peers that are responsible for sub-trees of worker peers. Figure **??** shows the graph representation of both the star topology and the tree topology.

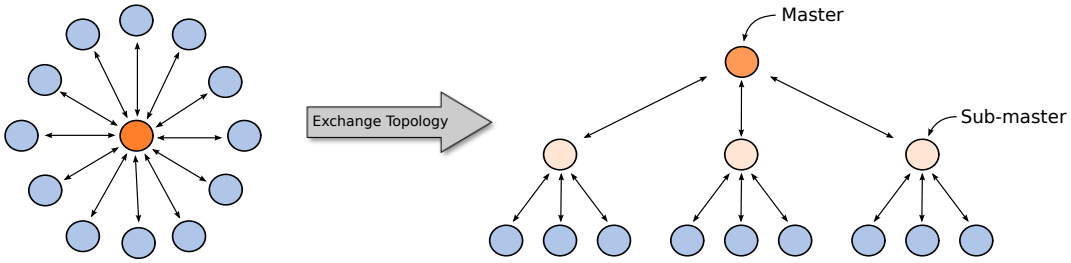


Figure 5.4: The exchange of the present star topology by a tree topology would increase the number of managing nodes that distribute workload.

The number of sub-master peers could be adapted based on the number of sample points. Thus, more sample points would lead to an increased width of the sub-master layer which leads to a better workload distribution and reduces the bottleneck effect.

## 5.3 Load Balancing Ideas

In the beginning of this work it was clarified, that load balancing will not be the focus here. Therefore, only rudimentary distribution algorithms such as round robin and consecutive distribution were implemented. But it was also claimed that it should be possible to build load balancing on top of the developed system. The following Sections will discuss ideas for load balancing based on the developed system.

### Graph Partitioning

Assume, a simulation domain is modeled by a graph and the graph contains more vertices than the number of peers in the global context. These peers need to be oversubscribed by hosted vertices. Therefore, the hosted vertices should be connected as close as possible with respect to the graph. So that communication of hosted vertices of the same host are as local as possible. This potentially minimizes the number of messages that are transferred over the network.

A well researched solution for this problem is partitioning of graphs. Partitioning divides the a graph into smaller partitions. A good partitioning algorithm reduces the number of edges running between partitions. Since the communication topology of the GVON is described by a graph, this graph can be partitioned by already existing graph partitioning tools.

### Metis - a Graph Partitioning Tool

The tool METIS and its related tools hMETIS and ParMETIS are established applications for graph partitioning. The developers claim that they provide high quality partitions, that their tools are two magnitudes faster than other widely used partitioning algorithms, and that graphs with several millions of vertices can be partitioned in 256 parts in a few seconds on current generation workstations and PCs. The tool is published under the Apache Licence Version 2.0., therefore, METIS could support the graph distribution methods as a library.

### Description of the Adapter Hardware Topology

Cluster systems are equipped with varying network systems providing a certain network topologies. Utilizing the knowledge about the network topology can increase application performance. The most general approach to describe the network topology of a cluster is a graph. A network graph describes physical connections between nodes and could be annotated with latency and bandwidth information. This graph could be the foundation of varying graph algorithm. For example could the distance between two nodes be estimated and utilized to reduce communication latency between peers.

With the existence of two graphs, one modeling the communication topology of the simulation and another modeling the network topology, a mapping between these graphs is possible. Instead a mapping from vertices to independent peers, vertices can be mapped onto peers modeled in the network graph. Information that were used to describe the network can be used to optimize this mapping.

In the oversubscribed case mentioned above, the communication graph could be first partitioned to the number of available peers. After that, the partitions will be mapped to the peers. An optimal mapping would result in an graph homomorphism. Deciding whether there exists a homomorphism from one graph to another, is NP-complete. Therefore, a heuristically approach that tries to maximize the amount of same adjacent vertices in communication and network graph should be chosen. Figure **??** shows a partitioning of a graph and a mapping of these partitions to peers.

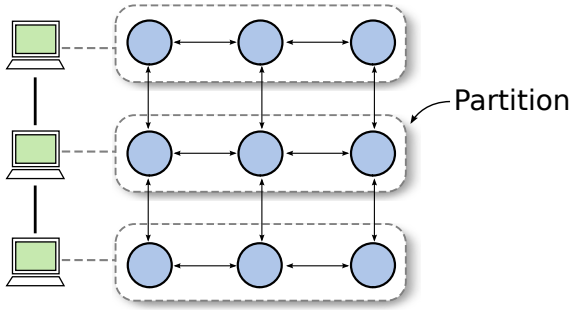


Figure 5.5: A two-dimensional grid is divided into three partitions. These partitions are mapped to the available peers with respect to their network topology. Each peer is oversubscribed by three vertices.

A mapping between these two graphs opens new possibilities for load balancing. The most simple load balancing approach is a load balancing at the initialization of an application. The communication graph is statically mapped to the network graph. This approach is sufficient for applications that do not change with respect to load distribution and graph topology.

But load distribution of simulations can change during run-time. The difference in workload among peers could be estimated by run-time measurements of a single time step. The application needs to initiate a rebalancing when the run-time of a host increases above a defined threshold.

## 5.4 Fault Tolerance Ideas

Similar to load balancing, fault tolerance of communication processes was not a topic of this work. Nevertheless, the developed system offers the possibility to implement fault tolerance techniques.

The main problem of a simulation is the failure of a peer during run-time. Some communication libraries such as MPI have the unfortunate behavior that the whole program will fail, too. Therefore, the simulation has to be restarted, if only a single peer fails. Since, the amount of computing hardware increases with each new cluster generation, it is not likely that a peer, host for vertices of a simulation, fails. The library user has usually no possibility to restart the failed peer, but the failed peer could be replaced or even ignored in further communication processes.

Assume, that each host checkpoints the state of its hosted vertices at fixed points of the simulation execution. This checkpoints could be written to hard disk of an network attached storage or each host could distribute the data of its hosted vertices to other host for backup reasons.

The network of hosts needs to notice the failure of hosts. Furthermore, the hosts need to know where a failed host stored its last saved state. Some host has to adopt the hosted vertices of the failed host. This so-called adopter host has to load the saved states of the adopted vertices before they will be announced together with his already hosted vertices. If the state of the loaded vertices is not up-to-date, other hosts need to roll back the state of their hosted vertices, too. When all vertices are synchronized to the same time step, the communication and ,hence , the simulation algorithm can be continued.

Redistribute the hosted vertices of a failed host to the remaining hosts, increases their utilization. Alternatively, the vertices could be adopted by a backup peer. Similar, to the previously mentioned method, the backup peer needs to load the saved state from its adopted vertices.

# 6 Conclusion

A system was developed with the requirements for an explicit modeling of the communication topologies of a simulation and an explicit mapping of these topologies to the hardware topologies of the computing system at run-time. This is necessary since modern simulations show heterogeneous behavior both in horizontal and vertical direction.

To address these heterogeneities, the developed system provides modeling and mapping through an intermediate layer in between a simulation application and an existing communication library. This layer maps common communication processes to an existing communication library and is build from three components: First, the communication abstraction layer that provides an general communication interface for existing communication libraries which are exchangeable through adapters. Second, a graph that models the communication topology of a simulation. Third, the graph-based virtual overlay network that provides a mapping from the modeled graph to the CAL at run-time. Furthermore, the GVON enables to utilize the communication topology modeled by the graph to provide communication operations on basis of this graph.

Policy based design allows the CAL to exchange the utilized communication library by template programming, which offers flexibility and minimizes the run-time overhead by compile-time optimizations. The Boost Graph Library was utilized as back-end to implement the graph class. This library provides a rich set of graph algorithms and comfortable deployment into C++ projects.

The system offers the advantage that the CAL can be adapted to the utilized computing system by exchanging the underlying adapter. The graphs can be generated from predefined graph generation functions, which provides the possibility to reuse these graphs. Furthermore, vertices of these graphs are explicitly mapped to peers of the CAL within the GVON at run-time. This mapping can be optimized to exploit maximum performance and be modified at run-time if this is required during execution.

An adapter based on MPI was implemented and tested in synthetic, real world, and flexibility benchmarks. The evaluation of these benchmarks turned out that the overhead with respect to an MPI implementation is negligible. The system has demonstrated the variation of the number of peers during run-time for the execution of a Game of Life simulation also with negligible overhead. Only the look-up of graph information is in some extreme corner cases a source of overhead.

The evaluation has shown that the developed system can be used to implement heterogeneous applications, load balancing and fault tolerance mechanisms. This was achieved through a consistent concept which allows to model an application by an abstract description. This description is provided by a very lightweight, yet powerful system.

# 7 Outlook

The developed abstract interface shows negligible run-time overhead with respect to the reference communication library MPI. The deployment into the previously mentioned simulation PIConGPU is the next step. The GVON interface would replace the existing communication code completely and furthermore provide the ability to exchange the underlying communication library and the utilized communication topology.

The implementation of further adapters would expand the field of application. So would an accelerator-aware adapter allow for direct data exchange between accelerator devices without the detour of the CPU. An adapter based on internet sockets would provide a distributed computing environment similar to grids.

Load balancing and fault tolerance are important topics for the upcoming exascale systems. Especially the remapping feature at run-time with respect to workload and energy consumption would be reasons for a deployment which allows for a scaleable utilization of the hardware ressources. All these problems are solved through a unified description which clarifies their similarities and therefore reduces the complexity of such problems significantly.