# Congestion-aware Simulation of Large-scale HPC Networks

**Bachelor Thesis**

**Author(s):**
Maag, Daniel

**Publication date:**
2016

**Permanent link:**
https://doi.org/10.3929/ethz-a-010740182

**Rights / license:**
In Copyright - Non-Commercial Use Permitted

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor Thesis

# Congestion-aware Simulation of Large-scale HPC Networks

| | |
|---|---|
| Student: | Daniel Maag |
| Student ID: | 12-920-740 |
| Supervisor: | Timo Schneider |
| Professor: | Dr. Torsten Hoefler |
| Submission date: | 16.09.2016 |

## Abstract

The goal of this work is to combine the two well established network simulation frameworks LogGOPSim and Omnet++. We combine them into one simulator by embedding the Omnet++ simulation kernel into LogGOPSim. This way LogGOPSim simulates the application layer but delegates the network simulation to Omnet++. We show how this cooperation can work, i.e., how LogGOPSim sends and receives messages over Omnet++ and how Omnet++ can reliably estimate the time of arrival for a message sent thought the network.

We then try to determine the influence of congestion on HPC applications by extracting application traces from the NAS Parallel Benchmarks and then simulating them. We run these traces on two simulators: on the LogGP model in LogGOPSim and on the new simulator we built. We see that for most tested application traces congestion has a minor influence on the finish time of hosts in the network.

# Contents

# 1 Introduction

High performance computing (HPC) plays an increasingly important role in computer science and various other areas. HPC helps drive new science and is therefore an active field of research that aims to steadily improve the performance of HPC clusters. Each year outstanding achievements in high-performance computing are awarded the Gordon Bell Award.

A HPC network connects a cluster of many processors that perform distributed tasks in parallel. Since such a cluster performs tasks in cooperation, the nodes need to communicate frequently in order to synchronize, for example to distribute the task or to share results. But because connecting all the nodes directly to each other is not feasible for larger HPC clusters, building a network capable of handling the traffic is important. We want the communication between nodes to be as fast as possible, since the communication is the performance and scaling bottleneck. To minimize the communication time, various kinds of network communication standards are used. Examples of these include Infiniband or Gigabit Ethernet.

In order to obtain quantifiable and comparative figures testing and benchmarking needs to be employed. However this is hard to do with different network topologies and different communication protocols within a physical system. Because many deciding factors differ depending on the hardware used, for testing different networks one would need to exchange many physical components of the corresponding networks. This implies a big effort for every possible configuration one might want to test. To avoid this effort, a simulation of the HPC network is a possible solution. When simulating, parameters like the used communication standard can be chosen freely without requiring any additional big effort. This concludes the main motivation behind and the importance of HPC network simulations, allowing the simple testing of existing and new communication protocols in different network topologies.

## 1.1 Organization

In this work, we build a new simulator by combining the OMNET++ and LOGGOPSIM frameworks. In Section 2 we first introduce LOGGOPSIM and after that OMNET++, followed by INET. We also show roughly how the existing LOGGOPSIM implementation simulates a network and what assumptions and simplifications are being made. Next, Section 3 describes how OMNET++ is embedded into LOGGOPSIM and how we can send a message through the network and estimate the time at which it will reach its destination. Further in Section 3.2 we demonstrate the installation of the simulator on a Ubuntu system and denote how to run a simple example simulation. Subsequently Section 3.3 shows that the simulation is indeed exact by comparing simulated transmission times to actual calculated transmission times. After that in Section 4 we will compare our simulator with the already existing `LogGP` model in LogGOPSim. This will be achieved by running real parallel application traces in both simulators and comparing the results. We produce the application traces with the NAS Parallel Benchmarks [1, 7]. In Section 5 we shortly introduce a Master thesis that is comparable to this work. The author V. Zorin also combined LogGOPSim and Omnet++, but instead of embedding one into the other, both are being run as separate processes communicating over IPC. Finally, in Section 6 we summarize our results and discuss the benefit of this work and its simulator.

## 1.2 Contribution

We embed the Omnet++ [8] simulation kernel into LogGOPSim [4]. This allows LogGOPSim to delegate the network simulation to Omnet++. Instead of approximating transmission times, the new simulation calculates exact transmission times by simulating the actual Ethernet network. Additionally the new simulator simulates queues in modules and packet drop when such queues get flooded.

This more accurate performance model for Ethernet comes at a cost. The new simulator is much slower than the existing LogGOPSim. We only run application traces for networks with 64 nodes or less reliably and in reasonable time.

# 2 Background

In this section we briefly explain the functionality and features of LogGOPSim [4] and Omnet++ [8].

## 2.1 LogGOPSim

LogGOPSim is a simulator framework to simulate large-scale networks that perform parallel algorithms. It utilizes the LogGOPS [4] model, which is an extension of the LogGPS model [2]. While LogGOPS allows accurate modeling of parallel algorithms in a congestion free network, LogGOPSim even goes a step further and additionally allows to simulate a simplified network with fixed routing and congestion. LogGOPSim has two models regarding the network, `LogP` and `SimpleNetwork`. `LogP` assumes a fully connected bidirectional network without any congestion. `SimpleNetwork` on the other hand uses hosts and switches to build and simulate a network. A more in depth view of `SimpleNetwork` is provided in Subsection 2.3.1.

## 2.2 Initialization

Here we are mainly concerned with a simulation in LogGOPSim in which a simple network is provided. In order to simulate an algorithm with LogGOPSim, we therefore need to specify the overall network topology and the operations of each respective node.

Chapter 2 in the paper on LogGOPSim [4] tells us that every parallel application can be modeled as a succession of tasks. These tasks are: send, receive, and calculate, the last of which can be modeled as a delay. The operation of each computing node in the network can therefore be characterized by a directed acyclic graph of tasks it has to perform. This graph defines a happens-before relation on all operations in a node. All these graphs of all the nodes combined are expressed in a single Group Operation Assembly Language (GOAL) file.

Listing 1: Example of DOT file

```
digraph Network {
  H0 -> S0 [comment="*"];
  S0 -> H0 [comment="H0"];
  S0 -> H1 [comment="H1"];
  H1 -> S0 [comment="*"];
}
```

Meanwhile the network topology is defined by a file in the graph description language DOT. With each direction of a connection between a pair of nodes, a simple routing table is included. For example for two nodes `H0` and `H1` that are connected with a switch `S0`, we have the dot file shown in Listing 1.

Instantiated with both of those files, LogGOPSim first builds the network based on the DOT file and then starts to simulate the application trace described in the binary GOAL file.

## 2.3 Simulation Loop

The simulation works with a priority queue that contains all executable operations in the order of their earliest starting time. An operation is executable if it is not dependent on any other operation that is not already finished. In each step of the simulation, we take the next operation from the queue and execute it if and only if the resources are not currently used by another operation. Otherwise, the operation is reinserted in the queue with a new earliest starting time. The new starting time being the earliest time the resources become available. Here by resources we mean the CPU and the NIC.

Arrivals of messages at their destinations are receive operations. When a message is sent, a receive operation is inserted with the expected time of arrival at its destination. A message can be delayed by congestion. In this case, the arrival operation of this message has an outdated arrival time and has to be updated. When this operation is taken from the queue, we reinsert it with an updated time of arrival.

### 2.3.1 Network

LogGOPSim implements the network and its usage in the header file `Network.hpp`. There, the parsing and construction of the network as well as the sending and traveling of messages in the network are implemented. `Network` is a dispatcher class with two possible kinds of networks; `SimpleNetwork`, which builds the network from a DOT file and `LogGP`, which simulates a fully connected, congestion free network. The simulation core interacts with the `Network` class via two functions, `insert` and `query`. We are mainly interested in the `SimpleNetwork` for the implementation, but we will use the `LogGP` model later in Section 4 to compare the congestion free implementation to our new model. More information about the `LogGP` model can be found in the paper about LogGOPSim [4].

`SimpleNetwork` builds the network with the hosts, switches, and the routing information included in the DOT file. When a message gets sent though the network it does not make one hop at a time, but rather occupies the whole path until it is fully received. The bandwidth of this link is then shared with all other messages on this link.

**Insert** The simulation core calls this function for every send operation. It generates a database entry that represents the new message to inject into the network. First, the path of this message is determined. Then, the algorithm checks along the path for the highest congestion, i.e., the most messages using a particular link. With this highest congestion along the way we can calculate the time the message needs to reach the destination. The Insert returns this estimated time of arrival for the simulation core to insert a receive operation into the queue of future events. When traversing the path, the algorithm updates the database entry for every message using a link of the path. Ever since the last update of a particular message, the congestion for this message did not change. Therefore, the remaining bytes can easily be calculated. After that, the maximum congestion is increased if necessary.

**Query** The simulation core calls this function for every receive operation from the queue. It checks the status of a message; If it has just arrived, the simulation core is notified of the arrival by returning the current time. If the message is still on the way, query calculates and returns the estimated time at which the message will reach its destination. The query should not overestimate the arrival time, i.e., the current time can never be smaller than the time at which the message arrived.

For every arriving message we also have to free any congestion caused by it. The algorithm does this by once again traversing the path and updating all the entries for messages using a link on the path. For every message it calculates the remaining bytes to transfer and the new maximum congestion.

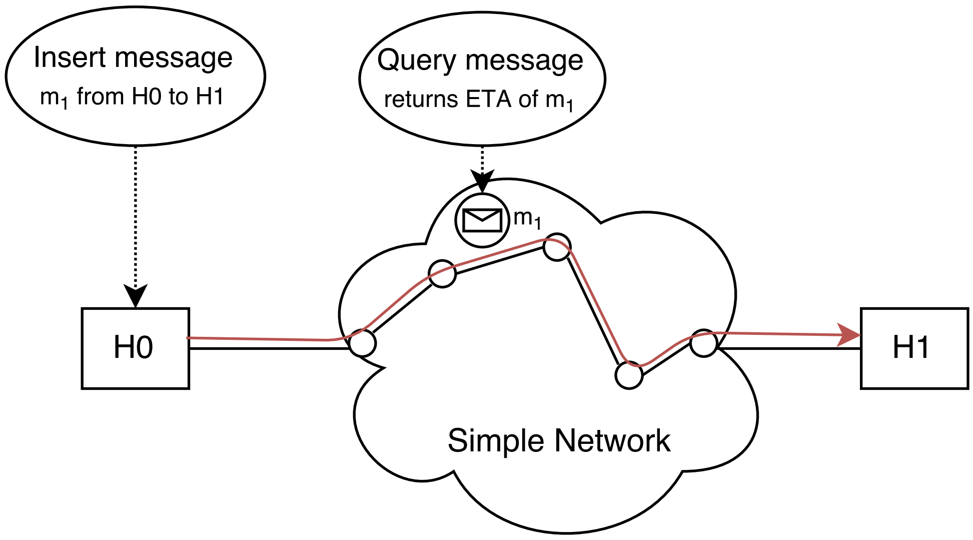


Figure 1: Insert and Query

## 2.4 Omnet++

OMNET++ [8] is a simulation library and framework, primarily for building network simulators. OMNET++ has different components such as the simulation kernel and the NED language. With these and some of the utility classes provided by OMNET++, the user can define simple modules in the higher-level language NED. These simple modules are the basic building block of any simulation and are the only modules with C++ code behind them. They describe the inner working of a part of the network, e.g., the exact functioning of a MAC protocol; How it handles an incoming message or packet etc. Modules have gates, which allow them to be connected to other modules and communicate with them by message passing. Besides the simple module there is a second kind of module, the compound module. They describe modules consisting of other modules. Compound modules are solely described in the NED language and have no C++ code by themselves. They are fully described by the code from the simple modules they contain. For example a compound module could describe a host computer with a gate for the Ethernet cable plug, a compound module describing the functioning of the NIC and a compound module describing the CPU.

This structure where everything is a module lets us describe an entire network as a compound module like the listing 2 shows. Channels are the connecting object between two gates. They allow to model parameters like transmission rates, propagation delays and other properties of physical links.

Listing 2: Sample of NED network definition

```
// imports omitted
network OmnetNetwork {
    types:
        channel C extends DatarateChannel {
            delay = 1ns;
            datarate = 100Mbps;
            per = 0.0001; // packet error rate (PER)
        }
    submodules:
        H0: LogGOPSimEthHost { }
        H1: LogGOPSimEthHost { }
        S0: LogGOPSimEtherSwitch {
            gates:
            ethg[16]; // number of ports of switch
        }
    connections allowunconnected:
        H0.ethg <--> C <--> S0.ethg[0];
        S0.ethg[1] <--> C <--> H1.ethg;
}
```

For the traditional use of OMNET++ one only needs to define the used modules in the NED language, the corresponding C++ code for all simple modules, and an initialization file (.ini). Another possibility is to embed the OMNET++ simulation kernel into another application.

### 2.4.1 Inet

The INET Framework [5] is the protocol model library of OMNET++. It defines modules for various protocols and network components, including the wired link layer protocols for Ethernet that are important for this work. INET encourages the use and modification of the existing modules in INET to build the desired network.

# 3 Implementation

The goal of this part is to simulate the network traffic with OMNET++ but let LOGGOPSIM simulate the application layer of each calculating node in the network. To achieve this separation, we embed the simulation kernel of OMNET++ into the existing simulation of LOGGOPSIM. A newly introduced class named `OmnetSimulation` serves simultaneously as the new simulation core of OMNET++, and as the interface between LOGGOPSIM and the network in OMNET++.

We use communication over Ethernet as basis for the simulation, since INET offers implementations for most of the modules in an Ethernet network. Moreover, we implement an application for the hosts and change the existing implementation for the MAC layer protocol for both the switches and hosts.

## 3.1 Algorithm

As seen in Section 2.3.1, LOGGOPSIM already has a network implementation. We extend `Network.hpp` for our needs by adding a new class `OmnetNetwork` which works similar to the existing `SimpleNetwork` class. More precisely, `OmnetNetwork` also defines an `insert` and `query` to send and receive messages to and from the network. In order to let OMNET++ simulate the network and all the traffic in it, we need three basic functionalities:

- initialization of the network in OMNET++
- sending messages from and to specific host
- keeping track of sent messages until they arrive at their destination

The last of these is needed to accurately predict the time at which a message will arrive at its destination. In this section we look at each of the three functionalities in more detail.

**Data Structure** We introduce a class `PktInfo` (definition in listing 3) that stores all relevant data for a single message. LOGGOPSIM could send far larger messages than the maximal Ethernet packet size allows. Therefore, a message is potentially split up into multiple packets, each identified by a sequence number. All the sent messages are then saved into two maps, one for the fully received messages and one for the messages currently in the network. The code snipped in listing 3 defines part of `PktInfo` and gives access to the two maps, which are defined in `omnetinterface.cc`. This code can be found in the header file `PacketList.hpp`.

Listing 3: Class PktInfo

```
using namespace std;
class PktInfo {
  public:
    // ...

    // following maps have the sequence number as key
    // <seqNo, map<modulename, t> >
    // where t = earliest departure at this module
    map<long, map<string, simtime_t> > travelingpkts;
    // <seqNo, TimeofArrival>
    map<long, simtime_t> arrivedpackets;

    // maximal eta returned to LogGOPSim
    simtime_t earliestarrival;

    // #segments LogGOPSim message got split up into
    int numberofsegments;
    // message ID from LogGOPSim
    const uint32_t logmsghandle;

    // ...
};

// traveling and finished packets
// in both maps LogGOPSim ID as key
extern map<uint32_t, PktInfo*> pktmap;
// LogGOPSim simulation time message finished
extern map<uint32_t, uint64_t> finishedpktmap;
```

### 3.1.1 Initialization of the Network

During the start-up of LogGOPSim the supplied DOT file will be parsed and stored to a graph. We use the existing parser in the class `TopoGraph` in `Network.hpp` and extend it the following way: We assume this graph to be connected but otherwise arbitrary, e.g., the graph could have multiple edges between two nodes or other types of loops. In Ethernet routing loops can introduce broadcast storms. For that reason we remove any potential loops from the graph by using Kruskals algorithm [6]. The resulting spanning tree needs to be parsed to a module definition in NED. As an example the DOT file in listing 1 would get parsed to the NED file in listing 2. This file then serves as input to the function `initialize_simulation` in

`OmnetSimulation`.

Even though INET supports spanning tree protocols during simulation, we choose to pre-compute a spanning tree because this way we know the exact hop count for every node pair. This allows us to estimate the time needed for the remaining path of a packet more accurately.

After the network is parsed to a NED file type, `OmnetNetwork` invokes the function `initialize_simulation` of the class `OmnetSimulation`. This function builds the simulation based on the network provided by the previous step. Additionally, this function gathers the information needed about the hosts, i.e., the MAC addresses and OMNET++ identification numbers of each host.

### 3.1.2 Sending Messages

As described earlier, LOGGOPSIM invokes `insert` to send a message. `OmnetNetwork` then invokes `send_from_to` from the class `OmnetSimulation`. This function is responsible to find the correct host module in the simulation. Each host module has an application layer module that lets `OmnetSimulation` invoke a function to send a message to the intended destination. This function also generates and saves the relevant information into `pktmap`, as seen in the Listing 3.

### 3.1.3 Query the Message Progress

As described in Section 2.3.1, we need to be able to estimate the remaining time until a message arrives at its destination whenever this message gets delayed by congestion. To achieve this, we need to keep track of every message and where each segment of this message currently is located in the network. We extend the implementation of the MAC protocol for this. In our OMNET++ simulation, time only passes for a message in two locations: either the message is in a queue in a MAC module, or it is currently being transmitted to the next node. Both these scenarios happen in or directly after the MAC module. Therefore, by monitoring the sending and receiving in the MAC modules, we know at any point in time where each packet is. Thus, regardless whether the packet gets put at the end of the queue or it gets sent through the channel, we can predict the time at which it will reach the next module. We store this information into the `PktInfo` object of the current message as a pair of the name of the next module and the earliest time the packet will reach this module. As soon as this information is saved, the current module is deleted in the data structure. This works since all the MAC modules of a single node process a packet at the same simulation time. Therefore a query command cannot occur between the deletion of the current module and the insertion of the next module.

Figure 2 serves as an example. Suppose the message $m_1$ is inserted into the network and after some time query gets called for this message. $m_1$ is between the green and orange nodes, either on the wire or in the queue of the green node. In this situation the orange nodes are stored with an estimation of the time $m_1$ will reach them. To calculate the time until $m_1$ reaches its destination, we take the shortest path among the orange nodes to estimate the remaining time.

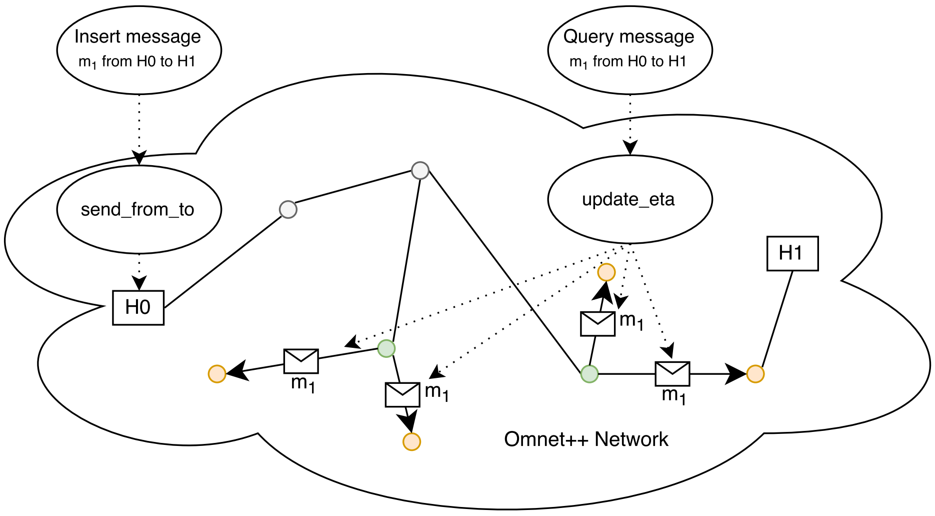


Figure 2: Omnet Network with a Message being transmitted

When `query` is invoked, `update_eta` is called which has to determine the shortest time until all packets that belong to the current message have arrived at their destination. When the message is already in `finishedpktmap`, we can return the current time to LogGOPSim to signal that the message has fully arrived. If that is not the case, we iterate over every packet of the message and find the packet that has jet the longest path to its destination. We return the time needed by this packet for its shortest path as the finish time of the message to LogGOPSim.

## 3.2 Installation

This section provides instructions for the installation of LogGOPSim [4], Omnet++ [8] and Inet [5] on a Ubuntu distribution. For other systems, please consult the installation instructions of the respective program.

These are included in the archives of each program and can be found at `LogGOPSim/README`, `omnet-5.0/doc/InstallGuide.pdf` and `inet/INSTALL`. The folder structure depicted in Figure 3 is needed for LogGOPSim to work properly. In this section we assume `/home/<user>/` to be the desired installation folder.

```
/home/<user>/
└─ LogGOPSim/
   ├─ omnetnetwork/
   │  └─ omnetinterface source files
   ├─ omnetpp-5.0/
   │  ├─ installation of Omnet++
   │  └─ inet/
   │     ├─ installation of Inet
   │     └─ src/
   │        └─ inet/
   │           └─ LogGOPSim/
   └─ LogGOPSim source files
```

Figure 3: Desired directory tree

### 3.2.1 Prerequisites

LogGOPSim, Omnet++ and Inet need several packages to be installed beforehand. Run this command to install them:

```
sudo apt-get install re2c graphviz-dev build-essential
gcc g++ bison flex perl tcl-dev tk-dev libxml2-dev
zlib1g-dev default-jre doxygen graphviz libwebkitgtk-1.0-0
qt4-qmake libqt4-dev libqt4-opengl-dev openscenegraph
libopenscenegraph-dev openscenegraph-plugin-osgearth osgearth
osgearth-data libosgearth-dev
```

Furthermore, LogGOPSim needs the package `gengetopt`, which can be found and downloaded at: `http://www.gnu.org/software/gengetopt/gengetopt.html`. Download Omnet++ and Inet from their official websites, `http://omnetpp.org` and `http://inet.omnetpp.org` respectively. In this guide, we will use the distributions `omnet-5.0-src.tgz` and `inet-3.4.0-src.tgz`. Choose a directory for the installation, e.g., `/home/<user>/`, and unpack the archives such that the structure shown in Figure 4 is created.

```
/home/<user>/
└── LogGOPSim/
    ├── omnetnetwork/
    └── omnetpp-5.0/
        └── inet/
```

Figure 4: Desired directory tree

### 3.2.2 Compilation

In this section we omit the leading `/home/<user>/` for convenience. Next, open a console, navigate to the OMNET++ root directory `LogGOPSim/omnetpp-5.0/` and add the OMNET++ folder to the `Path`. The directory can either be temporarily added with the command `./setenv`, or permanently by adding `export PATH=$PATH:$HOME/omnetpp-5.0/bin` to ~`/.bashrc`. Hereafter, in the top OMNET++ directory type `./configure` and when this script has terminated, type `make`. Use `make -j4` to compile OMNET++ using multiple cores. The OMNET++ installation can now be verified by running a sample simulation. To do this type `cd samples/dyna` and then run `./dyna`. If any problems occur, or additional information is needed, please refer to the installation manual in the OMNET++ distribution. These can be found in `/omnetpp-5.0/doc/`.

To compile INET, move the folder `LogGOPSim/omnetnetwork/LogGOPSim` to `LogGOPSim/omnet-5.0/inet/src/inet`. Then change the active directory to Inets main directory which is `LogGOPSim/omnetpp-5.0/inet`. Then type the command `make makefiles` and as soon as the makefiles generation is done, type `make` to compile the INET library.

To finish the installation, return to `/home/<user>/LogGOPSim/` and run `make`. Now `LogGOPSim` should be in the root folder of LOGGOPSIM and the simulation can be started by running `./LogGOPSim`.

### 3.2.3 Example Run

To run a test for the simulator or to see a concrete example, a sample is provided in the distribution. In the root directory of LOGGOPSIM type:

```
./txt2bin -i SharedLink.goal -o SharedLink.bin
```

This will generate a binary GOAL schedule for LOGGOPSIM. After that type:

```
./LogGOPSim --network-file="SharedLink.dot" -f SharedLink.bin -n
omnet -L 0
```

This will start a small simulation with 4 hosts and two switches. A figure of the network can be seen in the Figure 5.

## 3.3 Testing

In this section we present test cases that verify that the interface and the two combined simulations indeed yield realistic results. We compare the times measured in the simulation with a calculated reference time. The reference time is calculated manually and only considers the time consumed by OMNET++, i.e., the network. To reliably compare these two times, we use LOGGOPSIM with the parameters $o$, $g$, $G$, $L$, and $O$ set to 0. These parameters stand for the CPU overhead per message, the time gap between the sending of two messages, the cost per byte of message, and the overhead per byte of message respectively. This ensures that time only passes during the transport of a message. This way the only delay in the simulation stems from the network in OMNET++.

For all the calculations a propagation delay of 1ns and a bandwidth of 1000Mbit/s were used, unless stated otherwise. Furthermore we use the following notations:

| | | |
|---|---|---|
| Message length [bytes] | = | $m$ |
| Hops for message | = | $h$ |
| Datarate [Mbit/s] | = | $b$ |
| Delay [ns] | = | $d$ |

In the following Tables, the column Send denotes the time at which the message was sent at the originating host and the column Receive denotes the time at which the message reached its destination. In other words, Send is the time at which insert was called and Receive is the time query was called the last time for the message.

**2 Hosts 1 Switch** For this test we use the DOT file from Listing 1 and a goal file that first sends some bytes from H0 to H1 and after that the same amount of byte from H1 to H0. We first test with two different bandwidths of the network and after that for two message sizes, 1 byte and 40'000 bytes. Ethernet does not allow to send very short or very long messages. A message is therefore potentially split up and on each fragment padding and the header is added before sending it. The message sizes in the tables below are the actual sizes of the messages after these steps.

| 100Mbit/s | $m$ [bytes] | $h$ | Send [ns] | Receive [ns] | Time [ns] | $t_{ref}$ [ns] |
|---|---|---|---|---|---|---|
| H0 to H1 | 72 | 2 | 0 | 11522 | 11522 | 11522 |
| H1 to H0 | 72 | 2 | 11522 | 23044 | 11522 | 11522 |
| Round trip | 72 | 2 | 0 | 23044 | 23044 | 23044 |

Table 1: Round Trip with 100Mbit/s channels

| 1000Mbit/s | $m$ [bytes] | $h$ | Send [ns] | Receive [ns] | Time [ns] | $t_{ref}$ [ns] |
|---|---|---|---|---|---|---|
| H0 to H1 | 72 | 2 | 0 | 1154 | 1154 | 1154 |
| H1 to H0 | 72 | 2 | 1154 | 2308 | 1154 | 1154 |
| Round trip | 72 | 2 | 0 | 2308 | 2308 | 2308 |
| H0 to H1 | 40783 | 2 | 0 | 340970 | 340970 | 340970 |
| H1 to H0 | 40783 | 2 | 340970 | 681940 | 340970 | 340970 |
| Round trip | 40783 | 2 | 0 | 340970 | 340970 | 340970 |

Table 2: Round Trip with 1000Mbit/s channels

The reference time for the small messages can be calculated by the formula $t_{ref} = h * (\frac{m}{b} + d)$. In the case of the message with 40000bytes, the message gets split up into 27 frames, 26 with length 1526bytes and one with length 1107bytes. These numbers are after padding is applied and the header is pretended. Let $m_1 = 1526$ bytes and let n denote the number of message parts. Now for this transmission we get the formula $t_{ref} = \frac{m}{b} + \frac{m_1}{b} + h * d + (n-1) * \frac{p}{b}$ where $p = 96$bit. $\frac{p}{b}$ is added for every packet except the first because after a frame transmission the MAC module cannot immediately send the next frame, instead it waits for a short time and then starts the transmission of the next frame.
From Table 1 and Table 2 we can see that for this particular example the simulation is exact with variable channel speeds.

**Shared Link** In this test we look at the topology depicted in Figure 5. We look at the two following scenarios. In the first, message $m_0$ is sent from H0 to H2 and $m_1$ from H1 to H3. In the second scenario $m_1$ travels in the other direction. Since we use Ethernet connection in full-duplex mode, we expect scenario 1 to be slower than scenario 2. Like in the earlier test case, the actual size of the message after padding and with the header added is denoted by the number in parenthesis.

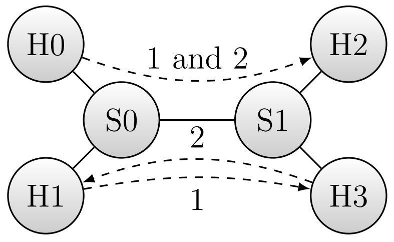


Figure 5: Shared Link Network

| Scenario 1 | $m$ [bytes] | $h$ | Send [ns] | Receive [ns] | Time [ns] | $t_{ref}$ [ns] |
|---|---|---|---|---|---|---|
| H0 to H2 | 1 (72) | 3 | 0 | 17283 | 17283 | 17283 |
| H1 to H3 | 1 (72) | 3 | 0 | 24003 | 24003 | 24004 |

Table 3: Shared Link - Same Direction

| Scenario 2 | $m$ [bytes] | $h$ | Send [ns] | Receive [ns] | Time [ns] | $t_{ref}$ [ns] |
|---|---|---|---|---|---|---|
| H0 to H2 | 1 (72) | 3 | 0 | 18242 | 18242 | 18243 |
| H3 to H1 | 1 (72) | 3 | 0 | 18242 | 18242 | 18243 |

Table 4: Shared Link - Opposite Direction

Reference time calculations: $t_{ref} = h * (\frac{m}{b} + d) + \frac{p}{b}$ where $p = 96$ bit. Like before, $\frac{p}{b}$ is added because the MAC module cannot send immediatly the next frame after the previous got finished.
In Table 3 and Table 4 we see as expected, that scenario 1 introduces more congestion to the network and is therefore slower as scenario 2.

# 4 Results

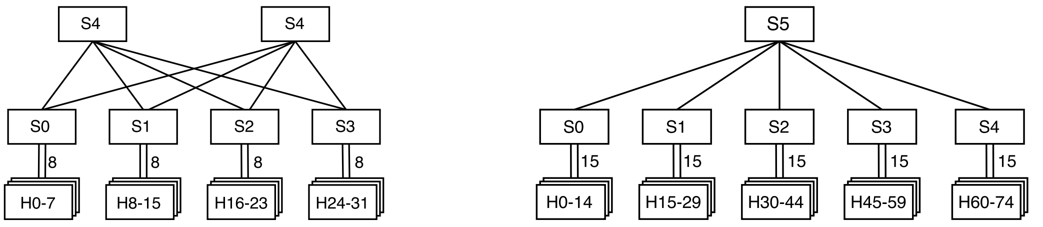
The goal of this section is to investigate the influence of congestion on HPC applications. We examine this by simulating different traces of the NAS Parallel Benchmark [1, 7] in LogGOPSim with and without congestion. To achieve this, we simulate one time with a Omnet++ network and the second time with the `LogGP` model. Since `LogGP` does not simulate any network but instead assumes all hosts to be connected directly, we need to find the suitable value for the input parameters of `LogGP`, such that a send from one host to another takes approximately the same time in the `LogGP` model as it would take traveling through the network in `OmnetSimulation`. Assuming we have such parameters, we can then compare the results obtained by the two simulations, and can estimate the influence of congestion on the application traces.

These are the parameters we need to specify when running LogGOPSim with `LogGP`:

- $L$, the maximal latency among two processors in the Network
- $o$, the CPU overhead per message
- $g$, the time between two message injections into the network
- $G$, the cost per byte of messages

## 4.1 Configuration

We run the simulations on two different network topologies. An example for both is depicted in Figure 6. Note that after the initialization of the simulation, the network as shown in Figure 6a will have a similar structure to the network shown in Figure 6b, with the difference that switches in the second layer have only 8 hosts attached to them instead of 15.



(a) Two-layered Network (b) Tree Network

Figure 6: Caption

**Parameters** We compare the simulations on a network with 64 hosts and an according number of switches. This results in a longest path of 4 hops for both networks. To find suiting parameters we calculate an estimation of each parameter and adjust them such that the times with the `LogGP` model match the times obtained with `OmnetNetwork` as close as possible in a congestion free network. The problem herein lies in the fact that for message sizes shorter than the maximal Ethernet frame length, the bandwidth gets approximately divided by the number of hops this message has to travel. In contrast to this, the bandwidth for very large messages remains roughly the same for different paths. Since an overestimation of the bandwidth would lead to a linear increase of the difference between the transmission times in `LogGP` and `OmnetNetwork`, we choose the bandwidth to model large messages accurately. $G$ describes the cost per byte of message. Therefore $G$ is equal to $1/bandwidth$. This choice for $G$ leads to an inaccurate modeling of short messages. We can compensate this partly by increasing the latency parameter $L$ of the network. Since the latency is constant per message, an increase affects smaller messages more, relatively to their size. We therefore test with both the actual maximal latency $L_1$ and an increased latency $L_2$. We find the actual maximal latency $L_1$ by calculating the time needed for the first bit of the shortest message possible to reach its destination. The second latency $L_2$ is chosen such that a message half the maximal Ethernet frame length is accurately modeled.

The parameter $o$ is calculated into every send and receive of `OmnetSimulation`. Therefore this parameter has to be the same in both simulations but can be chosen freely. We set the parameter $g$ to the standard defined by the INET MAC module, the MAC module pauses after a transmission of a frame for the time $96bit/bandwidth$.

$$\begin{array}{lcr} L_1 & = & 1732 \\ L_2 & = & 20440 \\ o & = & 1500 \\ g & = & 960 \\ G & = & 8 \end{array}$$

Table 5: Parameters for `LogGP`

Further parameters for our implementations are the size of the queue in every MAC module and the packet error rate of channels (PER $\in [0, 1]$). They are fixed at 1000 frames per queue and 0 respectively for these evaluations. The effects of these parameters can be explored in future work. Furthermore, all the simulations were run on a machine with a Intel Core i7-3520M with 2.90GHz and two physical cores, and 7.5GB RAM.

## 4.2 Simulation Measurements

| | | |
|---|---|---|
| Maximal host finish time [s] | = | $t_h$ |
| Simulation time [s] | = | $t_s$ |
| Simulation speed [ev/s] | = | $v_s$ |
| Average query per message | = | Q |

The following table shows the measured results of our simulations. The first column shows witch benchmark out of the NAS Parallel Benchmarks packet was used. All NAS Parallel Benchmarks were compiled with the size class $C$ and after that `liballprof-1.0` was used to collect the application traces. In the second column we see whether the `OmnetSimulation` or the `LogGP` model was used. For simulations in `OmnetSimulation` we state the used network topology by $N_1$ or $N_2$, for the two-layered network (Figure 6a) and the tree network topology (Figure 6b) respectively. For simulations in `LogGP`, we denote whether $L_1$ or $L_2$ was used.

| Benchmark | Simulation | $t_h$ [s] | $t_s$ [s] | $v_s$ [ev/s] | Q |
|---|---|---|---|---|---|
| BT | Omnet $N_1$ | 1432.926 | 7914 | 667.64 | 2.60 |
| | Omnet $N_2$ | 1432.994 | 6550 | 806.67 | 2.57 |
| | LogGP $L_1$ | 1433.185 | 24 | 220153.80 | - |
| | LogGP $L_2$ | 1433.238 | 24 | 220153.80 | - |
| EP | Omnet $N_1$ | 1063.496 | 1 | 6781.00 | 1.13 |
| | Omnet $N_2$ | 1063.496 | 1 | 6781.00 | 1.90 |
| | LogGP $L_1$ | 1063.496 | 0 | inf | - |
| | LogGP $L_2$ | 1063.497 | 0 | inf | - |
| LU | Omnet $N_1$ | 6429.736 | 4414 | 10284.70 | 1.17 |
| | Omnet $N_2$ | 6428.939 | 3920 | 11580.88 | 1.11 |
| | LogGP $L_1$ | 6425.537 | 171 | 265479.81 | - |
| | LogGP $L_2$ | 6428.592 | 224 | 186053.47 | - |
| MG | Omnet $N_1$ | 982.942 | 664 | 2305.36 | 1.89 |
| | Omnet $N_2$ | 983.946 | 654 | 2340.61 | 2.02 |
| | LogGP $L_1$ | 981.759 | 5 | 306152.00 | - |
| | LogGP $L_2$ | 981.814 | 6 | 218680.00 | - |
| SP | Omnet $N_1$ | 1211.946 | 13916 | 601.85 | 3.65 |
| | Omnet $N_2$ | 1213.26 | 11618 | 720.89 | 3.26 |
| | LogGP $L_1$ | 1182.028 | 28 | 299117.19 | - |
| | LogGP $L_2$ | 1182.343 | 26 | 322126.19 | - |

Table 6: Comparison on a 64 node network

From the results in Table 6 we can see that first and foremost, the times obtained by `LogGP` and `OmnetSimulation` are very similar with one exception. Other than the SP application trace, all the in-simulation times differs less than 0.2%. For the SP application trace, the time differs roughly 2.5%. Furthermore, the running time of the simulator dramatically increased, in the case of SP by a factor of over 500.

When inspecting the results a bit closer, we see that the two different networks do not have a big influence on the maximal finish time and the simulation time of `OmnetSimulation`. This is most likely due to the fact that the two networks $N_1$ and $N_2$ have a similar structure after the initialization and therefore similar traveling times for the packets. The Table 7 shows us what the average message size in each application trace is.

In the benchmark SP we have a difference of more than 2.5% in the simulated time. This is most likely due to heavy congestion in the network, that delays the completion of send operations. The other benchmark with about the same average mesage size is BT and this benchmark got similar simulated times as `LogGP`, i.e., within 0.02% from each other. This not uncorrelated is the fact that SP has also the worst running time compered to `LogGP`.

| Benchmark | Avg. Message size |
|---|---|
| BT | 82554 |
| EP | 20 |
| LU | 2442 |
| MG | 19072 |
| SP | 73442 |

Table 7: Average Message Size for NSA Benchmarks

## 4.3 Performance

We can see in Table 6 that `OmnetSimulation` increases the running time of LogGOPSim roughly by a factor between 15 and 550. With the current implementation simulating more than 64 calculating nodes for a application as large as some of the NAS benchmarks seems not possible on a single processor in reasonable time. We can see in Table 6, that the most time-consuming simulation took about 4 hours to complete. We can also see that generally LogGOPSim calls the query function under 4 times per message, in the best case only 1.11 times. In other words, we cannot gain a big speed up by improving the predictions of arrival times. We can safely say that the bottleneck is definitely the Omnet++ network, but to determine how to improve running times one would have to investigate more.

# 5 Related Work

LogGOPSim [4] is the large-scale network simulator we use as a basis for this work. Using the tools provided with the distribution of LogGOPSim, one can generate inputs for LogGOPSim out of an MPI traces. This allows to simulate real world application traces on the LogGOPSim simulator. LogGOPSim was previously used to show the effect of system noise on large-scale applications in a paper [3].

Omnet++ [8] is a simulation library and framework that allows to build simulations with relative ease. It was originally developed Andrs Varga and is now maintained by OpenSim Ltd.. It allows the user to build a wide variety of network simulations.

Inet [5] is maintained by a small part of the team behind Omnet++, but heavily leans on contributions of the community. Inet provides modules for many protocols, applications, and other components. These protocols can be used in combination with Omnet++ to build sophisticated simulations.

The objective of the master thesis "Packet Tracing in Simulation Environments" [9] of Vladimir Zorin is to simulate real world program traces on an Infiband network in Omnet++. To simulate the real world traces, his work uses LogGOPSim. To connect LogGOPSim and Omnet++ he takes a different approach. In his implementation both LogGOPSim and Omnet++ run as separate processes and communicate via inter process communication (IPC). He states in his thesis that his simulator only uses about 10% more time than a simulation solely in Omnet++. Furthermore he states that the simulation time grows proportional to the increase in number of nodes in the network and to the increase of the simulated time.

# 6 Conclusion

We built a network simulator by combining LogGOPSim [4] and Omnet++ [8]. The goal was to let LogGOPSim simulate the application of each node in a HPC cluster but delegate the simulation of the network to Omnet++. We achieved this by embedding the Omnet++ simulation kernel into LogGOPSim. Therefore we extended the existing implementation of a network structure in LogGOPSim such that no change to the LogGOPSim simulation kernel was necessary. The simulation builds Ethernet networks out of the network topologies defined by the input files to LogGOPSim. Furthermore, we extended some Ethernet modules from Inet [5] in such a way that allows us to track packets sent thought the network and estimate their arrival at their destination.

The previously existing network models implemented in LogGOPSim either assume a fully connected network or model message transmission in the following way: Instead of simulating each hop individually, a message transmission occupies the whole path at once and shares the bandwidth with all the packets using any part of this path. In contrast to this, the combination of LogGOPSim and Omnet++ is capable of simulating every hop of a Ethernet network individually. Furthermore, this simulation can model packet loss due to too much congestion or packet corruption. We implemented a direct resend mechanism in oracle like fashion to recover from such packet losses.

We showed with the NAS Parallel Benchmarks [1] that this implementation yields results that are comparable to the existing `LogGP` model. Only in one benchmark did we see a significant increase in the simulated time due to congestion in the network.

This implementation can be further improved by implementing more communication standards like Ethernet, such as Infiband.

# References

[1] Bailey, David H., et al. The NAS parallel benchmarks. In *nternational Journal of High Performance*, pages 63–73. Computing Applications 5.3, 1991.

[2] N. F. F. Ino and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Symposium on Principles and Practices of Parallel Progr.*, pages 133–142. ACM, 06 2001.

[3] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 11 2010.

[4] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 597–604. ACM, 06 2010.

[5] Inet Framework. `https://inet.omnetpp.org/`, 2016. [Online; accessed 29-August-2016].

[6] Joseph B. Kruskal, Jr. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In *Proceedings of the American Mathematical Society*, pages 48–50. American Mathematical Society, 02 1956.

[7] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html`, 2016. [Online; accessed 8-September-2016].

[8] OpenSim Ltd. Omnet++ Discrete Event Simulator. `https://omnetpp.org/`, 2016. [Online; accessed 29-August-2016].

[9] V. Zorin. Packet Tracing in Simulation Environments. Master's thesis, University of Oslo, 2011. [Online; accessed 7-September-2016].

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Congestion-aware Simulation of Large-scale HPC Networks

**Authored by** (in block letters):
*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Maag | Daniel |
| | |
| | |
| | |
| | |

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| Zürich, 16.09.2016 | [illegible] |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*