Master's thesis at the Software Engineering Research Group of the Institute of Computer Science

Modelling Communication behaviour of Parallel Programs

Überwachen und Strafen

Fabian Kovacs

fabian.kovacs@fu-berlin.de

Student Id: 5056093

First Reviewer: Barry Linnert

Second Reviewer: Prof. Dr.-Ing. Jochen Schiller

Supervisor: Barry Linnert

Berlin, 11.09.2018

Zusammenfassung

Die steigende Nutzung von high performance computing im industriellen Bereich verlangt eine andere Art von Scheduling als die üblichen FIFO/Queue-basierten Strategien, weil diese nicht in der Lage sind, Antwortzeiten effektiv vorherzusagen. Um Antwortzeit-Garantien geben zu können, bedarf es detaillierter Informationen über das Laufzeitverhalten und die Ressourcennutzung der Anwendungen. Zentral zur parallelen High Performance Anwendung ist die Interprozess Kommunikation, zur Koordinierung und den Austausch von Daten. Kommunikationszeit ist ein nicht-trivialer Anteil der Laufzeit solcher Anwendungen, wird in state-of-the-art Schedulern aber noch nicht effektiv berücksichtigt. In dieser Arbeit wird ein Linux Kernel Modul vorgestellt, welches die Kommunikation von parallelen Programmen überwacht um in späteren Abläufen des selben Programms diese Schedulern bereitzustellen. Zusätzlich wird die Kommunikation dieser Anwendungen mit vorherigen Abläufen verglichen um effektives Scheduling zu garantieren. Der Effekt dieses Monitorings auf Performance und Durchsatz einer typischen high performance computing Anwendung wird gemessen und überprüft.

Abstract

The increasing application of high performance computing in industrial settings demands more precise scheduling algorithms than the commonly applied FIFO-queues. Towards this end, schedulers need more detailed knowledge of the runtime behaviour and resource usage of applications. One of the central features of high performance computing is interprocess communication for coordination and data-exchange. The time spent communicating is non-trivial but not yet well integrated into state of the art schedulers. This thesis will introduce a linux kernel module for monitoring the communication of parallel programs. The module will be used to extract the message lengths for the scheduler and then be used in consecutive runs of the application to verify the repetition of that application. The impact on performance and throughput of high-performance applications through monitoring will be reported.

Selbstständigkeitserklärung

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht.

Table of Contents

Introduction

The field of high performance computing (HPC) has seen continuous growth ever since its inception. In high performance computing, data is so large and computation so complex that data-centres the size of industrial warehouses are required to satisfy demands: tens of thousands of high-end computers are arranged and connected to serve weather scientists, material designers, physicists, be it nuclear or cosmological or others.

While a common household computer might have between two and sixteen processor cores, a data-centre (or cluster) is composed of more than a thousand of those computers, highly optimized for fast computation. Even at those magnitudes, computations can see execution times upwards of multiple weeks.

High performance computing is a multi-disciplinary domain of applied software engineering, algorithm- and hardware design. To make full use of the available hardware a programmer has to be proficient in all three of them, and in addition the scientific domain. They have to be capable of writing correct and maintainable software; adapt their algorithms to the parallelism and latencies of an application distributed among thousands of compute nodes; make use of available hardware extensions and capabilities, and know its limitations (e.g.: caching behaviour, vectorization support or GPUs); and they need to be able to understand and verify their problems and solutions at hand.

Correctness of computation is obviously a primary concern, however at those scales, enabling and improving high performance is of the same importance: Where an improvement in performance might make a

commodity application feel more responsive or reduce the waiting time of the user by a few minutes (e.g.: Flight searching websites like SkyScanner[1]), a performance improvement of just 1% will reduce the execution time of a 3 month long-running application by close to 1 day.

High performance computing has been, up until the end of the last century, restricted to the public sector as documented on the Top500[2]: until 1997 only 32.6% of supercomputers belonged to the private sector. The increase in computing power has enabled the private sector to create their own high performance data-centres, which have been steadily gaining shares; sitting at 58.4% as of 01.06.2019. [1, 2]

Executing an application on a high performance cluster differs significantly from everyday applications. Instead of starting and immediately executing the application, a user will submit their

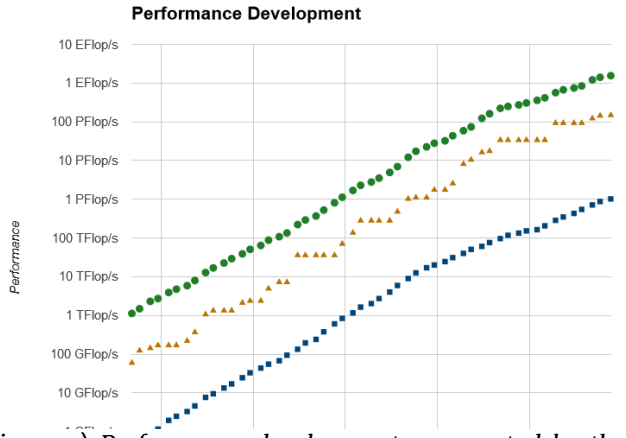

Figure 1) Performance development as reported by the Top500.

(Taken from: https://www.top500.org/statistics/perfdevel/)

[1] https://www.skyscanner.net

[2] The de facto list of high performance computing data-centres is maintained at Top500.org, an international effort to gather standardized performance information of "the 500 most powerful commercially available computer systems known to us." In addition to raw computation power, the Top500 documents key statistics of those supercomputers, such as their operating system, specific hardware and network layout, as well as the sectors they are used in (public/private, etc.).

application (called job) into a job queue, including a reservation for a number of compute nodes, and often also an estimation of running time. Those queues are usually executed according to a first-come-first-served (FCFCS) fashion, however, modulated with some fairness constraints. [3]

As the execution time of these applications can range from hours to weeks, submitting a task is not a guarantee for timely execution. Where unpredictability was an inconvenience to the academic user, it now falls under the profit considerations of industrial corporations.

Yu and Buyya define five dimensions for the quality of service of a cluster management system: Time, Cost, Fidelity, Reliability and Security. [4]

> *Time* — Users want to minimize their applications response time[3], and want to know when their computation is finished.
>
> *Cost* — Users want to minimize their expenses.
>
> *Fidelity* — Users want to receive correct results.
>
> *Reliability* — Users need guarantees about the execution and completion of their application.
>
> *Security* — Users need guarantees about the confidentiality of their datasets and applications' code.

The task of a cluster management system is to uphold these quality of service metrics. Towards those goals, it has access to a variety of scheduling strategies, all serving the goal of finding the proper

[3] Wait time and computation time.

placement of an application in time and space: when an application is executed, and on how many and which compute nodes.

Scheduling high performance applications

As stated above, scheduling is very often done in a very rudimentary manner: first-come-first-served, with an eye for fairness, to avoid situations where a single user monopolizes the queue and other users have to wait longer. [5] This can often lead to situations where the data-centre is not fully utilized by the head of the queue alone. A common technique to increase utilization is therefore backfilling, where tasks further back in the queue are scheduled, as long as their running time does not exceed that of the current longest-running task. [6] In this type of scheduling, a job is usually allocated a fixed partition of the data-centre without overlap with other jobs.

This, however, is still not optimal, as not every application utilizes all components of a compute node at the same time nor to their full capacity: for example, newer generation data-centres have FPGAs[4] or GPUs which might simply not yet be used by a deployed application[5]. Alternatively, an application might be in a phase, where it will move a lot of data to/from disk and therefore not do much computation at all. An advanced scheduling algorithm could therefore detect those situations and co-locate some jobs under the condition that their resource usage is exclusive, which would further increase utilization and reduce response time.

---

[4] Field-programmable gate array

[5] As an example, the protein clustering suite MMseqs2 is described by its authors as embarrassingly parallel, making little to no use of network communication hardware beyond setup and teardown.

While theoretically sound, this is not a commonly used technique, mostly due to the fact that data-centre operators and therefore schedulers do not have access to the necessary runtime information to effectively make those decisions. The runtime information relevant to a scheduler are:

1. Total running time
2. Number of Nodes used
3. Per Node Utilization of[6]:
    a. CPU
    b. RAM
    c. Network Communication
    d. Disk I/O
    e. Hardware extensions (such as GPUs or FPGAs)
    f. Idle times
4. Optionally: a user specified deadline

Not only is this information required for more intricate scheduling schemes, it is also necessary to create runtime estimations of an application: while the obvious upper bound on any computation's capability is the amount of data relative to the power of the system, the usage of peripheral devices such as networking hardware or hard disk drives also adds to the running time of an application. For example, the round-trip time of a network message to the publicly accessible Network Time Protocol server *o.de.pool.ntp.org* takes on average 22ms, however, it mostly consists of sending and receiving a network message. Therefore, CPU utilization is low in the meantime, but the time spent in that routine is static, if a scheduler fails to consider this,

[6] Notably, we are not only interested in this information as an aggregate, but over time.

it will not be able to effectively estimate runtime of applications. As will be outlined in more detail below, a central component of parallel programs is local network communication, a major contributor to the running time of parallel applications.

Plan based Scheduler — Scheduling for Quality of Service

There exist two major paradigms for scheduling of high performance applications: queue based- and plan-based schedulers. A queue based scheduler will execute programs in order of submission; when a program is finished the next program in the queue is executed. As mentioned above, modern queue based schedulers will still use some mechanisms to ensure fairness, and in addition can use backfilling to increase utilisation of the data centre. A plan based scheduler uses metadata and requirements submitted by the users to create an execution plan for all applications: when which application will be running on what nodes. Such an execution plan will usually be re-generated after a certain time has passed. A plan based scheduler can use auxiliary information to create a schedule. For example, it can use prioritization and fairness schemes to favor higher paying users, while not completely starving out other users. It can also use models and estimates of the resource usage of an application to improve scheduling.

As a plan based scheduler is predictable, it has better access to scheduling information and can therefore provide guarantees on the execution times of processes. As mentioned earlier, users want predictability in the execution times of their applications: especially in commercial settings users will work with deadlines that must be respected and guaranteed. If we also consider that an application might

be running in a distributed workflow[7] (i.e.: on multiple, connected data-centres), it becomes crucial to guarantee deadlines, to enable communication and synchronization. A plan based scheduler is therefore preferred in those settings.

Grid computing

Grid computing is a software engineering effort towards commoditization and consolidation of high performance computing. The word derives its meaning twofold: 1) The end goal of grid computing is computation, especially high performance computing, as ubiquitous as the power grid. 2) Towards that goal, instead of only building bigger and better data-centres, compute sites are to be transparently interconnected on a grid. The heterogeneous nature of these compute sites should also be abstracted away from the user, such that they only need to submit their application. Commoditization has also introduced Service Level Agreements (SLAs) into high performance computing, meaning a contractual agreement between both parties on the Quality of Service (QoS).

Another application of grid computing is the analysis of large data-sets such as that of the Large Hadron Collider at CERN. It is not uncommon for data-sets of this magnitude to be sent via mail-delivery, as online transmission is comparatively slower than the delivery of multiple hard-disk drives. In comparison, the size of a typical application can range from 10MB to up to 1GB, but is still far below 1% of the size of the dataset's size. An emerging approach is therefore to no longer move

[7] A use case that is only becoming more common with bigger datasets and more available compute resources.

the dataset, but the applications onto the hosting data-centre, therefore eliminating the need to move the large datasets.

Plan based scheduling is a critical component of grid-computing, to enable the successful negotiation, scheduling and execution of dependent parts of an application on multiple distributed compute sites with differing hardware and capabilities. The already complex task of runtime estimations is further complicated by the heterogeneous environment of a grid. A plan based scheduler for grid based environments can therefore improve its estimations by historic data and models of the execution of the applications it is executing. [7]

Parallel Programs

High performance computing requires much more compute power than can be accomplished by a single processor, or even a single multi core processor. A data-centre therefore necessarily consists of a large number of well-connected compute nodes. Making full use of a HPC data-centre therefore also requires a strong emphasis on parallel software design and programming. An application and its underlying algorithms are broken down, such that they can be executed in a distributed fashion on multiple compute nodes at once. It follows, that a core pillar of parallel program design is the ability to exchange information between compute nodes. A successful parallelisation can produce significant speedups simply due to the ability to scale better with more available resources.

An example — findMax

As an example, consider the implementation of finding the maximum in a list of numbers. The sequential solution to that problem is to scan

all values, carrying the maximum number in a variable, updating it when a higher value is encountered. A processor able to process 100 numbers per second would therefore process 1000 such numbers in 10s.

A naive parallel solution to the same problem might be: first distribute those numbers evenly among all processors, then execute the sequential algorithm per node, gather the intermediate maxima on a single node and find the absolute maximum among them. If we execute this parallel implementation on 4 cores of the same performance, it now takes every core just 2.5s to calculate their local maximum, the calculation of the final global maximum incurs an additional 0.04s. If we assume message transmission to take 0.1s per node[8], we are still left with a total runtime of $2.5s + 0.04s + 2 \cdot 0.3s = 4.04s$, which is an improvement of 59.6% compared to the sequential implementation.

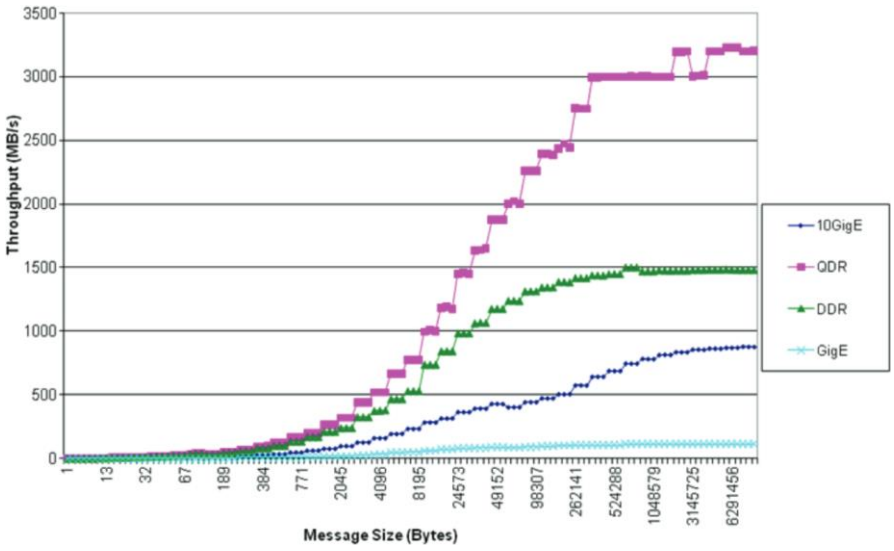

Figure 2) Throughput measurements of different connection technologies. Taken from [27]

[8] Excluding the one node that distributes the dataset and gathers the intermediate results.

SEQUENTIAL IMPLEMENTATION

```
findMax(N[]):
      max = -inf


      for n in N do:
            if n > max do:
                  max = n

      return max
```

PARALLEL IMPLEMENTATION

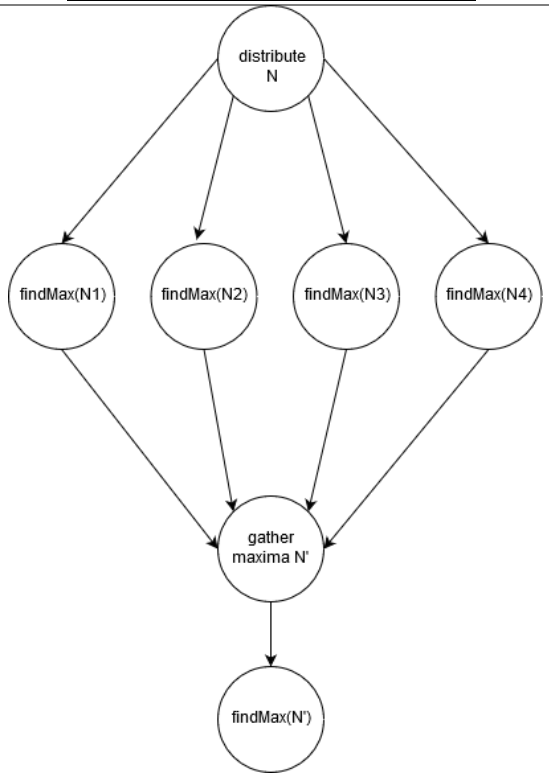

Figure 3) Pseudocode and visualization of sequential and parallel implementation of findMax algorithm. Note that the parallel implementation has four processors at once and therefore a smaller per-CPU dataset. However, the parallel implementation now has to coordinate and communicate, which is time spent on overhead instead of computation.

Network communication is a non-trivial component of an application's running time. While transmission of a single message might take only a few milliseconds it can take far longer for longer messages. The major influencing factors of message transmission time are the size of the message, the software implementation of the networking code, the transmission speed of the networking hardware, the medium and protocol used to transmit the message, the physical distance and connectivity[9] between the communicating parties.

Increasing parallelisation is therefore not a panacea to any long running or complex computation: increasing parallelisation also

[9] The specific layout of the network and current load on its components.

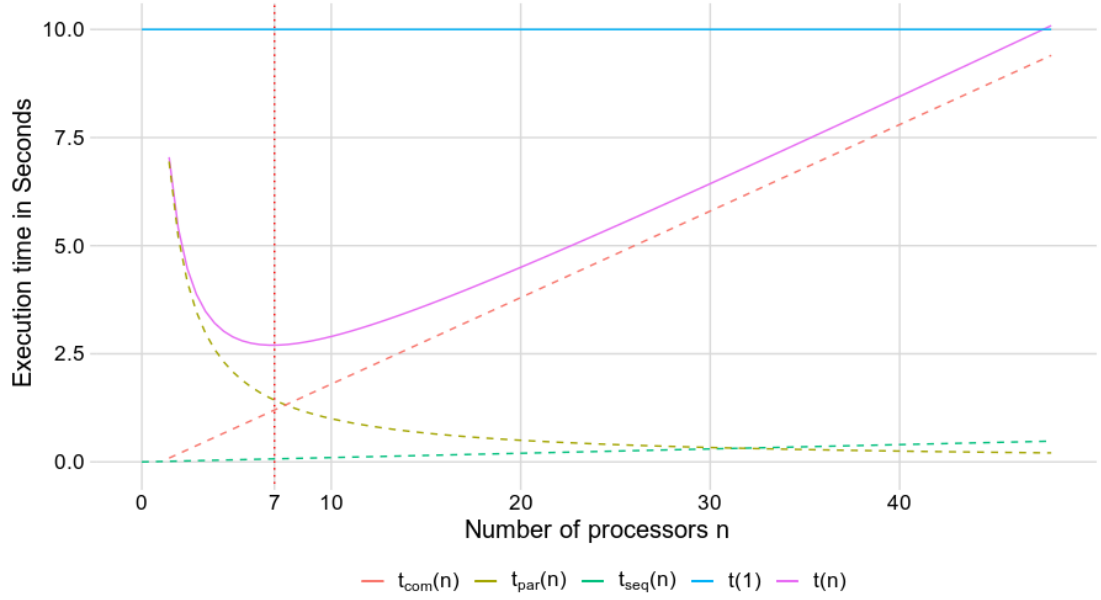

Figure 4) Visualization of execution time of sequential and parallel implementations, with decomposition of time components of parallel implementation.

increases time spent communicating. We define the speedup from parallelisation as $S(n) = \frac{t(n)}{t(1)}$, where $t(n)$ is the time it takes to compute a task with n nodes. The computation time of an application can be broken down into three components $t(n) = t_{par}(n) + t_{seq}(n) + t_{com}(n)$, where $t_{par}(n)$ is the time spent in parallel computation, $t_{seq}(n)$ the non-parallelizable part of a program, and $t_{com}(n)$ the time spent communicating.

From our example above $t(n) = \frac{1000}{n \cdot 100} + \frac{n}{100} + (n-1) \cdot 0.1, n > 1, t(1) = 10$. The speedup at 4 cores is therefore $S(p) = 0.31$. The minimum execution time with 7 cores would then be at 2.69s, this is due to the fact, that while the work done per node diminishes with each new node ($t_{par}(n) = \frac{1000}{n\ 100}$), the time spent finding the absolute maximum from the intermediate results increases ($t_{seq}(n) = \frac{n}{100}$), and more importantly, with every additional node, the execution time increases ($t_{com}(n) = (n-1) \cdot 0.1$).

A parallelisation is only useful as long as $t(1) > t(n)$ holds, i.e. as long as the sequential implementation is slower than the parallelised implementation using n cores. Which, in our example, holds up to roughly 47 nodes. If we also consider the cost of running or renting the nodes, the maximum sensible n is probably significantly lower than that.

It is therefore an important part of developing, executing and scheduling an application to consider an application's scaling, and thus finding an optimal selection of nodes in the context of the executing data-centre, which is dependent on the problem size and the amount of necessary communication. This optimization also includes the user's willingness to pay for a partition.

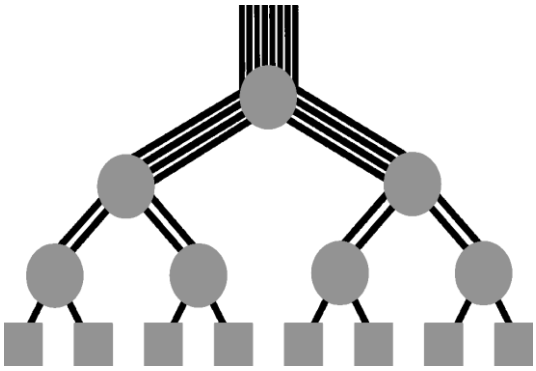

Figure 5) Fat Tree Topology: nodes are interconnected in a hierarchical structure, nodes higher up in the hierarchy have more connections. This enables rapid transmission through the network with low congestion, while having modest requirements on connection infrastructure. Tree structures are very easy to partition.
(Used by: Summit #1, #2 LC Sierra) (Image taken from [9]*)*

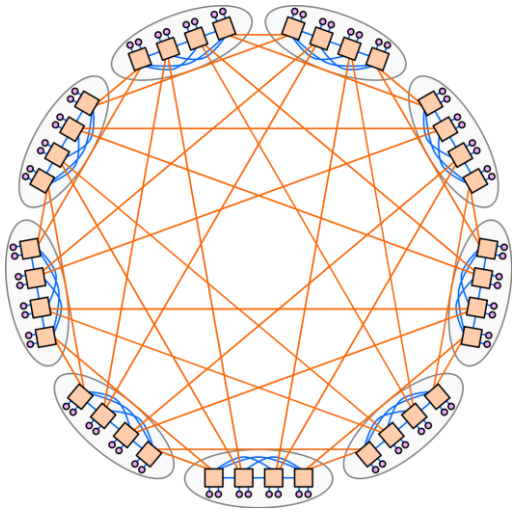

Figure 6) Dragonfly Architecture: fully connected groups of nodes, where every group has at least one link to another group. This enables short communication times and is easy to partition.
(Used by: Sunway TaihuLight #3, Piz Daint #5)
(Image taken from [10]*)*

Topology

The size of a high performance computing data centre puts pressure onto its designers in the placement and connectivity of its nodes: the network topology of a cluster. Multiple factors play into the consideration of topology design: connectivity, latency (min, max and average), throughput, ability to be partitioned, and cost. While a fully connected cluster has high connectivity and thus low latencies, while simultaneously being simple to partition, connecting every node with every node becomes very expensive, as the cost scales quadratically with every node. The typical primitives in topology design are trees, n-ary-cubes or -meshes, or combinations like dragonfly. [8] (See Figure 5 and 6)

The topology of a cluster has significant impact on the time it takes for a message to be transmitted and also impacts the ability of an applications communication

capabilities: for example, two nodes involved in a lot of communication should not be deep in two separate branches of a tree structure.

A QoS oriented scheduler aiming to provide effective deadline guarantees, therefore not only requires knowledge of the particular topology of an application, but also of each allocated node's communication.

Message Passing Interface — MPI

One of the most commonly used frameworks for the development of parallel programs is the Message Passing Interface standard (MPI)[10]. Libraries implementing the MPI standard offer a programming interface for parallel communication, as well as a toolkit for compiling and running distributed applications. They implement primitives for synchronous and asynchronous communication, including one-to-one (*Send* / *Receive*) and one-to-many (*Scatter* / *Gather*) communication, but also more complex schemes such as *MPI_Allreduce*, where all involved parties apply an operation to a distributed buffer, combining their results. MPI abstracts over the protocols and execution platform: a user will simply write their program using MPI primitives and will be able to execute it on a single machine, or on a high performance computing cluster. MPI will load

[10] A growing trend in high performance computing is the emergence of higher level interfaces for processing, e.g.: Apache Spark. This can be read as a decline of MPI, however most of these tend to still use MPI under the hood for execution.

and use the appropriate drivers and distribute the application accordingly.

MPI applications are usually run from a management system, this requires configuration specific to the executing cluster, including the available hardware and physical layout of the cluster: it is beneficial for nodes communicating a lot with each other to also be placed physically close to each other, similarly nodes acting as a communication hub might want to have a low mean distance to all nodes to allow fast communication with all nodes. Most data centres host multiple programs on a cluster at the same time, through partitioning of the cluster. This might be because the user has only limited use for the whole cluster or has only paid for a certain partition of the cluster. MPI can manage these requirements, to simplify execution on data-centres for high performance parallel computation.

High performance applications

While the scientific domains of high performance computing are varied, most applications focus on simulating parts of the natural world with numerical models. The following section will present some specific applications of high performance computing, as examples of HPC in some domains. It should be noted that this list is not exhaustive, and some applications are solutions to very specific problems that are not publicly available.

While it was originally mostly used in academia, limited by the availability of resources, technologies and brainpower, it has been seeing more use in industrial applications since the turn of the 21st century.

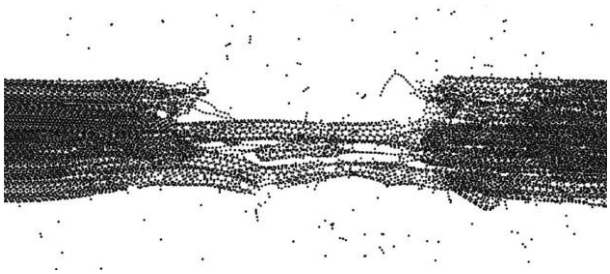

Figure 7) Model of the tensile response of bundles of carbon nanotubes containing 1.2M atoms, simulated with LAMMPS.

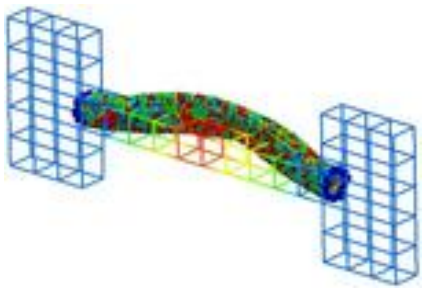

Figure 8) Electron/transport enhanced simulation of heating and deformation of a metallic carbon nanotube, simulated with LAMMPS.

The public sector has been the driving force behind high performance computing. The applications range from large scale universe simulations to microbial simulations, down to quantum scale. Lammps - Large-scale Atomic/Molecular Massively Parallel Simulator - developed at Sandia National Laboratories is a high performance application used for physical material modeling from the atomic level upwards[11]. One of the major applications of high performance computing is the simulation of nuclear

weapons: the US put live nuclear weapons testing on hold in 1992 and has since then started developing and testing nuclear bombs with their Advanced Simulation and

Computing Program in Los Alamos (#7 of Top500), Sandia Laboratories (#7 of Top500) and Lawrence Livermore National Laboratory (#10 of Top500).[12]

[11] More examples can be found at https://lammps.sandia.gov/pictures.html
[12] Stockpile Stewardship and Management Program

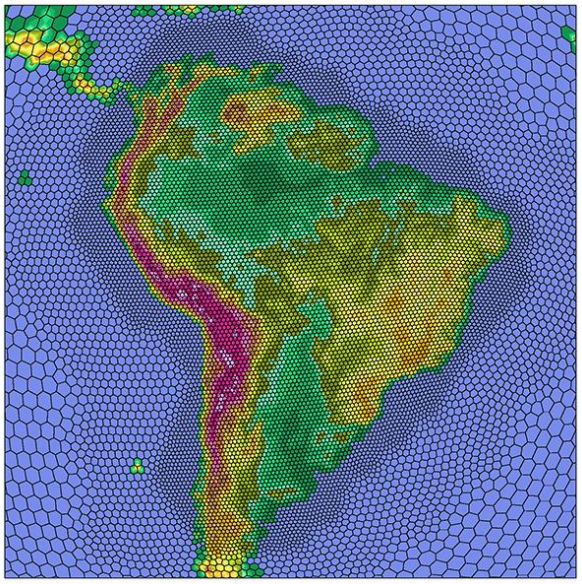

Figure 9) OLAM-Soil simulation with adaptive Grid.

The European Organization for Nuclear Research - CERN - requires high performance computing[13] to effectively handle its data: the team surrounding the Large Hadron Collider has so far released about 3 Petabyte of data, estimating that this is just 3 percent of their data. A dataset of this scope cannot effectively be handled at any other scale than a supercomputer, or in this case a super-computing grid.

Lastly, high performance computing is used in climate research, to create day to day weather-forecasts, but also to simulate and analyse climate change and the effect of human actions on climate. Toward ever greater precision, climate or planetary models incorporate more details into their computation: the Ocean-Land-Atmosphere and Soil Model (OLAM) simulates cloud dynamics and microphysics, atmosphere-surface interaction, and land surface processes, such as a regions groundwater, in addition they employ dynamic grid techniques

[13] CERN does not maintain their own datacentre, instead relying on the Worldwide LHC Computing Grid (WLCG), a network of more than 170 computing centres spread across 42 countries

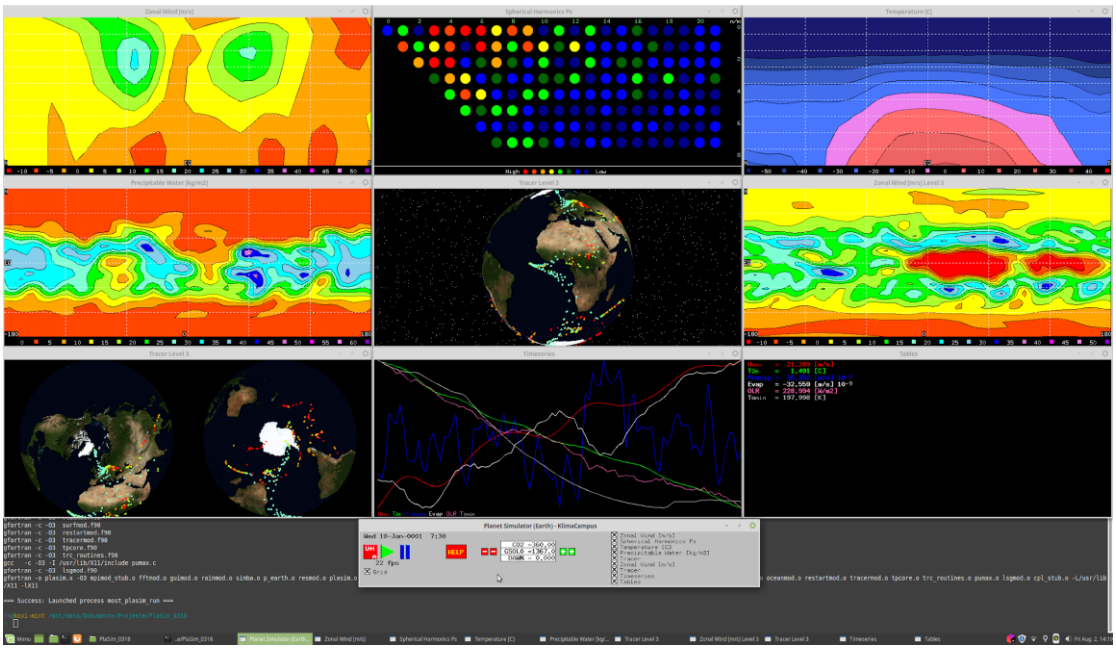

Figure 10) Screenshot of a running PlaSim simulation.

to adaptively reduce computational load on uninteresting data without completely ignoring it. [11, 12]

PlaSim (for Planet Simulator) is a general circulation simulation often used in weather forecasting and planetary simulation. PlaSim is developed by Haberkorn et al. at Meteorologisches Institut, Universität Hamburg. It is specifically developed to be user friendly (simple to set up, run and evaluate), for teaching purposes, while making slight concessions with regard to precision and detail, but still being representative of other forecasting programs such as Olam-Soil. [13] It comes with an extensive UI to configure and run PlaSim, which can also display a detailed live model of the running simulation. Plasim is written in Fortran90 and makes use of MPI for parallel communication.

RESEARCH TITLE	ORGANIZATION
Computational Studies of Protein-Protein Interactions	University of Chicago
Towards development of the structural determinants of the Glutamate receptor gating regulation by auxiliary membrane anchored proteins	Carnegie-Mellon
Curvilinear and Multipatch Methods for General Relativistic Astrophysics in the Gravitational Wave Era	Rochester Institute of Tech
Centre for the Physics of Living Cells	University of Illinois at Urbana-Champaign
Looking Out for the Little Guy: A Comprehensive Study of Star Formation in Dwarf Galaxies	Rutgers University New Brunswick
Relaminarization and Turbulence Suppression in Rotating Flows	University of Kentucky Research Foundation
Petascale Simulations of Binary Neutron Star Mergers	University of California-Berkeley
Quasars and Large Scale Structure:Gigaparsec-scale simulations confront Large Survey Data	Carnegie-Mellon University
A Hierarchical Multiscale Method for Nonlocal Fine-scale Models via Merging Weak Galerkin and VMS Frameworks	University of Illinois at Urbana-Champaign
The First Billion Years: a Petascale Universe of Galaxies and Quasars	Carnegie-Mellon University

Sample list of National Science Foundation awarded research for Petascale Computing Resource Allocations.
(Sources: https://www.nsf.gov/pubs/2014/nsf14518/nsf14518.htm, and https://www.nsf.gov/awardsearch/simpleSearchResult?queryText=PRAC)

PlaSim will be used in the case study further below, the reason being that it is fairly straightforward to set up while actually requiring HPC and having complex communication behavior. Both Olam-Soil and LAMMPS were also considered as case study candidates, but were impossible to setup without investing a lot of time.

Initial testing was done with a rudimentary implementation of an N-body simulation, where multiple physical bodies and their gravity induced movement is simulated. It is commonly used as an introductory application for parallel programming as it is easy to implement while requiring effective parallelism and communication. N-bodies is primarily used in astrophysical simulations of the universe. While the used implementation was easy to set up, it did not make use of many communication primitives and did not display a sufficiently complex behaviour to be considered representative of our target domain.

The United States National Science Foundation issues grants for scientific research considered important and relevant. One of their awards is for Petascale Computing Resource Allocations (PRAC). The awards are selected by committee and can therefore be considered representative of high performance computing's application domain.

High performance computing is also used in industrial settings for material simulation, optimization and stress testing. For example, before any car is put into an actual wind tunnel, it will already have undergone optimization and verification in simulated wind tunnels. This enables much faster turnaround. In architectural design HPC aides in the computational analysis of building structures, to test and verify new materials, shapes and construction technologies. [14] Modern plane design makes heavy use of high performance computing to optimize the shape of wings and turbines. A manufacturer can employ optimization algorithms coupled with physical simulation to achieve new designs:

"Gas turbine engine design begins with lower-fidelity models to explore the design space efficiently. [...] These models are used for evaluating engine subsystems [...] As the design process matures, higher-fidelity models are introduced. In high-fidelity analysis, the equations that represent the underlying physics are used to study the performance characteristics of different designs and shapes. Decomposing the parts, or the air around the parts, into a computational grid allows us to solve these equations. Capturing more geometry features, increasing the number of points in the grid, and extending the computational domain can achieve even higher-fidelity and more accurate analysis, but this increases the amount of computational effort considerably. As the design matures, more details are added to the models. These detailed models are larger and take longer to run, causing delays in the design cycle. Often the designers must balance the amount of detail in the models with the time

required to complete these calculations. One of the benefits of using HPC is that the run times of detailed analysis can be reduced from

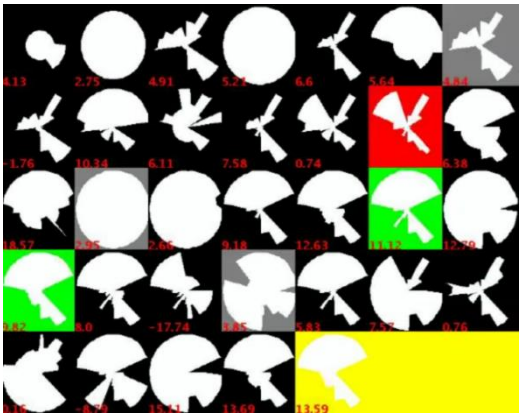

Figure 11) Genetic algorithm optimizing the shape of a wind turbine. (Screenshot taken from: https://www.youtube.com/watch?v=YZUNRmwoijw)

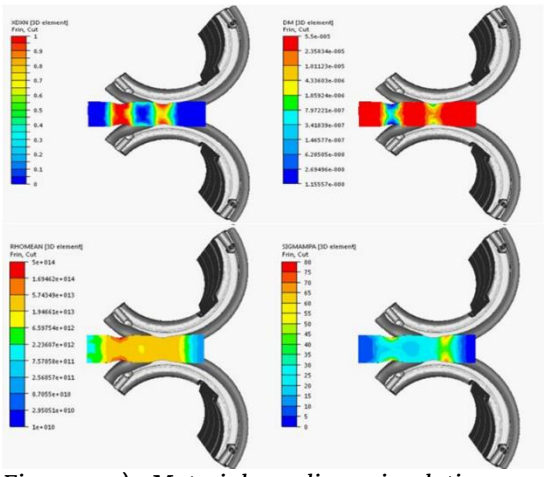

Figure 12) Material cooling simulation and optimisation. (From: https://www.semanticscholar.org/paper/HPC-SIMULATION-AND-OPTIMIZATION-OF-MATERIAL-FORMING-Fran%C3%A7ois-Jaouen/890479cd6aa3130d0f1790f152bf85da31fc7f7e)

> weeks to days and, with enough computing power, from days to hours." [2]

Towards modelling of communication

It should be clear by now, that a scheduler aiming to guarantee QoS and especially work within deadlines, needs as much information on the execution time and runtime behavior of an application. The runtime behaviour however is not apparent from the application's source code and therefore needs to be modelled in some way. The most basic way of modelling an application is by recording its behaviour in a controlled setting and using that as a baseline for future executions, reifying it as more recordings come along. Applied to the communication of an application, this would entail recording the communication channel and the length of the messages. As the focus of this thesis is high performance parallel programs, we will restrict ourselves to the usage of MPI, the resultant model will therefore consist solely of the length of every sent message[14] in order, without any identification of the communicating parties[15]. This data can then be used by a plan based scheduler in later executions, to estimate peak loads on communication channels and CPU bursts, and additionally to help estimate the time spent communicating. These estimations serve in scheduling these applications and create precise estimations of their execution time, in order to guarantee deadlines. As the scheduler relies on the model's correctness to guarantee deadlines, the communication of the program will also be monitored in production and then compared to a supplied model, deviations need to be detected and

[14] In bytes

[15] Notably, this stabilizes against MPI_Broadcast etc. (re-) ordering messages.

communicated to the plan based scheduler. The scheduler can use this information to adapt its model, alter its schedule, or outright terminate the program if the degree of deviation is too high.

While it might sound obvious that a program does repeat itself under the same or very similar conditions, this is still an assumption that has to be verified: while most HPC programs rely on generalized numerical formula for their computation, their underlying models might change subtly with minor differences in the starting conditions. For example, OLAM-Soil outlined above uses an adaptive grid to partition the geography and thus its working set onto compute nodes (i.e.: areas of more importance receive more workers), this partitioning can be dynamic and therefore the communication between neighbouring nodes can change over the runtime of an application. A digital butterfly-effect might just produce a major difference in the shape of the computation from just a minor difference in initial conditions. It is therefore important to verify if applications developed and optimized for high performance computation can effectively be monitored by modelling them based on their past behaviour and characterize if and how a program's execution might differ from another[16].

It is also important to consider that introducing monitoring onto a platform will in some way alter the performance of the monitored application and/or that of the whole system: firstly, monitoring will introduce a new application onto the system that is not dedicated to productive work and there will therefore be less resources available for computation; Secondly, monitoring will in some way have to intercept the communication of the monitored program, meaning that it will

[16] The completely dynamic case outlined is hopefully not common, and at this moment considered out of scope.

alter or interfere with the code handling communication, most likely slowing it down to some degree. The amount of communication done in HPC varies, but communication is still central; it follows, that slowdown in communication might produce significant performance losses. It is therefore important and necessary to characterize the effect of monitoring on the performance of these applications. In addition, as the aim of plan based schedulers is to provide quasi-real-time guarantees, it is also important to minimize that effect, but additionally have it be stable and well characterized.

Purpose of this thesis

To estimate the running time of a high performance parallel application, a scheduler requires deep knowledge of that application's behavior. Communication is central to the runtime of most parallel applications. The time to transmit a message heavily depends on the size of the message. The size of a message cannot effectively be extracted or estimated from an application's binary or source code. It is therefore necessary to monitor and then model an applications network communication. This information will be used to predict consecutive runs of the same applications, in an effort to predict their execution time under similar or the same conditions. To guarantee the effectiveness of those estimations it is also necessary to assess them in consecutive runs in production scenarios, capturing deviations relative to previous runs or estimations.

Outline

In the following sections, we will outline an effort towards monitoring a distributed parallel program. This will entail a discussion of monitoring mechanisms of the Linux operating system, and then a proposal of how to apply one of these to effectively monitor and model a program's communication behaviour. An implementation of that proposal will be shown and evaluated on a typical HPC application, and then on a UNIX networking utility. The evaluation will be done with respect to performance and network bandwidth impact.

Related Work

[7], [15] propose a system of resource usage negotiation which is then contractually agreed upon. To guarantee contractual compliance, a system needs to be in place, to precisely monitor a programs resource utilization. This entails computation time, memory usage, and networking behaviour. Glaß details an implementation towards guaranteeing computation time. [16] This paper will focus on network communication. Both rely on a central entity (usually the scheduler) that is responsible for handling contract breaches, and will only report to them. Meswani et al. detail an effort to modelling disk I/O of high performance applications, to then model and predict its impact on execution time. [17] NetworkCloudSim implements a general approach towards modelling of runtime behaviour of high performance in cloud applications. [18]

Requirements

The operator should be able to monitor a program, and either record its communication behaviour, or specify a past recording as reference. Deviations from a reference recording shall be detected and signalled to the kernel or another central entity (i.e.: the plan based scheduler). Programs should not need customization (e.g.: instrumentation, recompilation etc.) to be measured. The mechanism for monitoring should also not interfere with the programs control flow as well as not impede its execution performance in a meaningful way. Interaction with the monitoring program should be simple and make as much use of existing conventions of the target platform as possible, to be easily adoptable into existing solutions. The implemented program should be well documented and extensible. There should be well defined interfaces to implement new monitoring mechanisms. The implemented program should have well defined interfaces for extending its monitoring capabilities. There exists a clear interface to write and read program communication patterns.

Evaluation Criteria

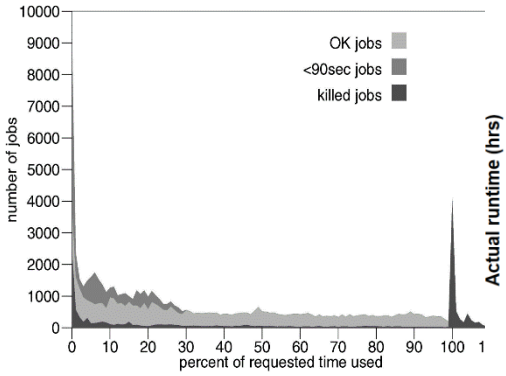

Figure 13) Runtime estimations relative to actual exection time. [19]

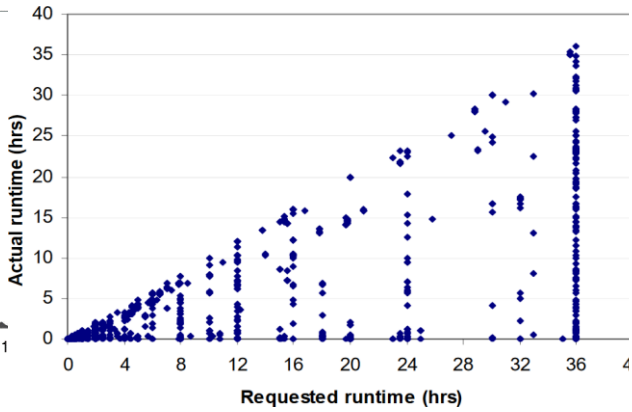

Figure 14) Runtime estimations compared to actual execution times. [20]

[19] analyses and then models the relation of user provided runtime estimates to the actual running time of application: they find that 10% of estimations are shorter than the actual running time, and the remaining 90% are evenly randomly distributed. [6], [20] compare user provided runtime estimates with actual runtimes and find that estimates are off by 57%. [21] observe that long running jobs tend to have more precise estimates; however note that this might be an artifact of the long jobs maxing out the allocated running time of the system. While this data does not provide us meaningful insights into acceptable margins of performance reduction, we can still assume that users will not notice marginal slowdowns.

We therefore require that:

- An applications execution time increase should not exceed 5% of its original execution time, on the same system without monitoring.

- The average network throughput of applications should not degrade by more than 5% of the baseline version, due to monitoring.
- No program may fail due to our monitoring activities — except when it does not behave according to the initial plan.

A Linux Kernel Module for Monitoring — General Implementation overview

The following sections are a discussion of mechanisms for monitoring a programs' network communication. Following that is an outline and proposal of a mechanism to monitor communication of parallel programs.

A solution needs to capture a processes' communication, then either store it for later usage, or compare it to past communication behaviour, if it does not conform, the violation needs to be reported. In addition, it should conform to our requirements and evaluation criteria outlined in REQUIREMENTS.

The Linux Kernel

At this time, Linux is the only operating system used in HPC, as reported by the Top500[17]. The benefits of using Linux for high performance computing are that it is highly portable, easily extensible, and allows for fine grained tuning towards specific use cases (e.g.: desktop computers, real-time computing, secured computing, or high performance computing). [22] Most of these properties derive from the

[17] https://www.top500.org/statistics/overtime/

operating system being developed and maintained completely open source, which allows users to alter the source code to their own requirements and also contribute these customizations upstream[18].

Monitoring and Probing in the Linux Kernel

Monitoring should be done in such a way that it does not interfere with the programs execution. A program should not need modification to be monitored, neither source code, nor binary or require a custom tool to be made compliant. [17] It may not significantly affect the programs' performance. It should be simple to implement and rely on established and stable mechanisms.

The Linux Kernel offers many ways of monitoring program activity at multiple levels, which will be expanded upon below.

eBPF / BCC

The most promising solution for safe monitoring is eBPF - extended Berkeley Packet Filters. It has been developed as an extension to the linux kernel and has been fully integrated since 2017. eBPF defines a virtual machine and language that can be injected into the kernel at predefined points. It has strict safety and termination guarantees, as well as a host of built-in kernel/user-space communication primitives. It is specifically designed to allow monitoring and probing of the kernel.

While eBPF has ostensibly matured to a critical point where it is usable, it is still not trivial to use effectively. [23] It requires third party solutions like BCC - the BPF Compiler Collection - for common use

[18] See eBPF or Linux Realtime, which were both maintained as kernel patches until a point of maturity, where they were then incorporated into the main tree.

cases, which at the time of writing is mostly written for usage with Python. Further investigation revealed that writing for eBPF is severely limited, bug ridden, error prone and the programmer is still required to have extensive knowledge of the execution context. Overall eBPF seems like it would be the perfect candidate for our problem, but critical usability problems and its relative stage of infancy led us to decide against using it.

LibC *shim*

Library shimming is a technique, wherein libraries are transparently replaced with other implementations that eventually delegate to the original library. This can be used to intercept calls to the original library by specific programs. This is a powerful tool to insert logging behaviour, fix bugs, or add permission layers. On Linux based systems shims can be installed by altering the environment variable *LD_PRELOAD* to contain the library-shim.

Libc is the C standard library, containing implementations of essential functionality for everyday C development, making it an ideal candidate as a non-intrusive entrypoint for program monitoring. It is also the API layer on top of Linux syscalls, which are the only way a user space program can directly interact with the operating system.

Not all programs and programming languages necessarily use libc as their mechanism for invoking syscalls (e.g.: the programming language Go), which would make them unable to be monitored. Even programs using LibC might be statically compiled and thus unavailable for shimming. It would also require a service to be communicated with and thus introduce more communication and a spread out implementation.

LTrace / STrace / PTrace

LTrace and STrace are both linux applications meant for monitoring programs and their execution. LTrace is used to observe the usage of library calls and their parameters by an application. STrace works in a similar fashion but for syscalls. Both are very useful for debugging and monitoring programs in development but have been shown to add significant overhead to production systems: a monitored application can run up to 442times slower. [24] From the STrace manual: "Known Bugs: a traced process runs slowly." They additionally emit data in text form, requiring additional processing.

STrace and LTrace have nonetheless proven essential for this thesis, as they enabled the analysis of running processes in detail to find the library- and syscalls of programs used for communication.

PTrace is a linux built-in mechanism for process tracing, it defines the syscall *ptrace* which can be used for setting up a tracing mechanism and allows the installation of probing points at system calls. However, it adds additional context switches to and from the traced program, to the tracing program and back, adding a non-trivial overhead to the programs' execution, making it a suboptimal solution for HPC applications.

KProbes

Linux KProbes are a kernel mechanism to inject probes at arbitrary positions in the kernel, they can either target a specific address or the symbolic name of a kernel routine. [25] When the probed point is executed, the execution is trapped into a user specified function and it's execution context (ie.: the register's content) is passed to the function, where it can be read and manipulated. KProbes also allow

probing when returning from that address, then called KRetProbes with limited concurrency.

The default KProbe implementation is implemented by setting a CPU breakpoint at the probed address into a function which uses a *notifier_call_chain* to execute all attached probes, after that the probed instruction itself is executed and then the return probe via the same mechanism. Breakpoints are relatively expensive, which is why they can be replaced with a jump instruction instead through optimization flags, which will then ostensibly have performance similar to normal library invocations.

KProbes are very deep in the kernel structure and their usage is therefore partially architecture dependent: the naming of registers may differ between architectures and is reflected in the *registers* struct passed to the probing function. Additionally: "Since it operates on a running kernel and needs deep knowledge of computer architecture and concurrent computing, you can easily shoot your foot." [25] The KProbe will also be on the path for every process initiating that routine and will therefore impact their performance. Additionally, there exists a trend in high performance computing to implement networking and file system drivers in user space, this has the benefit of removing syscalls from the critical path, being easier to debug and not interfering with other critical resources, this poses a challenge to monitoring with KProbes as they will no longer be triggered.

The kprobe manual contains some performance measurements: an unoptimized KProbe adds 0.99 µsec, while an unoptimized Kprobe adds 0.06 µsec to each invocation on the evaluation machine.

KProbes were chosen as the mechanism for monitoring because they have a well-defined interface and are simple to set up and tear down.

They also allow probing of arbitrary points in the kernel which makes them ideal for a system that aims to be flexible and extensible. Also, since this thesis aims to verify the modellability of parallel programs' communication behaviour, the mechanism employed does not necessarily need to be the safest or cleanest, but instead enable fast iteration. In our case study below, we will also review if our implementation has a meaningful impact on program performance.

Linux Kernel Module — MoNet[19]

The Linux kernel allows the extension of its functionality through kernel modules. Kernel modules can be dynamically loaded and unloaded at runtime without the need for restarting the system. Kernel modules will be executed in kernel space and have complete read and write access into the kernel space, allowing them to alter the content and behaviour at central data structures. Kernel modules are usually used to implement hardware drivers or to add new behaviour to the kernel. Kernel modules will be used as a means for easy access to user and kernel space information of the monitored programs.

A Linux kernel module was developed monitoring syscalls via kprobes. The Kprobe routine tests if the routine was triggered from a monitored program by checking the global variable *current->tgid*[20]. If it is not a monitored process, we resume the kernel routine, else we access the registers containing message length information and copy them onto

[19] MONitoring NETwork communication

[20] *Current* is the CPU-local global variable holding information about the current running process. *tgid* is the processes' thread group, which can differ from the process id (*pid*), when running a multiprocess program.

a FIFO-queue, then resume the kernel routine: we do not process the data at this point, as we want to resume execution as soon as possible.

The communication data is processed in a separate kernel worker thread which extracts the message lengths, this depends on the specific function the target process used for communication. When the kernel module is set to recording, it will store the message lengths sequentially in a buffer, if it is in restricting mode, it will compare the message lengths to user supplied lengths. Deviations from the input data are written to the kernel logs as errors and signalled to a central authority.

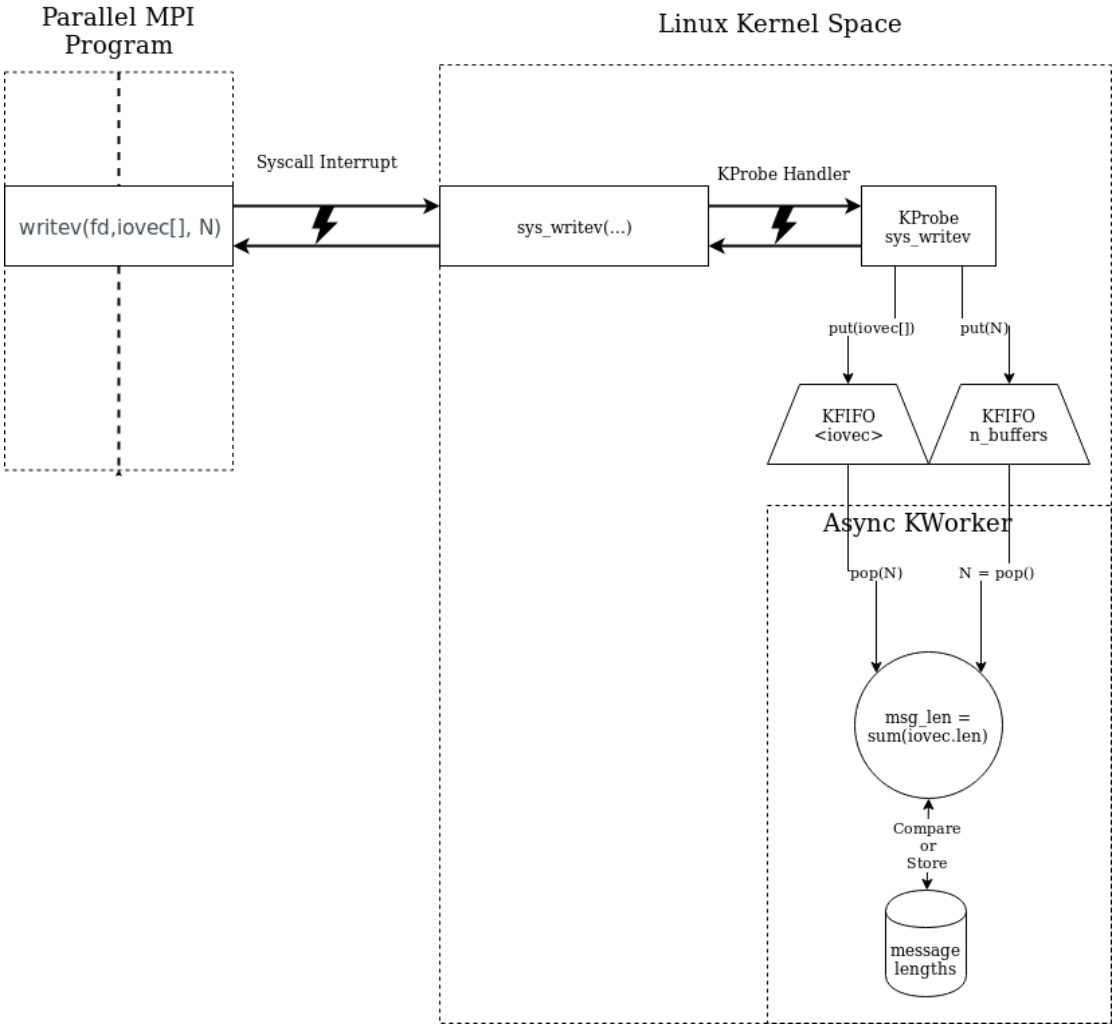

Figure 15) Program and data flow of monitoring with monet. This is for the syscall writev, which as will be outlined below used by OpenMPI to communicate when configured to use TCP.

The recorded data can be read from a character device at */dev/monet*, the message lengths will be output in csv format: the first column corresponding to the index of the message, the second column to the actual length in bytes.

The character device can also be written to, it is used for receiving previously measured message lengths in the same csv format as above (*id, length*; no header), but also for configuration purposes, to transmit

the pid of monitored process and whether the module should be recording data or comparing it, monitoring can also be turned off.[21]

Discussion / Evaluation

In the following section, two case studies will be presented: 1) a High Performance MPI application will be executed and monitored, this is our target domain and should demonstrate the capabilities and limitations of the developed tool. It will also showcase how the module can be set-up and utilized in a practical way. Additionally, it will be used in assessment of our evaluation criteria; 2) the command line tool cURL will be monitored to demonstrate the customizability of the developed module, as it relies on different communication primitives than OpenMPI. It will also be used to evaluate the impact of monitoring on throughput of machine-local network communication.

Execution Environment

The programs are executed on a desktop computer, running a fully updated Linux Mint 19.1, using the Linux kernel 4.15-20 (28 Jan 2018). It has a 4 core Intel Core i5-3450 CPU @ 3.10GHz processor, and 12GB RAM. They are not executed on a distributed cluster, with a wired connection, as that would complicate development and additionally, as wired communication is orders of magnitude slower than local communication this scenario will highlight the impact of monitoring, giving upper bounds on performance impact.

[21] For more details see the appendix.

Case Study 1: Performance Impact on PlaSim

This case study will be used to verify our evaluation criteria outlined in EVALUATION CRITERIA. Most important, it will be used to evaluate the performance impact on a high performance system. This thesis is specifically targeted at distributed high performance programs and making it crucial to test this application in a realistic scenario.

As outlined above, MPI is widely used in high performance parallel computation and is therefore ideal as a testing ground for this thesis. OpenMPI is an open source implementation of the MPI message passing interface standard.

Program Selection

The program to be executed in this case study should be a realistic representation of HPC applications, therefore it should put high demands on its execution environment and utilize multiple MPI communication primitives. Some programs use MPI only as a scheduling mechanism for cluster systems, these would not produce interesting communication patterns for us to observe.

As outlined above PlaSim is a program that does fulfill the necessary criteria on high performance applications and is extremely simple to set up, the case study will therefore be done using PlaSim. A single parallel program is not representative of the whole domain of high performance programs, however as we aim to understand the impact and feasibility of monitoring and modelling a programs' network communication, I consider it sufficient to analyse only one complex program and generalizing from there.

PlaSim's MPI usage is abstracted into a single library/file[22] containing all the routines to send the particular data to be sent, this allows the authors to change communication code easily, and is mostly used to change from a sequential implementation to a parallel implementation by including different files. It uses MPI_Broadcast (5times), MPI_Scatter (5times), MPI_Gather (4times), MPI_Allgather (2times) and MPI_Allreduce (3times). All of those calls are *1:n* or *n:n* communication mechanisms. The way the code is set up makes it hard to predict runtime communication behaviour, just from the source. Additionally, buffer size is dependent on the number of allocated jobs, and decided at compilation time, further limiting source code based estimations.

Evaluation Scenarios

The program will be run multiple times with different configurations and their execution times will be recorded:

1. The program will be run without the monitoring module inserted, as comparison base,
2. the program will be run with the module inserted, but not monitoring, to measure the modules performance impact on the system and other applications,
3. the program will be run with the monitoring module inserted and set to record it, to measure the performance impact it has on the monitored application,

[22] mpimod.f

4. and the program will be run with the monitoring module inserted and set to restrict, to measure the performance impact it has on the monitored application.

The collected data will be used in an evaluation of the modules impact on a high performance computing application.

OpenMPI can be configured to use many communication technologies (e.g.: Local IPC, TCP, Infiniband, DMA) and it can be expected to use different implementations for its communication in different configurations and communication primitives. It is therefore necessary to investigate which communication mechanisms are used beforehand. To this end, the MPI program is executed and traced with the above discussed program *strace* where its network communication syscalls will be captured.[23]

It was found that OpenMPI uses the *writev* syscall to communicate. *writev* is used to write data from multiple buffers sequentially into a single *file descriptor*[24]. The *writev* syscall itself triggers an interrupt into kernel mode which will call the kernel routine *sys_writev*, the architecture specific implementation for the *writev* syscall.

The *writev* syscall has the following interface:

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

[23] For a deeper explanation of how to discover the communication primitives, I refer to the Case Study on cURL below.

[24] A file descriptor does not necessarily imply a file, but is instead the base mechanism used by the Linux kernel to identify a source or destination for data. In this case it is a socket connection to the target node.

Writev receives three input parameters:

fd — the file descriptor of the file to be written to,

**iov* — a pointer to an array of iovec structs which themselves hold a pointer to a target address and the length to be written,

iovcnt — the length of the array at *iov.

As we want to avoid computation as much as possible in the kprobe routine, the array at *iov* is simply copied from user space onto a kernel-side fifo-queue[25] using the function *kfifo_from_user*. In addition, the length of the array *iovcnt* is pushed onto a seperate queue, which is then used in the kworker to compute the actual length of the message by summing over the *iovec#vlen* fields. The lengths are now either stored or compared to their supposed input. If a deviation is detected, we write out a message onto the kernel logs and send a signal to the central authority, reporting it.

Coverage of the Case Study

The case study will test if our monitoring efforts produce reproducible results over multiple executions of the same HPC application. They will also reveal the performance impact on a long running high performance application which is our target domain. It will also be tested how easy it is to setup and monitor a distributed application, which is crucial from a production standpoint.

[25] Found in <linux/kfifo.h>

Setup and Monitoring

Before running our application, the monitoring kernel module will need to be successfully compiled and inserted on each node.

As we need the target processes' pid for monitoring, an additional helper program was written: right after starting the program pauses itself, once resumed by an external signal, it will replace itself with a command line supplied program. This allows us to capture and communicate the executed programs pid before its execution.[26]

Reading Data

After successful execution of PlaSim, we can extract the recording from MoNet by reading the device at */dev/monet*.

```
$ sudo head /dev/monet
0,104
1,28
2,28
3,10184
4,1018
```

On average PlaSim sends 7.59GB over the span of 7:40min, with exactly 29,575,818 Messages, of an average size of 256.79byte. To recall from above: the signature of *writev* allows sending multiple buffers at once which are then concatenated.

[26] It is entirely possible for the monitored program to communicate its own pid to the module and set itself up for monitoring. However, as one requirement to this thesis a non-intrusive implementations, this route was chosen.

Performance Impact of MoNet

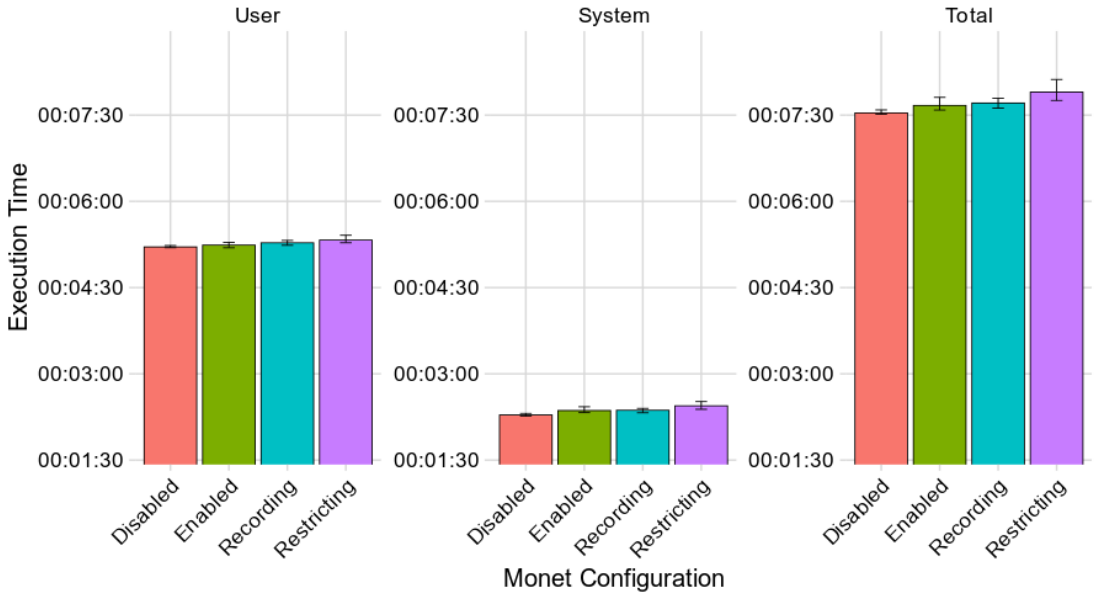

Figure 16) Absolute execution time of PlaSim, in all four configurations, for a one year simulation. User is the time spent computing. System is the time spent in Syscalls. Total is the total execution time.

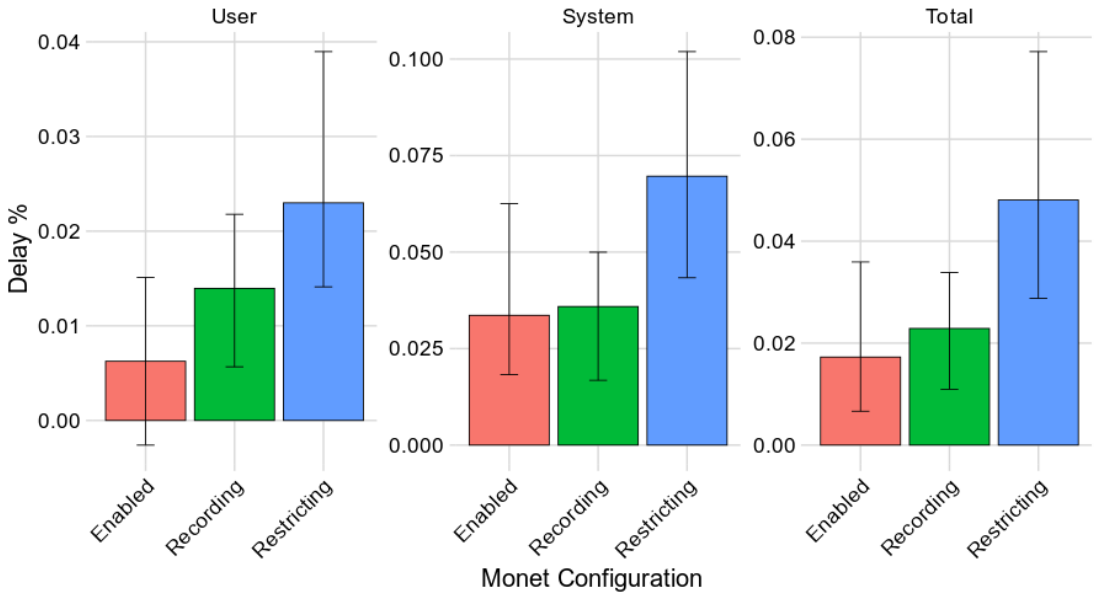

Figure 17) Execution time of PlaSim, relative to Disabled, in all four configurations, for a one year simulation. User is the time spent computing. System is the time spent in Syscalls. Total is the total execution time.

To measure the performance impact of MoNet, PlaSim was executed on the outlined hardware, in the above outlined configurations,

simulating 4 years. The execution times are measured using the unix *time* command, which not only measures raw execution time, but also time spent in syscalls. Before each execution, the *stress* utility will be run for 60s to put the system into similar initial conditions before execution. In addition, its working directory will be completely reset before execution.

Data Evaluation

CONFIGURATION	MEAN	MIN	MAX
Disabled	7:32 (+ 0.00%)	7:30	7:35
Enabled	7:40 (+ 1.73%)	7:35	7:48
Recording	7:42 (+ 2.29%)	7:37	7:47
Restricting	7:53 (+ 4.81%)	7:45	8:07

PlaSim total execution time

CONFIGURATION	MEAN	MIN	MAX
Disabled	2:17 (+ 0.00%)	2:15	2:18
Enabled	2:21 (+ 3.36%)	2:19	2:25
Recording	2:22 (+ 3.59%)	2:19	2:23
Restricting	2:26 (+ 6.97%)	2:23	2:31

PlaSim time spent in syscalls in Minutes

Simulating a year without monet inserted takes on average 7:32 minutes. With MoNet inserted, but not monitoring, this increases to 7:40min (+ 1.7%). When monitored, PlaSim takes on average 7:42min (+ 2.28%). Restricted PlaSim now takes 7:54min (+ 4.8%).

The time spent in syscalls is not significantly different, meaning that the additional work to store the message lengths when recording does not cause a significant slowdown. It is therefore surprising that the slowdown doubles when in restricting mode, as the kprobe's code is the same between restricting and recording. Additionally, restricting has the highest variance: having a minimum slowdown of 2.9% and a maximum of 7.72%.

The most likely explanation is the fact that MoNet's worker thread has more complex work to do than when set to recording: it has to fetch the old values, do a comparison and some computation on them. It will also log every deviation to the kernel logs with *printk*, which as per the kernel manual is a relatively expensive computation. This can also be observed in the CPU utilization measurements: in all other configurations, PlaSim utilises at least 396% of the CPU (> 99% of 4 CPUs), however when restricted it now is only able to utilise at most 395% (98.75% of 4 CPUs) and on average only 393% (98.25% of 4 CPUs).

Do MPI Programs repeat themselves?

It is crucial to understand if and how a repeat execution of a parallel program differs with regard to parallel communication. The assumption is, that a program running under the same configuration and same conditions will behave exactly the same, and have the exact same communication model. To verify this claim, we will run PlaSim under the same conditions[27] 8 times (two 4 year long simulations) and

[27] Set up as described in the section Setup and Monitoring. However, with a smaller grid size to have a shorter runtime.

compare the recorded lengths of the same MPI node with the recording of the first year of the first simulation.

On average PlaSim sends 6714094 messages of average size 794byte, sending 5.332GB over a whole year. The message lengths are identical over every simulation run, except for 4 messages per run, varying on average by 3.4Kbyte. This is below 10^{-6}%of the messages deviating. All deviations occur in the last 157 transmissions; however no concrete pattern is discernible, beside three years having a difference in the third last message transmission.

While there are some minor differences among repeat executions, they are relatively predictable (i.e. always exactly 4) and below a problematic margin of error. I would therefore conclude that PlaSim's communication does in fact repeat itself with regard to message size.

AVG.	YEAR 1	YEAR 2	YEAR 3	YEAR 4	YEAR 5
3.4Kbyte	6.2Kbyte	0byte	4.5Kbyte	3.4Kbyte	2.6Kbyte

Absolute difference between repeat-executions of the same year.

Case Study 2: cURL

One of the requirements of this study is flexibility of implementation, to verify this claim, a second monitoring implementation is detailed which can also be used as a blueprint for later adaption on to other programs. One of the most used networking programs on any linux system is the command line tool cURL[28]. It is used to transmit and receive data over networks implementing a large number of networking protocols. Additionally, cURL is a widely ported application

[28] https://curl.haxx.se

making it fit for verification of portability of the developed monitoring framework.

Coverage of the Case Study

The case study will give a tutorial on how MoNet can be used to monitor an application, specifically how to implement monitoring for a new syscall. In addition, it will measure the impact on raw upload bandwidth of an application with monet running.

Implementation

We begin by constructing our test application:

```
$ curl localhost:8080 -XPOST -d "Hello" -s > /dev/null
```

This will execute cURL and instruct it to send an HTTP POST request to localhost containing the message “Hello”. The response is discarded.

We will monitor cURL with strace to analyze its communication behaviour with the following command:

```
$ strace -e trace=%network -yy curl [...]
```

This will instruct strace to filter for networking related syscalls and add protocol specific information, producing the following output:

```
socket(AF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
setsockopt(3, SOL_TCP, TCP_KEEPIDLE, [60], 4) = 0
setsockopt(3, SOL_TCP, TCP_KEEPINTVL, [60], 4) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(80),
```

```
sin_addr=inet_addr("174.129.224.73")}, 16) = -1 EINPROGRESS
(Operation now in progress)
getsockopt(3, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
getpeername(3, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("174.129.224.73")}, [128->16]) = 0
getsockname(3, {sa_family=AF_INET, sin_port=htons(57290),
sin_addr=inet_addr("192.168.178.41")}, [128->16]) = 0
sendto(3, "POST / HTTP/1.1\r\nHost: localhost"..., 156,
MSG_NOSIGNAL, NULL, 0) = 156
recvfrom(3, "HTTP/1.1 404 Not Found\r\nContent-"..., 102400,
0, NULL, NULL) = 132
+++ exited with 0 +++
```

Most of the calls made are for setting up a network connection, then *sendto* is invoked. The string preview shows that the data sent is HTTP protocol specific, leading to the assumption that cURL uses *sendto* for sending data via POST. The linux manual explains that *sendto* is "used to transmit a message to another socket." The signature of *sendto* is:

```
ssize_t sendto(int socket, const void *message, size_t
length, int flags, const struct sockaddr *dest_addr,
socklen_t dest_len);
```

Where

socket — the file descriptor of the open socket

*message — start address of the message to be sent

length — length of the message to be sent

flags — optional protocol specific flags

*dest_address — address of the destination of the message

dest_len — protocol specific length of *dest_address

The return value is the number of actually written bytes[29], or a negative value in case of error.

The message length is extracted by reading *length*, the third parameter.

Implementing a probe for sendto requires implementing the following functions:

1. *static void monet_init_kprobe(struct kprobe* probe)*
   Is invoked at module initialization and used to configure the kprobe for monitoring *sendto*.
2. *static void monet_kprobe_handler(struct pt_regs *regs)*
   Which is invoked inside the kprobe handler, if the kernel module is set to monitor communication and the monitored application has invoked *sendto*. The input parameters are the register state of the calling thread, we can use those to retrieve the message length: the length of the message is stored in *sendto*s' third parameter and will therefore be stored in the third register *rdx*. The length will be pushed onto a kernel fifo queue and processed later.
3. *static bool monet_has_data(void)*
   Is invoked when the kernel worker thread is scheduled to store or compare data read in the kprobe. It is used to signal that there is unprocessed data.
4. *static void monet_kprobe_extract(void)*
   Is used to transmit the data for processing, where it will either be compared or again stored. It needs only to invoke the function *receive_entry*, which will then appropriately be handled.

[29] Which will not differ from the actual length, it is supposed to fail with a more specific error code instead.

Impact on Average Throughput of cURL

In the following we will evaluate the impact of MoNet on the self-reported average upload speed of cURL, when transmitting a 10GB of randomly generated data. The data is generated by reading */dev/urandom* which will generate a random stream of data without blocking. The unix tool *head* will be used to read 10GB from it. The data will then be used to send data as a POST HTTP Request to a locally running nginx instance that will immediately discard the data.

```
$ head -c 10GB /dev/urandom | curl -v -w
"UPLOAD=%{speed_upload}\n" -XPOST -d@- localhost:8080/
```

We evaluate the following scenarios:

1. Monet not running, as baseline

CONFIGURATION	AVERAGE UPLOAD	EXECUTION TIME	TIME IN SYSCALLS
Disabled	1.375 GByte/s (+ 0.0%)	9.88s (+ 0%)	9.36s (+ 0%)
Enabled	1.317 Gbytes/s (- 3.0%)	10.11s (+ 2.3%)	9.37s (+ 0.01%)
Recording	1.284 GBytes/s (- 5.4%)	10.11s (+ 2.3%)	9.58s (+ 2.4%)

Impact of monitoring on throughput of cURL, when sending 10GB of data.

2. MoNet running, set to record[30], to measure the impact on throughput on the monitored application,
3. MoNet running, not monitoring cURL, to measure throughput impact on an application not monitored but using *sendto*.

We measure a 5.34% throughput degradation on cURL monitored with MoNet, and a 3.0% degradation, when monet is running but not monitoring. Meaning, that a monitored application uploading data for 60 minutes will now run 3.2 minutes longer, and an application on a system with monet installed but not monitoring will now run 1.8minutes longer.

Future Work

As the KProbe interface is implemented directly on the register contents of the application, it is partially architecture dependent. This limits easy adaption onto other systems. However, the changes to the source code would be minimal.

The current implementation of MoNet only allows monitoring of one application at a time and one syscall. It is feasible to implement monitoring of multiple syscalls and also of multiple applications, but would require extensive refactoring of the current implementation. The current implementation of MoNet statically pre-allocates about 500MB of memory to store measurements, this only increases with more applications/syscalls and would require dynamic management.

[30] Since the critical path for restrict and record is identical and we are not performing a compute intensive task, only record is measured.

Additionally, the current implementation of the MoNet worker thread uses polling to check the queue, meaning that it will be run and scheduled even when it does not have work to do, which wastes some computation time. There are already mechanisms present in the linux kernel to handle these scenarios, though they have not yet been applied to MoNet (e.g.: spinlocks and thread parking). The worker thread seems to generate the most overhead of all components, and should therefore be optimized further. It could also be investigated, if the worker thread is strictly necessary or if the work could be pulled into the kprobe's logic.

The current implementation of MoNet relies on three wrapper scripts to execute and monitor an MPI application:

```
mpiexec [...] ./run-on-node.sh 2 "./trace-me.sh ./plasim.x"
./plasim.x
```

Where *run-on-node.sh* executes different applications depending on the MPI node, and *trace-me.sh* manages sending the pid to MoNet, and utilizes a third program to start/stop the monitored application. This is obviously very complex to setup and could be simplified. Part of this is down to the fact that the device */dev/monet*, can only be written to by the root user and the script *trace-me.sh* handles that. If we allow non-root users or a specific user group to write to */dev/monet*, this could be simplified, such that the wrapper program sends its own pid to */dev/monet*.

It should also be possible to encode more complex models than just static models. For example, already encoding uncertainties at some points in the model, or allowing variance etc. A completely different approach could also be modelling the applications throughput over

time ($\frac{\frac{MByte}{s}}{min}$), this could be useful in scenarios where the ordering of messages is not stable, but the amount of communication is.

The second case study evaluates the impact on throughput of curl. While measuring the impact on throughput of an application it does not measure the impact on throughput of a high performance application. While this is a useful metric, the work done to extract message lengths for MPI applications is more complicated, it should therefore also be evaluated in future work.

Closing Discussion

When evaluating PlaSim's repetition with regards to message lengths, we found virtually no deviations to past executions. Meaning that a representative high performance application can effectively be modeled based on past executions. The observed deviations all happened within the last few messages of the applications execution lifecycle and are therefore most likely used for gathering the results of the computation. It should be noted that new features in MPI2 allow dynamic allocation of nodes and therefore pose a challenge when modelling them. [26]

The performance impact of monitoring does on average not exceed our limit of a 5% slowdown. However, restricting did in fact cause a maximum delay of 7.7%. The variance when restricting - which would usually also mean a production environment - is not optimal and needs to be investigated further. It is noteworthy, that the additional work done in the kprobes does not add a significant amount of overhead (Enabled vs Recording), and it should therefore also be investigated how much work can be refactored into the kprobe itself further

reducing the complexity of the kernel module. The impact of monitoring on raw network throughput (5.34%) does also exceed our 5% limit. It should however be noted, that this is on a machine local connection, meaning that the performance degradation is a strict upper bound to wired connections, where network latencies are the dominant component.

Appendix

Interacting with the Kernel Module

All interaction with the kernel module is accomplished by writing to the kernel modules character device /dev/monet.

Changing Recording Modes

```
$ echo MODE=OFF > /dev/monet  # to turn off monitoring
$ echo MODE=RECORD > /dev/monet  # to turn monet to recording
$ echo MODE=RESTRICT > /dev/monet  # to turn to restricting
```

Passing the Process ID

```
$ echo PID=$PID > /dev/monet
```

Reading recorded message lengths

```
$ cat /dev/monet
```

Writing a single length restriction

```
$ echo "0,506\n" > /dev/monet
```

Writing a whole file of length restrictions

```
$ lbl sizes.csv > /dev/monet
```

Lbl is a custom utility writing a file line by line into the standard output or a supplied file. MoNet can only read files line by line. Using the *cat* utility therefore does not work.

line-by-line.c:

```
[...]

while ((read = getline(&line, &len, in)) != -1) {
    fprintf(out, "%s\n", line);
    fflush(out);
}

[...]
```

References

[1] E. Strohmaie, J. Dongarra, H. Simon, and H. Meuer, "Top500," 2019. [Online]. Available: https://www.top500.org/. [Accessed: 10-Sep-2019].

[2] M. Paulitsch, E. Schmidt, C. Scherrer, and H. Kantz, "Industrial applications of High Performance Computing," *Time-Triggered Commun.*, pp. 303–359, 2011.

[3] L. L. N. Security, "Batch System Primer," *Lawrence Livermore National Laboratory*. [Online]. Available: https://hpc.llnl.gov/banks-jobs/running-jobs/batch-system-primer. [Accessed: 10-Sep-2019].

[4] J. Yu and R. Buyya, "A taxonomy of workflow management systems for Grid computing," *J. Grid Comput.*, vol. 3, no. 3–4, pp. 171–200, Sep. 2005.

[5] Y. Yuan, G. Yang, Y. Wu, and W. Zheng, "PV-EASY: A strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling," *HPDC 2010 - Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, pp. 240–251, 2010.

[6] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 6, pp. 789–803, 2007.

[7] L. Burchard, M. Hovestadt, O. Kao, A. Keller, and B. Linnert, "The Virtual Resource Manager : An Architecture for SLA-aware Resource Management Faculty of Computer Science and Electrical Engineering."

[8] C. Minkenberg, "Interconnection Network Architectures for High-Performance Computing HPC interconnection networks," *Adv. Comput. Networks*, no. May, 2013.

[9] Konstantin S. Solnushkin, "Fat-Tree Design," *clusterdesign.org*. [Online]. Available: https://clusterdesign.org/fat-trees/. [Accessed: 10-Sep-2019].

[10] Wikimedia, "File:Dragonfly-topology.svg," *Wikimedia Commons, the free media repository.* 2017.

[11] R. L. Walko, "OLAM-Soil." .

[12] R. L. Walko and R. Avissar, "The Ocean-Land-Atmosphere Model (OLAM): a formulation for high resolution weather and climate simulation," in *AGU Fall Meeting Abstracts*, 2008.

[13] F. Lunkeit, K. Fraedrich, H. Jansen, A. Kleidon, U. Luksch, and E. Kirk, "Planet Simulator Reference Manual," 2007.

[14] L. I. N. Architecture, "SIMULATION IN ARCHITECTURE , Structural design for fabrication white paper," 2010.

[15] L. O. Burchard, B. Linnert, and J. Schneider, "A distributed load-

based failure recovery mechanism for advance reservation environments," *2005 IEEE Int. Symp. Clust. Comput. Grid, CCGrid 2005*, vol. 2, pp. 1071–1078, 2005.

[16] K. Glaß, "Plan Based Thread Scheduling on HPC Nodes," 2018.

[17] M. R. Meswani, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling and predicting Disk I/O time of HPC applications," *Proc. - 2010 DoD High Perform. Comput. Mod. Progr. Users Gr. Conf. HPCMP UGC 2010*, no. May, pp. 478–486, 2011.

[18] S. K. Garg and R. Buyya, "NetworkCloudSim: Modelling parallel applications in cloud simulations," *Proc. - 2011 4th IEEE Int. Conf. Util. Cloud Comput. UCC 2011*, pp. 105–113, 2011.

[19] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, 2001.

[20] C. B. Lee, Y. Schwartzman, J. Hardy, and A. Snavely, "Are user runtime estimates inherently inaccurate?," *Lect. Notes Comput. Sci.*, vol. 3277, no. March, pp. 253–263, 2005.

[21] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Modeling user runtime estimates," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3834 LNCS, pp. 1–35, 2006.

[22] T. P. Morgan, "One Linux Stack To Rule HPC And AI," *Next Platform*, 2018.

[23] J. Dileo and A. Olsen, *Kernel Tracing With eBPF - Unlocking God Mode on Linux*. Chaos Computer Club, 2018.

[24] G. Brenadn, “strace-wow-much-syscall,” *www.brendangregg.com*, 2014.

[25] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu, “KProbes - Linux Documentation.”

[26] M. C. Cera, G. P. Pezzi, M. L. Pilla, N. Maillard, and P. O. A. Navaux, “Scheduling Dynamically Spawned Processes in MPI-2,” in *Job Scheduling Strategies for Parallel Processing*, 2007, pp. 33–46.

[27] H. A. Council, “Interconnect Analysis : 10GigE and InfiniBand in High Performance Computing,” *Case Stud.*, 2009.