SooMPI - Sockets over MPI

André Lisboa Oliveira Costa

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. João Nuno de Oliveira e Silva

Examination Committee

Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. João Nuno de Oliveira e Silva
Member of the Committee: Prof. João Carlos Antunes Leitão

November 2017
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

First and foremost I would like to express my sincere gratitude, to the love of my life, Joana Santos, the biggest thanks for all your unconditional love and support since the beginning and even before my academic formation.

To the rest of my family, a big thanks for all of their support and all the good times spent together. To my awesome Sister, Sara Lisboa, that always found a way to cheer me up. My lovely Mother, Graça Oliveira, that helped me in this journey with everything I needed and even more. To my super Father, António Pedro Costa, for all his continuous encouragement. To my Uncles, Jorge Lisboa and Elisabeth Sousa, for their always super positive mood. Lastly, but not least to my Grandparents, Maria Adelaide Costa, António Costa, Maria Helena Oliveira and António Oliveira for their caring support and kindness.

I also would like to offer my deepest gratitude to my thesis supervisor Professor João Nuno Silva, for his constant support and opportunity to develop my work in an on and off basis, enabling me to continue developing my professional career and finish my academic goals in parallel. All his patience and prompt availability were crucial for both motivation and guidance to continue forward.
Resumo

O sistema SooMPI foi desenvolvido de forma a colmatar a necessidade de executar programas baseados no uso de sockets directamente em clusters de alta performance (HPC). Este sistema torna possível a execução de programas desenvolvidos em C/C++, funcionando sobre Message Passing Interface (MPI), e assim permitindo o desenvolvimento e investigação para sistemas de simulação de computação distribuída sobre sistemas de alto desempenho.

SooMPI permite a transformação de uma ou mais aplicações desenvolvidas em sockets para uma aplicação baseada exclusivamente em MPI, podendo assim ser executada em clusters HPC. Esta transformação é feita ao nível de todas as funções do API sockets e permite, não só a utilização dos endereços de Internet Protocol (IP) do cluster na interface MPI, mas assim como associar cada uma das aplicações transformadas a cada um desses endereços de IP. SooMPI foi analisado em dois ambientes computacionais com distintas aplicações e os resultados mostram que mesmo os sistemas mais complexos podem ser convertidos e utilizados sobre MPI.

A análise deste sistema garantiu o seu sucesso para a utilização de um código base em vários ambientes sem dificuldades e também a sua implementação em simuladores de rede compatíveis com clusters de alta performance.

Palavras-chave: Programação Socket, Programação MPI, Simulação de rede, Transformação de Código, Ambientes de execução.
Abstract

In order to mitigate the lack of ability to execute socket based programs directly in High-performance Computer (HPC) clusters, we developed SooMPI, that allows C socket programs to work over Message Passing Interface (MPI), thus opening the HPC environment to Network and Distributed Systems research areas.

SooMPI allows the transformation of regular network socket applications into a MPI based one, in order to execute them in HPC cluster. It converts regular socket functions and allows for the assignment of Internet Protocol (IP) addresses to the available MPI processes corresponding to each converted application. SooMPI was evaluated in two different computational environments and in a complex application, Apache Thrift. The results show that complex systems can be seamless transformed and executed in multiple MPI based environments.

This system’s analysis has been proved to be a successful mechanism for a single base code to be executed in multiple environments effortlessly. SooMPI will be used as a foundation for more efficient network simulators capable to execute in HPC clusters.

**Keywords:** Sockets programming, Network simulation, MPI, Code transformation, Execution environments.
# Contents

Acknowledgments .......................................................... v
Resumo ........................................................................ vii
Abstract ........................................................................ ix
List of Tables .................................................................. xiii
List of Figures .................................................................. xv
Glossary .......................................................................... xvii

1 Introduction .....................................................................
   1.1 Motivation ........................................................... 1
   1.2 Contribution ........................................................ 2
   1.3 Thesis Outline ....................................................... 3

2 Background ......................................................................
   2.1 Computational Infrastructures ................................. 5
      2.1.1 NOW - Networks of workstations .................. 5
      2.1.2 HPCC - High-performance Computer Clusters .......... 6
      2.1.3 Job Managers ............................................... 7
   2.2 Network Simulation ............................................. 7
      2.2.1 Parallel Network Simulation ........................... 9
   2.3 Parallel Programming and Process Communication in HPC 10
      2.3.1 OpenMP ...................................................... 11
      2.3.2 Message Passing Interface - MPI .................. 11
      2.3.3 Socket in HPCC ........................................... 12
   2.4 Program Transformation ....................................... 13
      2.4.1 Reflection ................................................... 13
      2.4.2 Code transformation ..................................... 13

3 SooMPI : Socket over MPI .............................................
   3.1 Objectives .......................................................... 15
   3.2 Solution Overview .............................................. 16
   3.3 SooMPI Architecture .......................................... 17
   3.4 SooMPI Implementation ....................................... 18
3.4.1 SooMPI Code Transformation ........................................ 18
3.4.2 SooMPI Library ...................................................... 20
3.4.3 SooMPI Function Implementation .................................. 21
3.4.4 Library Support Structures and Functions .......................... 23
3.4.5 SooMPI Node Configuration File .................................... 24
3.5 SooMPI Communication Process ....................................... 25

4 SooMPI Evaluation ......................................................... 29
  4.1 Test Environments .................................................... 29
  4.2 Functionality .......................................................... 30
  4.3 Suitability .............................................................. 30
    4.3.1 Apache Thrift .................................................. 31
    4.3.2 Apache Thrift with SooMPI Implementation .................. 32
    4.3.3 Apache Thrift with SooMPI Performance ..................... 33
  4.4 Correctness ......................................................... 35
  4.5 Performance ......................................................... 36
    4.5.1 SooMPI Communication Overhead impact on HPC Cluster .... 37

5 Conclusions .............................................................. 39
  5.1 Future Work ......................................................... 39

Bibliography ..................................................................... 41
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Time of execution of each client for both Apache Thrift implementations comparison</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Time of execution of 19 clients for both Apache Thrift implementations comparison</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Time of execution of each Test Program for performance analysis</td>
<td>36</td>
</tr>
</tbody>
</table>
## List of Figures

3.1 SooMPI Overview ......................................................... 16  
3.2 SooMPI Architecture ................................................... 17  
3.3 Overview of the Script Transformation .............................. 18  
3.4 Overview of the Script Transformation with several Programs 19  
3.5 SooMPI Connect header and MPI TAG .............................. 22  
3.6 SooMPI Send header and MPI TAG .................................. 22  
3.7 Library Support Structures and Functions relation .............. 23  
3.8 Structure and usage of the soompi_hosts.conf .................... 24  
3.9 Usage of the soompi_hosts.conf together with MPI machine file 25  
3.10 Connect() and Accept() temporal diagram with SooMPI ........ 26  
3.11 Send() and Recv() temporal diagram with SooMPI ............. 27  

4.1 Example of Server and Clients in Apache Thrift with SooMPI 33  
4.2 Overhead Test with 100Mbits byte at a time ....................... 37
Glossary

**API** Application Programming Interface. 1

**DM** Distributed Memory. 10

**DSM** Distributed Shared-Memory Multiprocessor. 6

**HA** High-availability. 6

**HPC** High-performance Cluster. 2

**HPCC** High-performance Computer Clusters. 1, 6

**I/O** Input/Output. 31

**IP** Internet Protocol. 1

**IPoIB** Infiniband-over-IP. 9

**LBC** Load-balancing clusters. 6

**MPI** Message Passing Interface. 2

**NOW** Networks of Workstations. 5

**OpenMP** Open Multi-Processing. 11

**QoS** Quality of Service. 6

**RPC** Remote Procedure Calls. 10

**SM** Shared Memory. 10

**SmPL** Semantic Patch Language. 13

**SSF** Scalable Simulation Framework. 10

**SSH** Secure Socket Shell. 29

**TCP** Transmission Control Protocol. 1

**WAN** Wide area network. 5
Chapter 1

Introduction

Today’s parallel communications infrastructures, particularly computer clusters, have been crucial assets for industry and scientific communities considering its capability for handling high demanding tasks processes and performance-to-cost ratio. However, they have presented distinct limitations to several areas of computer science. Distributed systems’ researchers need to directly access the IP (Internet Protocol) layer in order to run their code based on the sockets API (Application Programming Interface), using these infrastructures to its full potential.

1.1 Motivation

In Network and Distributed Systems research areas, network simulation is the main tool used to investigate behavior of a computer networks. The algorithms developed for research, use mostly the systems sockets API which directly depends on the access to the environment's IP layer.

High-performance Computer Clusters (HPCC) networks are usually designed with different abstractions than those offered by typical Transmission Control Protocol (TCP) and, despite the direct use of socket API in this environment being possible, it is not trivial since its access information, such as knowing beforehand available computer’s IP addresses, is limited and introduces connection management logic not considered on the source code.

Typically, HPCCs are managed by Job Schedulers, applications that provide control over batch jobs and distributed computing resources. These computing resources, such as computer nodes, are used according to the job’s demand or its availability. With such dynamic node allocation and considering that one node can support several processes that will share the same IP address, it is impossible to use specific IP addresses to match specific processes. Moreover, a process per node that utilize its IP Address compromises the cluster effectiveness.
In High-performance Clusters (HPCs) programming, a conventional used and de facto standard is the Message Passing Interface MPI [1], since it provides a high-level abstraction for sending and receiving information with optimized performance, while allowing the application programmer to conveniently neglect connection management in the process. With such capabilities, MPI earned a considerable reputation among HPCCs [2].

Most HPCC jobs that run MPI and do not take into account specific IP address processes but keep all MPI's socket API [3] usage hidden from the programmer due to security and performance assurance reasons. MPI uses ranks to designate senders and receivers, each one of these ranks only supports one program defined by the current job. Socket communication requires distinct protocols for proper port and address management in HPC environments that usually use advanced interconnects, such IP over InfiniBand [4] and SDP over InfiniBand[5], and do not achieve the bandwidth and latency of a exclusive MPI implementation.

In order to execute directly socket based programs or algorithms in HPC without detriment, like network simulation programs, it is necessary to offer a translation layer. This layer will allow the execution of applications developed using the sockets API calls on top of MPI calls without the need to change the socket source code or use other protocols for node management.

This translation layer, besides the transformation of the API calls, will offer a mechanism to match IP addresses to actual reserved computational nodes. This mechanism matches transformed programs to specific IP addresses for each MPI job, which can be easily allow for network topology arrangement and optimization.

1.2 Contribution

To answer the issues stated in section 1.1 the SooMPI was created. This system consists of a custom library containing base C socket functions working over exclusively MPI, a transformation script that converts every socket function into one of those in the library and a node configuration file that essentially dictates the topology to be used by this system.

The custom library communication uses a small header indicating information typically used by the socket API but not MPI so its behaviours and properties are maintained. The transformation tool can append several C socket applications into one output program running over MPI and, with the node configuration file support, can bind each original application to one IP address of the network.

This system was successfully tested in asynchronous and synchronous communication routines in HPC, where socket communication and file descriptor management were seamlessly transformed to MPI.
SooMPI has been implemented in Apache Thrift application and also tested in several different programs. These test programs covered socket connection oriented and datagram oriented implementations, and also both at the same time. The performance was slightly better in transformed programs rather than the achieved on the original programs, when tested in an Ethernet network environment. Still, this solution has a small header in each message that is most noticeable in large data throughput with small messages’ sizes or with high process usage in communication.

1.3 Thesis Outline

The present thesis is divided into 5 chapters and the remainder of this document is organized as follows:

Chapter 2 provides an overview of the background related to this thesis, different infrastructures, state of the art in network simulation, parallel network simulation and parallel programming, an overview of the MPI and socket usage in these environments and finally an analysis of possible code transformation options.

Chapter 3 presents the structure of SooMPI, its objectives and further expose its architecture, implementation of the code transformation, library and configuration node, finalizing with an overview of the SooMPI communication process.

Chapter 4 is a thorough evaluation of SooMPI in terms of suitability, in which its implementation on the Apache Thrift framework is validated according to is functionality, correctness and system performance.

Chapter 5 culminates on the conclusion of this thesis and opportunities for forthcoming work for this system.
Chapter 2

Background

This chapter provides an overview of available options in terms of computational infrastructures considered according to motivation requirements and, more specifically, the communication and simulation within HPCC, including parallel network simulation and parallel programming. Furthermore, we present the MPI standard, including its purpose, characteristics, distribution used and socket libraries behavior in HPCC. Finally, we end the chapter with the reflection method evaluation and code transformation.

2.1 Computational Infrastructures

The computer cluster’s structure is defined by a local computing system comprising a network interconnecting a set of independent computers and can range from associated clusters connected over WANs (Wide area networks) or High-speed backbone networks to common localized clusters. For scientific research, the two main parallel computing infrastructures used are NOW - Networks of workstations and HPCC - High-performance Computer Clusters (or simply HPC).

2.1.1 NOW - Networks of workstations

Networks of workstations (NOWs) [6] are a popular and cost-effective alternative to parallel computers. In these networks, processors are connected in a flexible way, using irregular topologies, allowing for incremental expansion capability. Moreover, these node’s capability can be easily increased by adding memory or additional processors.

Individual research workstations are powerful tools and are easier to integrate into existing networks rather than dedicated parallel computers. Considering that the development tools for workstations are more matured when compared to the contrasting proprietary solutions for parallel computers, Worksta-
tion clusters become a cheaper and immediate alternative to specialized high performance computing platforms.

NOWs have been evolving from distributed memory programming model to shared memory one. However, when compared to HPCC, NOWs are less tightly-coupled and not always use distributed shared-memory multiprocessors (DSM), leading to higher message latency and lower network bandwidth. With these characteristics, we can see that the workstation environment is better suited for non-communication-intensive, its LAN network has high message start-up latencies and low bandwidths.

2.1.2 HPCC - High-performance Computer Clusters

HPCCs utilize supercomputers and computer clusters to address complex computational requirements, such as applications with significant processing time or data-processing requirements. As NOWs, HPCCs are generally used in scientific research and compute-intensive situations. They use specialized multi-processors with custom memory architectures, highly-optimized for numerical calculations, and have a high-degree of internal parallelism.

To take advantage of these characteristics, HPCCs typically use Gigabit Ethernet over a dedicated system, having a significant niche in the high performance computing space with nearly 50% of the systems on the TOP500 list [7], so Quality of Service (QoS) demands are not typically a concern. Furthermore, it is supported by advanced interconnects like Infiniband, Myrinet or Quadrics [8]. These interconnects work with very high throughput and very low latency and are used for data interconnect, both among and within computers, requiring low processing overhead and ideal for carrying multiple traffic types.

There are several cluster categorizations worth mentioning:

- **High-availability (HA) clusters** - which are implemented mainly for the purpose of improving the availability of services provided by the cluster, but not commonly used in research since it relies mostly on a server base model;

- **Load-balancing Clusters (LBC)** - that operate by having all workload come through one or more load-balancing front ends and then distribute it to a collection of back end servers, also not focused on the situation considered;

- **High-performance Computer Cluster (HPCC)** - are implemented in order to provide an increased performance by splitting a computational task across many different nodes in the cluster. One of the most popular HPC implementations is the Beowulf cluster [9] with its nodes running Linux as the free OS software to implement the parallelism and with custom programs designed to exploit the parallelism available, this is where the message-passing model in general, and MPI in particular, dominate parallel programming of commodity clusters. These HPC are typically man-


aged by job schedulers, applications for controlling unattended execution of jobs or simply batch scheduling.

2.1.3 Job Managers

Job managers [10] are commonly used in order to administrate available resources in HPCC according to the job requirements. In this way, they evaluate available assets and make a schedule that determines the order in which jobs will be executed, controlling these tasks on available supercomputers (or nodes) within the clusters and organizing submitted jobs based on priority, resources requested, and availability. Additionally, maintain a list of available computer resources and reports the status of previously submitted jobs to the user.

Besides the program to be executed, the job also contains information regarding the number of processors requested, estimated run-time, priority level, parallel or distributed execution, and specific I/O requirements. Once submitted to the Queue and executed, a log file is usually created containing all information regarding its run-time, including stdout (standard output) application information.

The job manager TORQUE, used in this work, is used to submit Batch jobs to the cluster from an appropriate login node to a specified queue. To submit a job via TORQUE, we need to write a simple shell script that wraps your job called a "submission script". This submission script has two parts, the "directives" part, telling the scheduler how to setup the computational resources for the job and the actual "job" part, containing the commands to be executed during this job. The directives that can be indicated in TORQUE are the total number of nodes to use, the maximum time that the job can utilize (or Walltime) and the memory requested for the job (usually the memory of a node times the number of nodes chosen). All submitted jobs and queues can be then monitored and managed by simple TORQUE commands.

Typically, the only interface to access the cluster nodes in HPC is handled by these managers according to its availability and job demands, so although the socket library can be used, the IP Address information can only be controlled program wise, not guaranteeing to match the wanted IP addresses for data exchange and leading to an inefficient application performance.

2.2 Network Simulation

Current development and testing of distributed algorithms requires the creation of two distinct code sets: one to execute on simulators and one to be deployed. Real deployments can be complex and expensive so network simulation can be a good alternative before implementing them.

Available simulation tools allow the definition of computation network nodes, its topology and sim-
ulates all communication message exchanges. Although most of these systems can be executed on computational cluster using MPI, they do not take advantage of the large number of available computational nodes. It is extremely complex and demanding to execute real distributed algorithms in thousands of nodes. The most notorious network simulation softwares available have the following limitations:

- **Cloud Sim** [11]: Provides a generalized and extensible simulation framework that enables modeling and experimentation of emerging Cloud computing infrastructures. Cloud Sim supports data center network topologies making it possible to test performance of their provisioning policies in a repeatable and controllable scenario, free of costs for both the environments considered in this work, TCP socket and MPI topologies. It uses Java as a programming language and does not run real applications limiting its usage versatility. Moreover, it contains a very limited communication model which does not process separately inter-core, inter-processor and inter-server communication [12].

- **Ns-2** [13] and Ns-3 [14]: Allows for TCP simulation, routing and multicast protocols usage over both wireless and wired networks, the most common network technologies and applications. Both versions are built as a library which may be statically or dynamically linked to a C++ main program or a C++ wrapped by Python coding. Real-time network simulation presents an unrealistic latency due to complex application demands, not achieving hundreds of nodes without compromising performance.

- **OMNet++** [15]: Component-based kernel C++ simulation library and framework, capable of building network simulators, extensible and modular. Its library also includes the infrastructure to assemble simulations from different components, such as socket TCP and MPI topology. However, also struggles to support functioning hundreds of nodes due to its library complexity.

- **Peer-to-Peer topology Simulator** [16]: Peer-to-peer systems can achieve very large scale simulation involving millions of nodes, whose interactions can be challenging to mediate. **PeerSim** is the most relevant software in this area, its cycle-based engine minimizes the need for synchronization allowing scalability while avoiding deadlocks. Unfortunately, it demands a lot of computational power do so, all connection information are hard to manage besides its high computer power challenge in a large simulation [16].

The only widely available network emulator suitable for the system developed in this work is Planetlab [17]. Research groups are able to request a PlanetLab lot in which they are able to experiment with new services under real-world conditions and on a planetary-scale. Since it uses nodes scattered over the internet, it allows the execution of algorithms in real scenarios despite being impossible to control the network links between processes.
2.2.1 Parallel Network Simulation

Network Simulation models often exhibit extensive simulation run-times due to considerable computational complexity. To mitigate this, it makes sense to harness the parallel processing power and distribute the workload of a simulation model across multiple processing units in a large scale distributed computing. However, for this environment, any parallel network simulator must be custom designed for the specific particular parallel simulation engine, so all services regarding communication and synchronization must be constructed with the network and process characteristics in mind. For HPCC, these simulation software must be tailored specifically for its structure, which can be time and effort demanding since interconnect and each node complexity may vary.

Parallel Network Simulation programs use discrete event simulation, in which a list of pending events is defined and triggered in order. These events, the network model and its configuration are programed by an input program that does not consider anything beyond the simulation at hand. So if the network simulator is not aware of advanced interconnects, such as the Infiniband usage, must use workarounds, like the protocol IPoIB (*Infiniband-over-I*) [4], in order to use TCP/IP over it.

Thus, the parallel program in effect is totally managed by the network simulation application and not in the hand of any researcher using it, possibly achieving unbalanced task load across the parallel infrastructure. The protocol used is programed using socket functions and its logic is built on top of it.

Returning now to the most notorious network simulation software discussed in section 2.2, its parallel network simulation capabilities are as follows:

- **Cloud Sim**: Supports message-passing applications, allowing for provisioning policies to performance tests in a repeatable and controllable scenario free of costs for MPI topologies. Also allows for precise evaluation of scheduling algorithms in scientific MPI-based applications, including the modeling of a data interconnection network [12].

- **Ns-2 and Ns-3**: Only Ns-3 supports MPI topology, being an upgrade when compared to Ns-2 in terms of performance and mostly in implementation. Currently, dividing a simulation for distributed purposes in ns-3 can only occur across point-to-point links [14].

- **OMNet++**: Several attempts have been made with OMNet++ to harness the computational capacity and interconnect characteristics of HPCC using its modularity properties. It still requires lot of computational power, besides associated complexity when implementing such network characteristics and HPCC topology aspects[18].

- **Peer-to-Peer topology Simulator**: MPI and the Socket TCP topologies are supported but not optimized. The distribution *PeerSim-Kademlia-MPI* [19] is prominent as a distributed simulator for large-scale peer-to-peer networks based on the Kademlia protocol (distributed hash table for decentralized peer-to-peer computer networks) and handling the communication between simulation
machines using MPI, making it the closest option for the demands in our motivation section 1.1. The simulator code is based on PeerSim, referred on the section 2.2. PeerSim-Kademlia-MPI supports simulations of networks comprised of up to 10 million peers. However, all interactions between peers are performed using RPCs (Remote Procedure Calls), each of which is comprised of a request and a subsequent response which can limit its possible applications.

- **SSF project**: Scalable Simulation Framework (SSF), also counts with two illustrious implementations, SSFNET’s commercial Java and SSFNet for C++ (DaSSF) both popular for their parallel network simulation capabilities. These implementations are a package of SSF-based components for modeling and simulation of Internet protocols, including networks at and above the IP packet level of detail. Parallel execution can be made maintaining multiple event queues, and executing them on multiple processors with proper synchronization. Despite allowing for stable and scalable high performance in compact simulation platforms, it is not designed for specific clusters of parallels infrastructures such as HPC. [20]

With all these available possibilities, it is still impossible for network and distributed systems researcher, to execute the real code in these simulation softwares across thousands of nodes or to execute it in a library level simulation without compromising performance.

### 2.3 Parallel Programming and Process Communication in HPC

Parallel computing is accomplished by splitting up large and complex tasks across multiple processors. In order to organize and orchestrate parallel processing, our program must consider automatically decomposing the problem at hand and allowing the processors to communicate with each other when necessary while performing their work. This introduces a new overhead, the synchronization and the communication itself.

Computing parallelism can be roughly classified as Distributed Memory (DM) or Shared Memory (SM) class. In Distributed Memory (DM), each processor has its own memory which are connected through a network that can exchange data, thus, limiting the DM performance and scalability. In Shared Memory (SM), each processor can access all of the memory, resulting in automatic distribution of procedurally iterations over several processors - autoparallelization, explicit distribution of work over the processors by compiler directives or function calls to threading libraries. If this overhead is not accounted for, it can create several issues like bottlenecks in the parallel computer design and load imbalances.

The dominating parallel programming paradigms are OpenMP and MPI, both programmable in Fortran, C and C++. Both of these standards inherently use different models of programming in a complementary manner, since parallel programming can combine distributed memory parallelization on the node interconnect with shared memory parallelization inside each node.
A hybrid MPI/OpenMP implementation model has been considered but not followed through due to its complexity and high level of expertise required. It does depend on having good MPI and OpenMP performances. Besides, performance total gains with a hybrid implementation when compared to a single MPI or OpenMP approach are not guaranteed [21]. In the following subsections we will review both single usages.

### 2.3.1 OpenMP

OpenMP (Open Multi-Processing) [22] is an API that supports multi-platform shared memory multiprocessor architectures and the de facto standard supported by most major compilers (Intel, IBM, gcc, etc). In a shared-memory system, every processor has direct access to the memory of every other processor, meaning it can directly load or store any shared address.

The memory access can be managed, certain pieces of memory may be declared as private to the processor, which provides a simple but powerful model for expressing and managing parallelism in an application. In these type of systems, using a straightforward shared-memory programming model we can achieve software scalability. OpenMP programs are limited in their speed-up by the size of the available shared memory machine, whereas MPI programs can more easily scale to larger clusters.

### 2.3.2 Message Passing Interface - MPI

MPI is a message-passing standard that makes it possible to write portable and parallel applications in distributed-memory systems such as computer clusters. MPI libraries support generic communication protocols, such as TCP/UDP and Ethernet, and high-performance message-passing protocols that can achieve an equal or much higher performance than sockets over any network, appealing as an obvious choice for many HPCC applications. It concerns itself only with the library API semantics, while leaving the implementation details to library developers, thus allowing for portability and general efficiency across a vast range of hardware platforms.

MPI uses objects called communicators and groups. These objects can define the way in which processes may communicate with each other and are required by most MPI functions as an argument. Within a communicator, such as \texttt{MPI\_COMM\_WORLD}, which identifies all processes involved in a computation, every process has its own unique integer identifier assigned by the system as soon as the process initializes, this is the rank of the process and it is used by the programmer to specify the source and destination of messages.

Since the communication essentially works on a point-to-point manner, the MPI library has defined functions corresponding to the most basic pair-wise communication operations such as, send and receive, for blocking and non-blocking communication that are significantly simpler than creating and using
TCP sockets. To address specific nodes in a cluster for MPI computations, the argument \texttt{--hostfile} can be used, followed by a file containing this node’s list hosts.

A wrapper compiler, \textit{mpiCC} for C programs and \textit{mpiC++} for C++ programs, takes care of linking them with the MPI libraries, this helps the programmer to use a standard C/C++ programming language compiler together with the MPI libraries.

On one hand, the most commonly used distribution, the OpenMPI [23], is found in most of TOP-500 supercomputers [2], maintaining parallel processing programs to be executed over advanced computer clusters. On the other hand, the MPICH is a high-quality reference distribution of the latest MPI standard (currently MPI-3), and is the basis for a wide range of MPI derivatives [24]. MPICH is also the distribution used in this work. It aggregates a complete implementation of the MPI-3 standard, which provide an extremely high competitive performance, like latency and bandwidth, and provides a stable platform for research and commercial development while supporting a wide variety of HPCC platforms and environments.

When compared to the socket API, MPI lacks one-sided communication operations that would permit to put and get messages and nonblocking collective communication operations.

### 2.3.3 Socket in HPCC

The socket libraries are commonly used by network programmers and general-purpose computing. For the HPCC environment, socket communication is mostly used for out-of-bounds communication in a thriving MPI application. However, since each node runs several processes, sending information to a specific process IP address is rather impossible once its IP address is shared among all processes in that node. The same arises when using threads about to be launched under the same MPI rank, its distinct communication cannot be made, hence cannot be used.

This occurs since sockets were not designed with HPCC infrastructure in mind, they must be implemented satisfying a required high throughput and low latency. Moreover, in conventional networking, using socket streams can create bottlenecks when it comes to transmitting data, this happens because socket routines do not offer the scalability over cluster that MPI does. However, message-passing standards, like MPI, cannot transmit specific messages directly to a known IP address in the network, which in socket environment is rather trivial.

So, TCP Socket communication limits all of the cluster’s potential, being rather functional than efficient in communicating with specific machines using its IP address that is not minor task in HPCC using the MPI standard.

Sockets typical work in asynchronous communication, contrary to MPI that work synchronously, and can handle an unreliable connection in this way. Such important characteristic must be also contemplated and maintained in our system.
2.4 Program Transformation

Since our systems will perform code transformation, it is necessary to evaluate the possibilities that better apply to it.

2.4.1 Reflection

Reflection [25] is a self analysing approach for a program to introspect its own structure. This analysis is used to change the effective behavior and structure at run-time. A reflection-oriented program component can monitor the execution of code segments and modify them according to a desired purpose. There are several tasks where reflection can be used: remote method invocation, serialization, object dumps and database interfaces.

Even though several programming languages support this mechanism, including Python and Java which provide built-in reflection mechanisms, C and C++ are not part of them. Using Java and Python has been considered due to their reflection capability. Yet, both do not have an official MPI implementations or they are implemented with different degrees of success and compatibility.

Both C and C++ compilers need to know all functions and classes at compile-time, which limits adding new ones at run-time according to task necessity [26]. In some application domains, such as our system, this is absolutely necessary. Furthermore, not only do we need to create new functions in run-time, we also need to manipulate specific IP address and library usage (such as the MPI library).

Other than the language limitations, it is not advised to perform any changes directly in socket.h library since it would destroy socket resilience and possible having outrageous results. Instead we chose to create a new library with same behavior functions that is linked with code transformation.

2.4.2 Code transformation

A source-to-source compiler translates between programming languages that operate at approximately the same level of abstraction, contrary to a traditional compiler that translates from a higher to a lower level programming language.

The appliance using source-to-source transformation allows us to maintain the code structure that we do not want to change. This is done focusing in the redirecting of program’s calls and adding a custom library seamlessly for that purpose. Besides this, pattern matching is also used to keep the program logically sound and avoid compilation programs afterwards.

One solution suitable for this work is Coccinelle [27]. This program matches and transforms C code based scripts on a user made package in a language SmPL (Semantic Patch Language) where
the patch specifies desired items to be changed. The proposed solution seems more suitable for its simplicity and effectiveness.
Chapter 3

SooMPI : Socket over MPI

We present SooMPI system in this chapter. Starting with its objectives in section 3.1 and followed by its overview in section 3.2. Then, we dive into the system’s architecture in section 3.3 and its implementation in section 3.4. In this system’s implementation we explore in detail the code transformation script and its process, system library, its characteristics and its node configuration file. Finally we finish with an overview of this system communication in section 3.5.

3.1 Objectives

In order to answer all problems stated in our motivation, the main objective of this system is: Executing multiple processes in HPC, in which every task is assigned to one IP Address so they can be arranged at the programmer’s will.

To accomplish the objective, we have taken the following steps:

- Creating a library capable of withstanding any C socket connection or connectionless functions, working over MPI, without compromising its behavior;

- Conceive a transformation script that implements this library to handle C/C++ programs, with the ability to append them and assign them each to a specific IP address within a HPC cluster with low implementation cost;

- Test its implementation on several test programs and in a complex application such as Apache Thrift framework;

- Evaluate all implementations’ performance in different computational environments.
3.2 Solution Overview

SooMPI is a system that allows an efficient execution of several programs developed in C using sockets, in HPC clusters using a MPI communication layer exclusively. SooMPI comprises a library containing the essential C socket functions working exclusively over MPI and a translation tool that transforms every socket function in one of this library. This translation tool can append several C socket applications into one output program running over MPI and can bind each original application to one IP address of the network.

The representation of the overall functioning can be seen on Figure 3.1. SooMPI uses the translation tool (soompi_trans.py) to transform input C socket programs, as well as a set of library functions (soompi_lib.o) that are linked on the process. The programmer only needs to develop the processes code (program.c) and at a later stage match each process to different IP addresses (soompi_hosts.conf).

![Figure 3.1: SooMPI Overview](image)

The soompi_trans.py script converts all socket calls to the SooMPI developed calls that are stored in soompi_lib.o. After transformation, the newly created source code should be compiled and linked to the librarysoompi_lib.o. Finally, the node configuration file (soompi_hosts.conf), must be created indicating what IP address should run which application converted.

This system allows for a practical adaptation for any program in the TCP socket environment without having to overthink the implementation on MPI, while allowing for any task or application load distribution within the cluster. In the next section 3.3 we describe its architecture.
3.3 **SooMPI Architecture**

_SooMPI_ answers technical and operational requirements following the architecture illustrated on Figure 3.2. We can see that this system takes as input several programs that use socket communication, identified as _Main 1(_ and _Main 2(_ working over socket UDP or TCP, that do not interact necessarily with each other.

![Figure 3.2: SooMPI Architecture](image)

Both applications are transformed by the Code Transformer, a Python script (_soompi_trans.py_) which aggregates two main tasks: Code Parsing, that transforms inspected code looking from socket functions and convert them to SooMPI functions, and Code Adapting, which maintain the C program structure by converting the input _Mains_ into functions of this program and constructs a new program main function that will call these ones converted. Once converted, the Code Transformer links the SooMPI library (_soompi_lib_) that will be used instead of the socket library.

The transformed program has both input _Mains_ on top of the _SooMPI Runtime Environment_. This Runtime Environment initiates and finalizes the MPI processes and establishes the logic behind executing each _Main_, if the output program is being executed on specific IP addresses. This IP address management within cluster, is done using a node configuration file that must be created outside this transformation and contains the combination of IP address and name of the program in which it will be executed.

The output program will work on HPC environment, more precisely on its Job Manager, which will schedule it after being submitted. Once executed it still performs any exchange of information as originally design in its input _Main_ applications according to the programmed logic, but working over MPI. The original socket library, _sys/socket.h_, is still included in the transformed program, so that any structures, integral values and any out of scope SooMPI library function, can still be used as any other library originally included.
3.4 *SooMPI Implementation*

*SooMPI Implementation* can be divided into three main parts: the Code Transformation described in subsection 3.4.1 where we go through the process of transforming input applications into an output final program, the System Library in subsection 3.4.2, its functions’ implementation in subsection 3.4.3 and ending with the Node Configuration File that allows for the application distribution for a specific IP address in subsection 3.4.5.

3.4.1 *SooMPI Code Transformation*

The transformation process, converts a C/C++ program from the socket API to the MPI API and is performed by the `soompi_trans.py` script. Which transforms all of source files, indicated as argument, and is programmed using Python language. It handles transformations by creating a new output C program source code file with all these changes.

*SooMPI Code Transformation for a Single Program*

An overview of the transformation process steps can be seen on Figure 3.3 and followed by a brief description of each one.

![Figure 3.3: Overview of the Script Transformation](image)
1. The transformation applies a regular expression method in order to identify every socket function to be converted. All socket functions receive the system prefix “soompi:”, so the functions on the added library are used instead, while all arguments stay the same;

2. The SooMPI library is added on the header of the file so all its functions, now implemented on program1 code, can be used;

3. The main function of the output program is created. It provides the system Runtime, that handles the initiation and termination of the MPI interface, including the functions MPI_Init() and MPI_Finalize() respectively;

4. A function with the program1 name is created to reference the input program that is now functioning with the SooMPI library over MPI;

5. The input program1's transformed code is added as a function with its name, including headers, to the transformed program. In this way it can be later invoked and used by the system Runtime.

Once the transformation is completed, it follows general MPI application compilation rules, using the compiler wrapper mpiCC for C programs or mpiC++ for C++ programs.

SooMPI Code Transformation for Several Programs

For several applications transformation, the process is similar. The script can receive as arguments, programs that are to be transformed together, it will then append them into one output source file. This process is further illustrated in Figure 3.4:

Figure 3.4: Overview of the Script Transformation with several Programs
1. The transformation applies a regular expression method in order to identify every socket function to be converted in both input programs. Socket functions will also receive the system prefix “soompi,” so SooMPI functions can be used instead, while the arguments stay the same;

2. The SooMPI library is added like before to the header of the file;

3. The main function of the output program is created and provides the system Runtime. Besides, the MPI interface initiation and termination, will contain the logic behind using each input program according to the node configuration file, further described in subsection 3.4.5.

4. Each program transformed has a function with its name created and is placed under control of an extra management function, soompi_programmatch. This function makes sure that each one runs exclusively when launched on the IP address stated on the file soompi_hosts.conf.

5. The input programs’ header is added in the beginning of the file with logic code aggregation;

6. The input programs’ transformed code is added as a function with its name to the transformed program, after all headers, and can later be invoked and used by the system Runtime.

Using this script for several programs, we can easily combine different arrangements, such as client and server topology. In the Steps 5 and 6, instead of being written in the output transformed file, can be instead inserted in a new file per input program. In this way we can avoid compilation issues with repeated libraries on its headers.

### 3.4.2 SooMPI Library

The SooMPI library is a compilation of all major socket functions and maintains the same logic of each one of them but performed over MPI. All functions use the prefix soompi(· · ·) and have no argument changes when compared to its socket counterparts. A brief description of each one of the functions of this library and how implicitly they work is as follows:

- **soompi_socket():** Opens a new socket, using socket(), to reserve that file descriptor integer and fills all the socket’s information in the fd_port structure.

- **soompi_bind():** Binds an address and a port to a file descriptor updating that descriptor’s information in struct fd_port.

- **soompi_listen():** The correspondent file descriptor in struct fd_port is now a passive socket and signaled the maximum length to which the queue of pending connections may grow.

- **soompi_connect() and soompi_accept():** These primitives are related essentially in their set-up method in which perform a hand-shake method. After exchanging information regarding each other communication ranks and binded ports, this information is stored in each process file descriptor information. Once this connection is established, with further use of soompi_recv and soompi_send the information saved is fetched and the data transfer proceeds. This process can be seen on section 3.4.5.
• **soompiRecv() and soompiSend():** Very straightforward usage of the `MPI_Recv()` and `MPI_Send()` respectively, where the size of information transmitted is returned and the necessary arguments, such as the rank of the processes, are fetched or stored on the referred support structures. Since `MPI_Recv()` receives all messages of the same rank, including messages directed to different ports, the `soompi_recv()` must look for any saved pendent message in the `pendent_message` structure every time it is used. If the message received do not match binded port or the file descriptor used, it is saved on this list.

• **soompi_recvfrom() and soompi_sendto():** Same principle of `soompi_recv` and `soompi_send` but can work in connection-mode or connectionless-mode sockets. These functions take additional parameters allowing the caller to specify the recipient of the data, or to be notified of its sender in a datagram communication.

• **soompi_close():** Closes associated file descriptor and resets related information.

• **soompi_finalize():** Runs exclusively `mpi_final` to terminate MPI execution environment.

• **soompi_poll():** Waits for one set of file descriptors to become ready to perform I/O as the original counterpart. It goes through all pendent message lists to find if any file descriptor in the indicated set has any message to be received, both for setup and normal messages. If there is no pendent message, uses the MPI functions `MPI_Probe` and `MPI_Iprobe`, for blocking and non blocking file descriptors respectively, returning any encountered events related to that set of file descriptors. Since we are using file descriptors virtualization and non-blocking MPI functions, their effectiveness is limited by the program’s full synchronization, otherwise it can lead to unforeseeable results.

• **soompi_fcntl:** performs the operation `F_SETFL` or `F_GETFL`, described as an argument, on an input file descriptor.

### 3.4.3 SooMPI Function Implementation

When exchanging information using any of the referred functions, besides the raw data transmitted, there is also complementary information for this system in the beginning of each message.

This header contains the *destination port* number and it is necessary to solve the absence of explicit port information in MPI connections that are being used. On the TCP case, the *port* number is used, by the receiver, to check if there is any file descriptor opened by the sender using that port. The *sender_rank* is being sent to maintain a robust connection, since MPI can fail to get that information due to synchronism problems despite having mechanisms for it. For the time being SooMPI does not support any broadcast systems.

Besides the *port* to which the message is to be sent and *sender_rank*, the default MPI communicator `MPLCOMM_WORLD` is always used, so all processes involved in the computation are considered. There
are two different headers and MPI information when exchanging data, for setup purposes and once the connection is already established. For that distinction we will look into the *soompi.connect* and *soompi.send* functions:

**SooMPI Data exchange for *soompi.connect()***

Regarding the *soompi.connect()* function, a special setup MPI TAG is used to distinguish between messages from already established connections. Its header indicates that its purpose is to establish a connection it also contains the receiver address so that, the receiver (using *soompi.accept()* can check if it is destined to him. An illustration of the messages’ header and MPI TAG used can be seen on the following Figure 3.5.

![Figure 3.5: SooMPI Connect header and MPI TAG](image)

**SooMPI Data exchange for *soompi.send()***

For the *soompi.send()* function, the connection is already established, so it is only necessary to send, on the message’s header, the general information referred. However, it uses a different MPI TAG when compared with *soompi.connect()*). The illustration for this case can be seen on the following Figure 3.6.

![Figure 3.6: SooMPI Send header and MPI TAG](image)

All MPI functions used are synchronous, but in order to maintain and guarantee the robustness of sockets in asynchronous situations, all messages received are stored in each process, when it is not the correct destination for later usage, not breaking the flow of communication. This can increase the wait time since the program must go through all the pendent messages list in each message reception. However, this does not compromise the communication process since it is logic is handled by the process.
Apart from these functions, this library also includes additional functions and structures that supports it, we expose them on the following subsection 3.4.4.

### 3.4.4 Library Support Structures and Functions

For the *SooMPI* library, we can establish all structures interaction in Figure 3.7. This refers to a single process that is associated to a single rank number and an IP Address.

![Library Support Structures and Functions relation](image)

Figure 3.7: Library Support Structures and Functions relation

To handle descriptor behavior in *SooMPI*, the support *fd_port list* structure is being used. This structure pair an IP address and a binded port to a single file descriptor (used as an index of this list). It also gathers all file descriptor properties during the program run time. All ports and file descriptors are still being reserved for its usage in order to prevent errors when running other applications simultaneously.

All processes have access to a *matrix rank-addresses* which compiles the information provided by the node configuration list regarding each rank and associated IP Address of the cluster. Every process executed under this MPI communicator has access to this information.

Several processes can have multiple file descriptors created even if they are treated as a single rank. Every process will need to redirect messages from one of two lists of pending received messages, one related to setup messages (*setup pendent message list*) and one related to general messages (*pendent message list*). Both of these lists will have all messages and setup messages saved, respectively, and will also contain the port addressed for each one. So, in short, the library comprises the following custom support structures:
• *struct fd_port*: Structure that groups IP address and binded ports of the source and destination, socket type, protocol associated and information of the state of that fictional descriptor.

• *struct pendent_message* and *struct setupmessage*: Saves and enlists all the received application messages and setup messages, respectively, that are not handled by the ms_recv() primitives or other similar ones, also contains the port and address from the original message.

• *matrix rank_addresses*: Compiles the correspondence between rank and IP addresses as defined as library input.

When translating the MPI's rank of the process to the IP address of the private network and *vice versa*, for MPI functions' usage, two new functions are needed for that purpose, we have called them *mpi_define()* and *mpi_undefine()*; and they use the node configuration file in the system to get the rank assigned to the IP address and program used. These functions are used regularly by most of the SooMPI library.

### 3.4.5 SooMPI Node Configuration File

The *soompi_hosts.conf* allows on each MPI process, the assignment of IP addresses. Each line of this file will contain one IP address and one main function name. When starting a new process, the SooMPI library will assign it the corresponding address and run the suitable main functions. An example of this structure and usage can be seen in Figure 3.8.

This file will always be needed to run SooMPI; it will contain a list of private addresses that range from 10.0.0.0 to 10.255.255.255 each one will correspond to a MPI process with a rank number based on the order of appearance in the file. The MPI machine file, that maps all nodes in the cluster for
MPI instances, can still be used and differ from the node configuration file since it corresponds to physical addresses only and will only relate to machine management, while `soompi_hosts.conf` is used for application management ends.

**SooMPI Address Assignment for NOW clusters**

For NOW clusters, the IP Address assignment process can be further controlled. Using the MPI machine file option, with the argument `–hostfile` followed by the file itself, can define the MPI rank number to specific IP addresses. Using this ability, we can use `SooMPI` node configuration list together with file to define specific machines’ rank. A practical illustration of this can be seen on Figure 3.9.

![Figure 3.9: Usage of the `soompi_hosts.conf` together with MPI machine file](image)

### 3.5 **SooMPI Communication Process**

The communication process that results from this implementation is further described in this section. In practical examples, we will analyse the functions `soompi_connect(···)`, `soompi_accept(···)`, `soompi_send(···)`, `soompi_recv(···)`, `soompi_sendto(···)`, `soompi_recvfrom(···)` and `soompi_poll(···)`.

**`soompi_connect(···)` and `soompi_accept(···)`**

The `soompi_connect(···)` performs the following procedures:

- Extracts the IP address from the `sockaddr` structure and goes through the matrix `rank-addresses` so the destination rank is known and can be used on the MPI API;
- Sends a connect message to the destination rank, using the TCPSETUP TAG, which contains the following information: “SETUP” word to identify as a setup message, destination port, destination address, rank of sender;
- Receives confirmation from destination with the the word “CONFIRMSETUP” indicating that the connection established was successful.

The data sent to the other process will identify the sender and allow for the creation of the necessary data structures in order to establish the connection, and forward later messages.

On the other side of the connection establishment, the `soompi_accept(···)`, performs matching procedures:

- Goes through pendent setup message list if there is any and uses the first message found, removing it from the list;
• If there is no previous pending message, receives connect message, checks if it has the “SETUP” word, and uses the remaining information to check if it is the correct destination for that message;
• If the message checks a valid setup message, updates virtual socket information status;
• Sends a confirmation message containing the word “CONFIRMSETUP” indicating that the connection establishment was successful;
• Decrements the available connection number backlog argument defined previously in `soompi.listen()`.

The temporal diagram of these two functions process is illustrated on Figure 3.10.

![Figure 3.10: Connect() and Accept() temporal diagram with SooMPI](image)

`soompi.send(···) and soompi.recv(···)`

After a connection is established, the function `soompi.send(···)` performs the following procedures:

• Gathers connection information - port and IP address, from the `fd_port` structure using the function’s file descriptor;
• Converts the gathered IP address to its MPI rank so it can be used on the MPI API;
• Sends message using a custom MPI TAG “DATATransfer”, to the wanted rank containing: destination port, rank of sender and the data itself.

Its counterpart, `soompi.recv(···)` will respond with the following procedures:

• Goes through the pendent message list looking for any received message in this process (possibly for other port);
• If a message is found, removes it from the list and retrieves the wanted length of it;
• Continues the search in the pendent message list if the wanted length has not been satisfied and if no message has been found waits for a new message;
• Retrieves data received and its total length. If the received message is not for this port, saves it in
the message list with its sender and port information.

The soompi_send and soompi_recv temporal diagram process can be seen illustrated on Figure 3.11.

![Figure 3.11: Send() and Recv() temporal diagram with SooMPI](image)

**soompi_sendto(· · ·) and soompi_recvfrom(· · ·)**

Both the functions soompi_sendto(· · ·) and soompi_recvfrom(· · ·), respectively send and receive mes-
sages through connectionless-mode socket. The process performed by the soompi_sendto(· · ·) function
is as it follows:

• Gathers connection information - port and IP address from the sockaddr structure;
• Converts the gathered IP address to its MPI rank, using the matrix rank-addresses in order to be
used on the MPI API;
• Sends message, using a custom MPI TAG “DATATRANSFER”, to the wanted rank containing:
destination port, rank of sender and the data itself just like any message sent by soompi_send().

Its counterpart,soompi_recvfrom(· · ·) uses the following procedures:

• Proceeds like soompi_recv(· · ·) in which goes through a pendent message list looking for any re-
ceived message in this process (possibly for other port), removing it from the list and if no message
has been found, Waits for a new message;
• Compares the received message information with the information bined in its port;
• Retrieves data received and its total length. If the received message is not in this port, saves it on
the message list with its sender and port information.
soompi_poll(···)

The soompi_poll(···) function’s purpose is waiting for one of a set of file descriptors to become ready to perform I/O, returning any event detected. The procedures performed by this function are the following:

- Goes through the set of file descriptors indicated and, for each one, performs the consecutive procedures;
- Search for any message directed to the current file descriptor, so it can be the setup pendent message list if the file descriptor is waiting for connection messages or general pendent messages;
- If no message is found, it waits for any message for that file descriptor, using the MPI functions MPI_Probe and MPI_Iprobe, for blocking and non blocking file descriptors respectively;
- Once a message is found, indicating that file descriptor is ready to perform I/O, returns the associated event on the arguments of this function. For instance, if there is data to read the POLLIN value, defined in poll.h, is returned.
Chapter 4

SooMPI Evaluation

The purpose of the present chapter is the evaluation of SooMPI in terms of suitability, functionality, correctness and performance. Firstly, in section 4.1, Test Environments, we present both environments used to evaluate this system, a NOW cluster and a HPC cluster. Following with Functionality, in the section 4.2, where we validate the behavior of our system after transformation. In section 4.3, Suitability, we assess our system implementation in other software and proceeds to analyze its implementation in the software framework Apache Thrift. In the section 4.4, Correctness, we compare several tests results in both socket and our system implementation in a NOW cluster. And finally, in section 4.5, Performance, we validate the performance of our system in a HPC cluster, including the introduced overhead in data exchange in HPC.

4.1 Test Environments

The evaluation performed accounted for 2 different computational infrastructures, a NOW and a HPC cluster. These have the following characteristics:

- **NOW** – non dedicated cluster comprising 5 computers with 4 available cores each (20 cores), running Ubuntu, MPICH version 3.2 and GCC compiler version 7.1 and connected by a 1Gb/s Ethernet switch;

- **HPC** – Accelerate HPC cluster [27] comprising 80 nodes with 20 available cores each (1800 cores), running MPICH version 3.1.3 and GCC version 4.7.2, and connected by InfiniBand.

The NOW infrastructure grants access to every node, using SSH (Secure Socket Shell), configured to perform SSH login without password so MPI communication can be done without any issues. On the other hand, the HPC infrastructure access is made by the job manager TORQUE [29], in which we can submit jobs and will be then executed according to our resource allocation its availability.
4.2 Functionality

Assessing now the SooMPI functionality, we consider if the overall function of any program transformed maintain its functions without no drawbacks. The validated functions in SooMPI library were:

- **Socket management** – socket, bind listen, connect, poll, fcntl
- **Stream communication** – connect, send, recv
- **Datagram communication** – sendto, recvfrom

Converted programs using this set of functions are executed with the same outcomes on top of MPI and on top of Socket communication, for both NOW Cluster and HPC Cluster environments. Thus, no difference in behavior has been found and even the return values of each functions were correctly tested. An comparison evaluation between this both environments can be seen in the following section 4.4 with specific tests preformed information. All performed tests were made without out-of-bounds socket communication, however out-of-bounds MPI communication was also successfully tested.

Generally, MPI functions tend to lack robustness in the event of failure which is accounted for within each function in this system. For that reason and considering that any file descriptor created is a visualization of a real one, our solution cannot offer the same robustness of the API socket primitives.

The functions `soompi_send()` and `soompi_receive()` differ from each other substantially in regard of data access. In the sender’s end there is only the need to towards to the destination. The receiver's end must go through all the saved messages list for its IP address looking for the one with the wanted port and file descriptor. We can surmise that the receive part is more process heavy, time consuming and, in case of an unbalanced back and forward communication, the information sender will be on average quicker, which can break any homogeneity in the process.

With the creation of a socket using `soompi_socket()` both a new file descriptor and a new entry on structure `fd_port` are created, thus the file descriptor will have two configurations. When using it outside SooMPI library can lead into unpredictable results despite the transformation script preventing this effect almost completely since all socket functions are converted.

4.3 Suitability

SooMPI system has been developed with the implementation capability to different applications effortlessly, allowing access for a HPC environment execution access at low cost implementation. In order to corroborate this aspect, the software Apache Thrift was chosen due to its versatility and scalable communication qualities.
4.3.1 Apache Thrift

Apache Thrift is a powerful framework that can expand its usage and facilitate programming in a modular matter. It supports cross-language network and performs code generation for RPC, resulting in unambiguous scalable communication among components in cloud and other network environments but not HPC clusters[30]. Moreover, having clean abstractions for data transport and serialization, facilitates the programmer to reason about the objective of the application. This transport layer works exclusively using the sockets API making it a relevant and suitable application to implement our system SooMPI.

Thrift’s compiler generates the base code to be used to build RPC clients and servers in several languages, taking predefined data and services by the user. It offers flexibility regarding transport type and also on communication protocols since it is possible to define a mechanism to map data structures to a wire-format (e.g. binary, JSON), both of these properties are supported in this work’s characteristics. All information in transit is handled how it was configured on its Thrift file, its transport layer is where we want to implement SooMPI, so its processor layer, client-server security and other functionalities that Apache Thrift provides behave accordingly.

Transport in Apache Thrift

The Transport layer provides a simple abstraction for reading and writing in both directions to the network, thus it can be transformed for the MPI domain with the usage of our system’s library. This enables Thrift to decouple the transport from the rest of the system (e.g. serialization/deserialization).

The supported and most distinguished transports classes provided by thrift are:

- **TSocket**: Main class that manage the creation of blocking socket Input/Output (I/O) for transport, both in Client and Server side.
- **TFileTransport**: Transport information to be written into a file.
- **TBufferedTransport**: Stores data to an internal buffer that it doesn’t actually write out until flush is called.
- **TFramedTransport**: Required when using a non-blocking server. It sends data in frames, preceded by length information.
- **TMemoryTransport**: Transport that uses memory for buffer based I/O.
- **TZlibTransport**: Performs compression using Zlib, software used for data compression, together with another transport.

For server side applications, Thrift provides the following options:

- **TSimpleServer**: A single-threaded server using standard blocking I/O, used to create new Transport objects for incoming connections.
- **TThreadPoolServer**: A multi-threaded server also using standard blocking I/O.
Apache Thrift's main options in transport uses fundamentally socket I/O as support, the SooMPI library is suitable in such framework with low cost implementation. Let's evaluate in the next sub-section 4.3.2, the Apache Thrift application with this system.

4.3.2 Apache Thrift with SooMPI Implementation

Since Apache Thrift is a multilayered program, that support several languages RPC clients and servers exportation, the original usage method of SooMPI, that uses the transform script soompi_trans.py is not viable. Instead of transforming the original software using this method, the SooMPI library was manually linked to the Apache Thrift compiler for the C++ language and any socket function used will be converted using the same process as this system's transformation script, adding the soompi prefix to all used socket functions. These changes were made on classes TSocket and TSimpleServer, responsible for all communication interactions and server side interactions respectively.

A new C++ Apache Thrift compiler was added to this software, based on a copy of the initial C++ compiler and by editing the classes TSocket and TServerSocket, in which the MPI interface is also initiated and finalized. Once the language option soompi is chosen when generating RPC source files, it will function exactly as before but transports classes will work exclusively over MPI.

Both functions of the SooMPI library soompi_poll() and soompi_fcntl() were developed in order to answer its usage and file descriptor manipulation in Apache Thrift under the classes changed, TSocket and TServerSocket. The function soompi_fcntl(), although it covers all tasks performed in this software, it does not cover all available options that this function can perform.

With the new Thrift compiler now linked to soompi_lib.o, it still generates source code to be used to build RPC clients and servers in MPI environment taking all data types and service interfaces described on thrift's input file by the user, and maintaining the serialization properties of thrift making use of its functions now in an exclusively MPI interface. The thrift generation process must concur with the new option soompi, like thrift -r -gen soompi <filename.thrift>, in which the filename.thrift is a thrift source file with no special treatment.

Once output generated files are compiled, the node configuration file must be also created, having in mind that the first address entry will be the application server. As default, the rank 0 of all processes will be the server and, since it differs from clients in this RPC topology, they must be both launched in the same MPI communicator group. This can diminish the RPC application versatility since that all clients must be launched together with the server but they must be arranged in the same MPI communicator so they can interact with each other. So, besides the soompi_hosts.conf file necessary to SooMPI and compiling with the MPI wrapper mpiC++, the programer does not need to change any other aspect of the original program. The usage of this implementation is further illustrated on Figure 4.1.
In comparison with the original Apache Thrift software, we have to account for considerable information load of the server node receiving several messages for different ports. With this greater load of messages, the server will scale poorly with more connections and can be node demanding. In a Big O complexity analysis, we have to add as a worst access to the single access list of pending messages $O(N)$, besides any memory handling costs in the process. We also have to add several accesses to the $fd_port$ array that stores the information corresponding to any file descriptor integer, since it is directly accessed by its index we account for the weight access of $O(1)$ for each function.

In single communication cases, there is no efficiency drawback when using large messages apart from the computer specifications [31], but in a Server-Clients scenario is where overall efficiency decays. We can infer that beyond memory access the overall performance of this solution relates with MPI performance on the cluster.

This implementation was tested in the NOW cluster and its results can be seen in the next subsection 4.3.3.

### 4.3.3 Apache Thrift with SooMPI Performance

To validate Apache Thrift with SooMPI Implementation, we used the NOW cluster, comprising 5 computers running 4 processes each (20 cores).

In this NOW cluster were performed 2 tests on both original TCP and MPI environments for C++. For the first test, Test program 1, 1 Megabyte were sent from the clients to the server, an integer at a time.
(corresponding to 2 bytes that is the smallest integer type supported by Thrift) with server confirmation, in which the server will increment the integer and send it back the client. For the second test, Test program 2, 100 Megabytes were sent from the clients to the server, an integer at a time as well but only with server confirmation at the end.

Both Thrift files for these Test Programs with the respective Thrift services were created, both using the \texttt{TBufferedTransport} method described in sub section 4.3.1, and then the C++ clients and servers were generated on an Apache Thrift software, version 0.9.3, both on original and SooMPI forms. Once all files generated and compiled and it was added the node configuration file, \texttt{soompi_hosts.conf}, in which only the first address is the server of these tests.

The times of execution of each Test for a single client, in cluster NOW of Apache Thrift implementation, for both tests, can be seen on the following table 4.1:

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
 & Thrift using Socket & Thrift using MPI \\
\hline
Test Program 1 & 216.454 secs & 153.520 secs \\
Test Program 2 & 65.324 secs & 61.21 secs \\
\hline
\end{tabular}
\caption{Time of execution of each client for both Apache Thrift implementations comparison}
\end{table}

The execution time for Apache Thrift using MPI with SooMPI implementation was generally smaller when compared with the original software over socket. For the first Program Test, in which more messages were exchanged, the time it took to complete was significantly higher on the original software over socket.

Since it is a RPC, it is important to test several clients communicating with one server. We have used the same Test Programs but, instead of running a single client, we have used 19 clients and 1 server in total for both Apache Thrift over socket and Apache Thrift over MPI (making all 20 processes available in the NOW cluster). For the Apache Thrift over MPI case, all clients and server must be launched simultaneously so they can share the same MPI group communicator. For the Apache Thrift over socket, the same amount of clients were launched by SSH commands simultaneously as well but the synchronization in this case is nonexistent, so naturally will have a lesser influence in each other performance. The times of execution of each Test for 19 clients, in NOW cluster of Apache Thrift implementation, for both tests, can be seen on the following table 4.2:

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|}
\hline
 & Thrift using Socket & Thrift using MPI \\
\hline
Test Program 1 & 224.22 secs & 191.59 secs \\
Test Program 2 & 66.36 secs & 76.41 secs \\
\hline
\end{tabular}
\caption{Time of execution of 19 clients for both Apache Thrift implementations comparison}
\end{table}

For multiple clients performance, Apache Thrift over socket had almost the same time of execution
when compared with the single client case. For SooMPI Apache Thrift though, the time of execution were higher in both cases, we can notice both the overhead impact on our implementation and the synchronicity impact of launching all processes at the same time that is not that noticeable in Apache Thrift over socket.

For every test performed, the SooMPI Apache Thrift implementation has showed the same results and behavior when compared to the original Apache Thrift version proving the correctness of this implementation in this software, but further analysis of SooMPI correctness can be seen on section 4.4.

The faster performance in most of Apache Thrift over MPI, suggests the same conclusion as it will be seen in section 4.5, where we got around less 10% latency in SooMPI tests when compared to its socket counterpart. In Apache Thrift Case, besides the prioritization of the usage of available high-speed network if available, the file descriptor function `poll()` are simplified version when compared to the real one. The `soompi_poll()` indicates that the file descriptor is active if there is already pendent messages for that process, which can lead to a faster performance than the original socket implementation. It can also lead for the oppose effect, in which due to clients affluence, the pendent message list is vast and will probably lead to more wait time during its access, this has been seen in our several clients tests that had higher time of execution.

### 4.4 Correctness

Besides maintaining the functionality across functions and applications, SooMPI must also maintain them across environments. For that, in this section we intend to validate this system in terms compatibility with NOW clusters, HPC infrastructures but also with multiple versions of compilers, MPI and network layers.

The access by the user to both infrastructures is generally different as described in section 4.1. For the HPC clusters, the access is made using TORQUE that do not make all IP addresses of the cluster known to the user not allowing for a correctness test since the socket API is not trivially used.

Nevertheless, the NOW cluster was used to validate our solution correctness. The same applications were run in its original Socket form and then converted to MPI form using our system transformation. For this validation, were used 20 processes at total (4 processes per computer). Both were tested with different network architectures, besides Client/Server also a ring and a mesh topology were successfully used by changing both the SooMPI Node configuration file without compromising the programs functionality.

Besides the correctness validation, we compares the communication performance of a original C programs and the ones transformed. For this performance comparison, consider the following 3 programs tested:
Test Program 1 - a file of 50 Megabytes were sent from machine 1 to all the other 4 machines in a TCP connection oriented method;

Test Program 2 - is the same test as Program 1 but executed in a connectionless oriented method.

Test Program 3 - 10 Megabytes of information were sent, a byte at a time with confirmation from the receiver, from computer to computer with those 5 computers in a ring formation. The byte sent was an integer that would be incremented on the receiver side and sent back, confirming that the byte has been received.

In the TCP socket version of Program 3, each process would receive the data from one of its neighbor and forward it to the process with the following address (e.g. from 10.0.1.1 to 10.0.1.2). The execution times of each one of these programs were compiled in the following table:

<table>
<thead>
<tr>
<th>Test Program 1</th>
<th>Using Socket API (secs)</th>
<th>Using SooMPI (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Program 2</td>
<td>13.4</td>
<td>12.26</td>
</tr>
<tr>
<td>Test Program 3</td>
<td>15.2</td>
<td>15.0</td>
</tr>
</tbody>
</table>

As we can see on table 4.3, the SooMPI presented less around 10% latency in our tests, when compared with the socket solution before being transformed. More the information exchanged, the more noticeable will this improvement, a seen in Program 3.

Based on these results, we can infer that SooMPI introduces overall performance improvement over TCP socket solution since MPI will prioritize the usage of available high-speed network, considering that will prioritize the usage of available high-speed network if available (using advanced interconnects such as Myrinet or InfiniBand). This increased performance may also be due to network communication optimization done in the MPICH library.

Besides these 3 tests, we also sucessfully tested file transfer, asynchronous and synchronous communication routines and small messages transfer using all available nodes on the cluster. The programs tested were socket connection oriented, datagram oriented and both at the same time.

4.5 Performance

In this section we analyse the SooMPI performance. On section 4.4 we have validated the SooMPI correctness over the NOW cluster. For the HPC cluster, regulated by a job manager access, we successfully tested the performance only of the transformed program with the maximum number of processes available (1800 processes).
These programs test comparison, between socket and MPI implementations, cannot be replicated on the HPC cluster since its addresses are not available for the users to test socket API implementation and its access is regulated job manager. However, with this system implementation, a small header is added in each message exchanged. We will use the HPC cluster to test its implication on general system performance on the following sub section 4.5.1.

4.5.1 SooMPI Communication Overhead impact on HPC Cluster

In order to maintain all socket characteristics across sender and receiver, the SooMPI communication introduces a simple header in each message. This results in a small overhead introduced by this system, and its impact is further evaluated in this sub section.

To evaluate it, we have developed a program in which a comprised of multiple processes, on nodes organized as a ring, transfer 100 Megabytes across them in our HPC cluster using 1000 processes. This program was constructed both in Socket TCP, in order to be transformed then validated over MPI in this cluster, and in purely in MPI functions, to provide a time execution reference within the same HPC cluster. Both programs were launched by job manager but with the same availability of the cluster to avoid different test conditions.

This impact can be seen on the Figure 4.2 bellow, where we compare a transmission of 100 Mbits using this library and regular MPI communication. This was tested using 50 nodes with 20 processes, each with a byte at a time transmission, an integer, followed by an acknowledgment that is the same integer incremented. The Figure 4.2 shows the total time that the information took to go through the ring for both implementations, pure MPI and SooMPI.

![Image: Figure 4.2: Overhead Test with 100Mbits byte at a time](image-url)
We can conclude that SooMPI does add a small overhead in all communication performed. For 1000 processes, the difference in time was around 10%. This is more noticeable the more number of processes are used for the current job the more time it takes to finish going through the ring when compared with pure MPI communications. We can also deduce that the smaller the messages used the more percentage occupancy of this header will have in the message transmitted, which can overstay its presence in large throughput of information beside a numerous of nodes usage.
Chapter 5

Conclusions

SooMPI proved to be efficient in the deployment of applications developed using C and socket API into a HPC environment with MPI support. The transformation is straightforward to the programmer and even complex applications, such as those developed with Apache Thrift, can take advantage of it. SooMPI also allows for the deployment of various network architectures, such as P2P or Client/Server, into the MPI environment without program changes, thus promoting the existence of a single code base and allowing for several topology options’ modeling.

The SooMPI transformation has also showed that keeps the same behavior and functionalities of transformed programs in HPC, demonstrating the correctness of its implementation.

Up until now researchers in certain areas (such as network of distributed systems) were not able to use large scale HPC infrastructures to run their code, due mainly to limitation to the access to IP layers by job schedulers. Performing a simulation on a network simulation software is way more code and time demanding than using this system. With SooMPI these researchers can now take advantage of these infrastructures effortlessly.

5.1 Future Work

The work on SooMPI has still room to grow. The solution does not offer the hybrid usage of MPI and socket functions at the same time, for the same descriptors, and thus reducing out-of-bounds communication potential, rather it set it up for socket over MPI usage and not the traditional one.

SooMPI system process lacks Out-of-bounds socket communication due to the fully conversion of any socket connection code with no difference in treatment. Thus, unless this out-of-bounds communication is added after transformation, it will not be contemplated. Selective code transformation reveals itself as an is enticing option to be developed in the future.
Multicast communication has not been developed nor support synchronous I/O multiplexing (e.g. function `select()` or `pipe()`) since it needs to handle true file descriptors directly and not this system's virtualization approach.

Continuous work regarding quality of service for this system will also be developed, such as enhancing it using dynamic topology allocation regarding its application and infrastructure.

All things considered, this version of SooMPI conceived will be used as a foundation for the development of more complex network simulators (with definition of network topology and QoS on certain links) allowing the execution of more complex, more realist simulations, more efficiently in HPC infrastructures.
Bibliography


