SooMPI - Socket over MPI

André Costa
Instituto Superior Técnico - Universidade de Lisboa
INESC-ID Lisboa
Email: andre.costa@tecnico.ulisboa.pt

Abstract—In order to mitigate the lack of ability to execute socket based programs directly in HPC clusters, we developed *SooMPI*, that allows C socket programs to work over 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 be executed in HPC cluster. It converts regular socket functions and allows for the assignment of IP addresses to the available MPI processes corresponding to each converted application. *SooMPI* was evaluated in two different computational environments and with multiple applications. 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.

Index Terms—sockets programming, MPI, network simulation, code transformation, execution environments.

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.

2 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 environments 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].

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.

3 Background

Considering all aspects raised in introduction, we need to analyse available options regarding computational infrastructures, its communication and simulation properties and limitations.

3.1 Computational Infrastructures

The computer cluster 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).

3.1.1 NOW - Networks of workstations

Networks of workstations (NOWs) [3] 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.

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.

3.1.2 HPCC - High-performance Computer Clusters

HPCCs utilizes supercomputers and computer clusters to address complex computational requirements, such as applications with significant processing time or data-processing requirements.They use specialized multiprocessors 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 uses 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 [4], so Quality of Service (QoS) demands are not typically a concern. Furthermore, it is supported by advanced interconnects like Infiniband, Myrinet or Quadrics [5]. 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. HPCC are typically managed by job schedulers, applications for controlling unattended execution of jobs or simply batch scheduling.

3.1.3 Job Managers

Job managers [6] are commonly used in order to administrate available resources in HPCC according to the job requirements. In this way, they evaluate available assets and makes a schedule that determines the order in which jobs will be executed, controlling tasks on available supercomputers (or nodes) within the clusters, organizing submitted jobs based on priority, resources requested, and availability. Since the nodes in the cluster are allocated by these managers according to its availability and demand, it is not efficient using specific nodes to run programs nor it is trivial using its specific IP Addresses to exchange data within the cluster.

3.2 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. Most notorious network simulation software available present the following limitations:

- *Cloud Sim* [7]: Besides data center network topologies and message-passing application simulation, does not run real applications limiting its usage versatility. [8].
- *Ns-2* [9] and Ns-3 [10]: Presents an unrealistic latency due to its complexity and does not achieves hundreds of nodes without compromising performance [10].
- *OMNet++* [11]: built for building network simulators, but struggles to support functioning hundreds of nodes due to its library complexity [11].
- *Peer-to-Peer topology Simulator* [12]: Achieves very large scale simulation such as millions of nodes and its cycle-based engine minimizes the need for synchronization to allow for scalability while avoiding deadlocks. However, Socket TCP topologies are not optimized and still demands lot of computational power. *PeerSim*

The only widely available network emulator suitable for the system developed in this work is Planetlab [13]. 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.

3.3 Parallel Network Simulation

In 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 interconnects and each node complexity may vary.

Returning now to the referred network simulation software discussed in section 3.2, its parallel network simulation capabilities are as follows:

- *Cloud Sim*: OpenMP and MPI, allowing for provisioning policies performance tests in a repeatable and controllable scenario free of costs for MPI topologies. Also provides precise evaluation of scheduling algorithms in scientific, MPI-based applications, including the modeling of a data interconnection network [8].
- *Ns-2* and Ns-3: Only Ns-3 supports MPI topology though, being an upgrade face 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 [10].
- *OMNet++*: has some implementations regarding HPCC simulation and typical HPCC interconnects characteristics which requires a lot of computational power [14].
- *Peer-to-Peer topology Simulator*:MPI topologies are supported but not optimized. The distribution *PeerSim-Kademlia-MPI* [15] is prominent as a distributed simulator for large-scale peer-to-peer networks based on the Kademlia protocol and handling the communication between simulation machines using 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, SSFNETs commercial Java and SSFNet for C++ (DaSSF) both popular for their parallel network simulation capabilities. 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. [16]

3.4 Parallel Programming

Parallel computing is accomplished by splitting up large and complex tasks across multiple processors. This introduces a new overhead, the synchronization and the communication itself. The dominating parallel programming paradigms are OpenMP and MPI, both programmable in Fortran, C and C++.

OpenMP (Open Multi-Processing) [17] supports multiplatform shared memory multiprocessor architectures, in which every processor has direct access to the memory of every other processor, meaning it can directly load or store any shared address. This memory access can be managed which provides a simple yet powerful model for expressing and managing parallelism in an application. Thus, 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.

3.4.1 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.

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 *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.

The MPI implementation most commonly used, *Open-MPI* distribution [18], is found in most of TOP-500 supercomputers [2], maintaining parallel processing programs to run 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 [19], and is also the one 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.

3.4.2 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.

3.5 Program Transformation

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

Reflection [20] is a mean for a program to analyze and introspect its own structure, which 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. Despite many programming languages providing built-in reflection mechanisms, C and C++ are not part of them. 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 necessity [21]. Furthermore, not only we need to create new functions in run-time and 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, instead we chose to create a new library with same behavior functions that is linked with 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.

4 *SooMPI* : Socket over MPI

We present *SooMPI* system in this section. Starting with its overview in sub section 4.1. Then, we dive into the system's architecture in sub section 4.2 and its implementation in sub section 4.3. In this system's implementation we explore in detail the code transformation script and its process, system library and logic associated, its node configuration file. Finally we finish with an overview of this system communication in sub section 4.4.

4.1 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 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*).

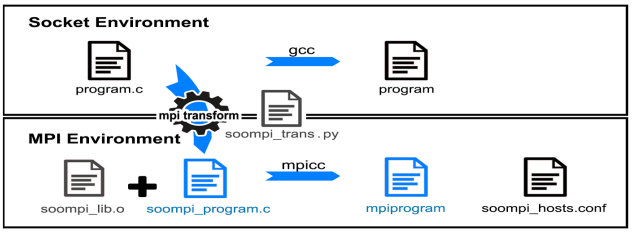

Fig. 1: SooMPI Overview

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 library*soompi_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 4.2 we describe its architecture.

4.2 *SooMPI* Architecture

SooMPI answers technical and operational requirements following the architecture illustrated on figure 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.

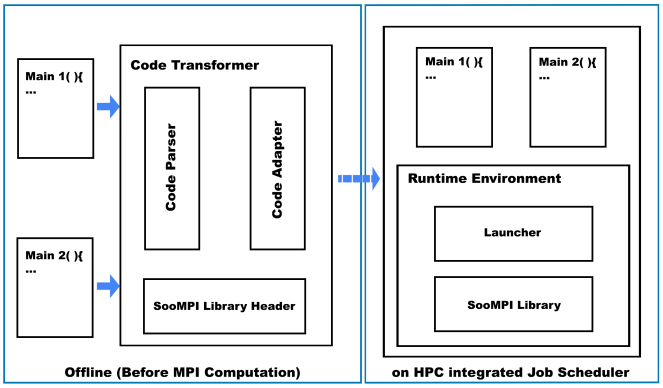

Fig. 2: SooMPI Architecture

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* to 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.

4.3 *SooMPI* Implementation

SooMPI Implementation can be divided into three main parts: the code transformation the *SooMPI*library, its functions implementation and support structures, the node configuration file and finally she *SooMPI*communication process.

4.3.1 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.

The script transforms the input application into a program function with the application's name, then it is later called by the main function. The script also adds the *soompi_lib.h* library in its header together with any header of the input application. The transformation apply regular expressions in order to identify every socket function to be converted. Once identified, all socket functions receive the system prefix "*soompi_*", so the functions on the added library are used instead, but all arguments stay the same.

The main function of the output program provides the system Runt time, it handles the initiation and termination of the MPI interface, including on the program *MPI_Init()* and *MPI_Finalize()* respectively. Contain also an extra management function *soompi_programmatch* for each application converted that makes sure that each one run exclusively when launched on the IP address stated on the file *soompi_hosts.conf*. The whole code transformation, for several programs, process is further illustrated in figure 3:

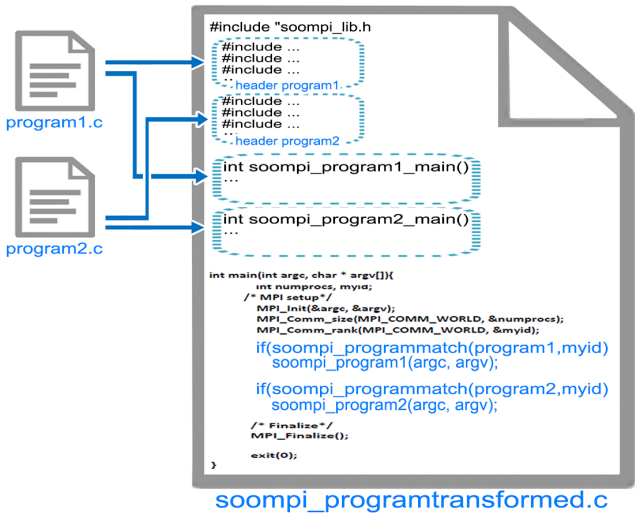

Fig. 3: Overview of the Script Transformation with several Programs

For several applications transformation, indicating which programs are to append as arguments when launching this scrip, the process is similar with special attention to its headers. These input applications are also transformed into functions, all its headers are placed on the beginning of the file and followed by each application code as a function, they are called in the system Run time and managed by the function *soompi_programmatch*. Using this method, we can easily combine different program arrangements, such as clients and server. Once the transformation is completed it follow general MPI application compilation rules, using the compiler wrapper *mpiCC* for C programs or *mpiC++* for C++ programs.

4.3.2 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_connect(···)* 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:

The *SooMPI* library is a compilation of all major socket functions and maintains the same logic function of each one but performed over MPI. All functions use the prefix *soompi_(···)* and have no arguments changes wen compared to its socket counterpart. 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 infotrmation 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 4.3.4.
- *soompi_recv* and *soompi_send:* Very straight forward 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 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_finalize* to terminate MPI execution environment.
- *soompi_poll():* Waits for one set of file descriptors to become ready to perform Input/Output 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.

Besides these functions, this library also includes additional functions and structures that supports it, we expose them on the following subsection 4.3.3.

4.3.3 *Library Support Structures and Functions*

For a single process, we can establish all structures interaction in figure This refers to a single process, that is associated to a rank number and an IP Address. To handle descriptor behavior in *SooMPI* , a support structure is being used, *fd_port list*, that pair an IP address and binded port to a single file descriptor (used as an index of this list). It also gathers all its properties during the program run time. All ports and file descriptors are still being reserved for its usage 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.

Several processes can have multiple file descriptors created, despite being treated as only one rank. This 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, respectively, all messages and setup messages saved, will also contain the port addressed for each one.

4.3.4 SooMPI *Node Configuration File*

The *soompi_hosts.conf* allows the assignment of IP addresses to each MPI process. Each line of this file will contain one IP address and one main function name. When staring a new process, the *SooMPI* library will assigning it the corresponding address and run the suitable main functions. An example of this structure and usage of this file can be seen on figure 4.

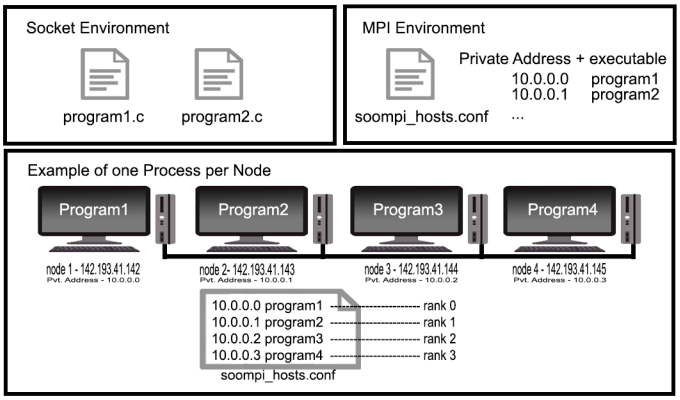

Fig. 4: Structure and usage of the *soompi_hosts.conf*

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.

4.4 *SooMPI* Communication Process

The communication process that results from this implementation is further illustrated in this section. In practical examples, we will analyse the *SooMPI* functions that correspond with *connect(···)*, *accept(···)*, *send(···)* and *recv(···)*.

The *soompi_connect(···)* starts with extracting the address, transforming it into MPI rank, sending a connect message to found process containing: destination port, address, rank of sender, with a custom setup MPI tag, and finally receiving confirmation. The data sent to the other process will identify the sender and allows the creating of the necessary data structures to establish the connection, and forward later messages. On the receiver end, the *soompi_accept(···)* go through the pendent setup message list if there is any, then it receives the connect message or if the wanted message is on the pendent list uses that instead and saves the received one, updates virtual socket information, anf finishes by sending the confirmation message containing also the port, address and rank of the sender and decrement the backlog argument defined previously in *soompi_listen(···)*. The temporal diagram of these two functions process is illustrated on figure 5.

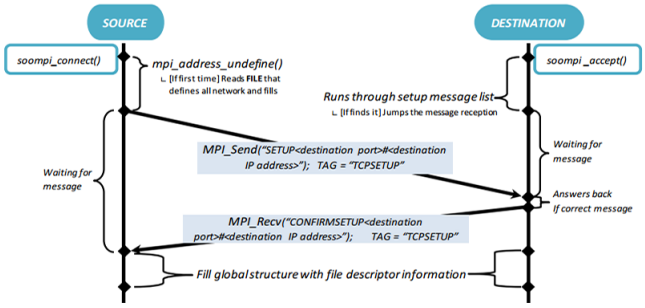

Fig. 5: Connect() and Accept() temporal diagram with SooMPI

After a connection established, the function *soompi_send(···)* gathers the connection information - port and address to send message (address is converted in rank number), then sends the message to wanted process containing: destination port, rank of sender and using a custom MPI tag different from setup tag. Its counterpart,*soompi_recv(···)*, go through the pendent message list looking for any already received in this process (possibly for other port) and, if a message is found, removes it from the list, retrieves it and then waits for the rest of the message if the indicated length is not satisfied. If received message is not for this port, saves it on message list with its sender and port information.

5 *SooMPI* EVALUATION

The purpose of the present chapter is the evaluation of *SooMPI* in terms of suitability, functionality, correctness and performance. Firstly, in sub section 5.1,Test Environments, we present both environments used to evaluate this system. Following with Functionality, in the sub section 5.2, where we validate the behavior of our system after transformation. In sub section 5.3, Suitability, we assess our system implementation in other software and proceeds to analyse its implementation in the software framework Apache Thrift. In the sub section 5.4, Correctness, we compare the results of our system in a NOW cluster versus HPC cluster. And finally, in sub section 5.5, Performance, we validate the performance of our system for several tests, in a HPC cluster, including the introduced overhead in data exchange.

5.1 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 [22] 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 [23], in which we can submit jobs and will be then executed according to our resource allocation its availability.

5.2 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 5.4 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.

5.3 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.

5.3.1 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 [24]. 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.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.

5.3.2 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.

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.

In single communication cases, there is no efficiency drawback when using large messages apart from the computer specifications [25], 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 sub section 5.3.3.

5.3.3 Apache Thrift with SooMPI Performance

To validate Apache Thrift with SooMPI Implementation, we used the NOW cluster, where 2 tests were performed 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 Thift) 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 createdand 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,*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 1:

TABLE 1: Time of execution of each client for both Apache Thrift implementations comparison

	Thrift using Socket	Thrift using MPI
Test Program 1	216.454 secs	153.520 secs
Test Program 2	65.324 secs	61.21 secs

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.

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 5.4.

The faster performance in most of Apache Thrift over MPI, suggests the same conclusion as it will be seen in section 5.5, where we got around 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.

5.4 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 5.1. 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 sucessfully used by changing both the *SooMPI* Node configuration file without compromising the programs functionality.

The tests performed consisted in 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. Further information of the tests performed and execution times are described in section 5.5 in which we present a performance comparison between socket and MPI environment in this NOW cluster.

5.5 Performance

In this section we analyse the *SooMPI* performance, this will be done comparing the execution of test programs between an original application and its transformation using this system. On a NOW cluster, where we compare the performance of the original and transformed programs referred on section 5.4. And, on the HPC cluster, regulated by a job manager access, we tested the performance only of the transformed program with the maximum number of processes available.

5.6 Performance tests on NOW cluster

On the NOW cluster, since both interfaces sockets and MPI were supported and acceded with no limitations, it was possible to compare 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, in which 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. Finally, Test Program 3, has 10 Megabytes of information 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 2: Time of execution of each Test Program for performance analysis

	Using Socket API (secs)	Using SooMPI (secs)
Test Program 1	13,4	12.26
Test Program 2	15,2	15,0
Test Program 3	2130,3	1667,53

As we can see on table 2, 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.

These programs test comparison cannot be replicated on the HPC cluster since its addresses were not available for the users to test the 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 5.6.1.

5.6.1 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 6 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 6 shows the total time that the information took to go through the ring for both implementations, pure MPI and SooMPI.

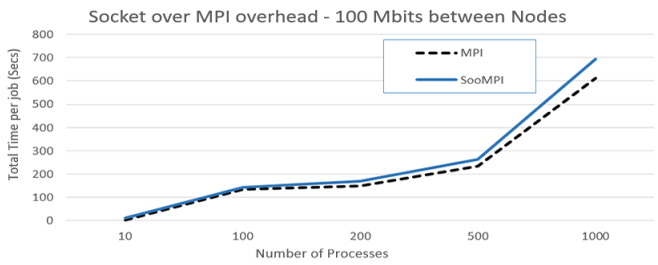

Fig. 6: Overhead Test with 100Mbits byte at a time

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.

6 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.

6.1 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.

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.

References

[1] William Gropp and et al. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[2] Hans MEUER, STROHMAIER, and et al. Top500 supercomputer site, 2016.

[3] Thomas E Anderson and et al. A case for now (networks of workstations). *IEEE micro*, 15(1):54–64, 1995.

[4] Dennis Abts and John Kim. High performance datacenter networks: Architectures, algorithms, and opportunities. *Synthesis Lectures on Computer Architecture*, 6(1):1–115, 2011.

[5] Rubén D Sousa De León. State-of-the-art network interconnects for computer clusters in high performance computing. *Computational Science and Engineering M.Sc. Program*, 2005.

[6] Dror G Feitelson and Larry Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *In IPPS96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. Citeseer, 1996.

[7] Rodrigo Calheiros and et al. Cloudsim: a toolkit for modeling and si-mulation of cloud computing environments and evaluation of resource provi-sioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[8] Harald Richter. About the suitability of clouds in high-performance computing. *arXiv preprint arXiv:1601.01910*, 2016.

[9] Kevin Fall and Kannan Varadhan. The network simulator (ns-2). *URL: http://www. isi. edu/nsnam/ns*, 2007.

[10] Alberto Alvarez, Rafael Orea, and et al. Limitations of network emulation with single-machine and distributed ns-3. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 67. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[11] András Varga, , and et al. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST, 2008.

[12] Niko Kotilainen and et al. P2prealm-peer-to-peer network simulator. In *11th International Workshop on Computer-Aided Modeling, Analysis and Design of Communication Links and Networks*, pages 93–99. IEEE, 2006.

[13] Brent Chun and et al. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Com. Review*, 33(3):3–12, 2003.

[14] András Varga and AY Sekercioglu. Parallel simulation made easy with omnet++. In *Proc. 15th European Simulation Symposium and Exhibition*, 2003.

[15] Philipp Andelfinger and et al. Parallelism potentials in distributed simulations of kademlia-based networks. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014.

[16] Sunghyun Yoon and Young Boo Kim. A design of network simulation environment using ssfnet. In *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pages 73–78. IEEE, 2009.

[17] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[18] Edgar Gabriel and et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

[19] University of British Columbia, Ohio State University, Myricom, Microsoft, Intel Corportation, IBM, and Cray Inc. High-performance portable mpi, 2017.

[20] Giuseppe Attardi and Antonio Cisternino. Reflection support by means of template metaprogramming. In *International Symposium on Generative and Component-Based Software Engineering*, pages 118–127. Springer, 2001.

[21] Duraid Madina and Russell K Standish. A system for reflection in c++. *Proceedings of AUUG2001: Always on and Everywhere*, page 207, 2001.

[22] Intel Corporation. University of lisbon: High-performance plasma physics. Technical report, Intel Corporation, 2014. http://fw.to/SnT46DR.

[23] Theodor Gassmann and John A Barlage. Electronic torque manager (etm®): An adaptive driveline torque management system. Technical report, SAE Technical Paper, 2004.

[24] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[25] Lawrence Livermore National Laboratory. Llcmpi performance topics, 2014.