UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

João Fellipe Uller

A Simple MPI Library for Lightweight Manycore Processors

Florianópolis
2021

João Fellipe Uller

A Simple MPI Library for Lightweight Manycore Processors

Trabalho de Conclusão do Curso do Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Márcio Bastos Castro, Dr.
Coorientador: Pedro Henrique Penna, Me.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Uller, João Fellipe
A Simple MPI Library for Lightweight Manycore Processors / João Fellipe Uller ; orientador, Márcio Bastos Castro, coorientador, Pedro Henrique Penna, 2021.
117 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2021.

Inclui referências.

1. Ciências da Computação. 2. Manycores Leves. 3. Sistemas de Execução. 4. MPI. 5. Computação de Alto Desempenho. I. Castro, Márcio Bastos. II. Penna, Pedro Henrique. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

João Fellipe Uller

A Simple MPI Library for Lightweight Manycore Processors

Este Trabalho de Conclusão do Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 18 de maio de 2021.

Prof. Jean Everson Martina, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Márcio Bastos Castro, Dr.
Orientador
Universidade Federal de Santa Catarina

Pedro Henrique Penna, Me.
Coorientador
Pontifícia Universidade Católica de Minas Gerais

Prof. Frank Augusto Siqueira, Dr.
Avaliador
Universidade Federal de Santa Catarina

Prof. Odorico Machado Mendizabal, Dr.
Avaliador
Universidade Federal de Santa Catarina

This work is dedicated to my colleagues and siblings, who accompanied me througout this course, to my parents, who helped to build the foundation I needed to persevere in life's challenges, to Caroline, that always cheered me up when I needed it most, and finally, to God who made it all possible.

ACKNOWLEDGEMENTS

I would like to thank all those people who helped in any way with the development of this undergraduate dissertation. First, I thank my both advisors, Márcio Bastos Castro and Pedro Henrique Penna, who, from the beginning, had all the attention and commitment to make this the best it could be. I thank for João Vicente Souto and the other colleagues from the research group who also were directly involved in the present work, being an essential part for its execution. In addition, this work was partially supported by Conselho Nacional de Desenvolvimento Científico e Tecnológico – Brasil (CNPq) and by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) under the Capes-PrInt Program (grant number 88881.310783/2018-01), to whom I am grateful for the incentive given that fostered the development of the present work.

Everything should be made simple as possible, but no simpler.
(Albert Einstein)

RESUMO

Nas últimas décadas, melhorar o desempenho de núcleos individuais e aumentar o número de núcleos de alta potência por *chip* foram as principais tendências na construção de processadores. No entanto, esta combinação levou não apenas a um aumento no poder computacional, mas também a um aumento considerável no seu consumo de energia. Há uma preocupação crescente entre a comunidade científica a respeito da eficiência energética dos supercomputadores modernos. Nos últimos anos, muitos esforços têm sido feitos em pesquisas, buscando soluções alternativas capazes de resolver este problema de escalabilidade e eficiência energética. O desempenho e a eficiência energética providos pelos *manycores* leves são inegáveis. Contudo, a falta de suporte avançado e portátil para esses processadores, como interfaces padrão de alto desempenho para o desenvolvimento de código portável, torna o desenvolvimento de *software* um desafio. Atualmente, duas abordagens são empregadas tentando aumentar a programabilidade em *manycores* leves: Sistemas operacionais (SOs) e sistemas de execução (*runtimes*). A primeira fornece portabilidade mas expõe interfaces de programação complexas no nível do SO aos desenvolvedores. Já a segunda se concentra em fornecer interfaces ricas e de alto desempenho, as quais são específicas do fabricante e resultam em *software* não portável. Portanto, as soluções existentes forçam os desenvolvedores a escolher entre a portabilidade do *software* ou um processo de desenvolvimento mais rápido. Para resolver esse dilema, neste trabalho é proposta uma biblioteca MPI leve e portável (LWMPI) projetada do zero para lidar com as restrições e complexidades dos *manycores* leves. A LWMPI foi integrada a um SO direcionado a esses processadores, oferecendo assim uma melhor programabilidade e portabilidade implícita para *manycores* leves, sem incorrer em sobrecargas de desempenho excessivas que inviabilizariam o seu uso. Para fornecer uma avaliação abrangente da LWMPI, foram utilizadas três aplicações de uma suíte de *benchmarking* representativa, usada para avaliar o desempenho de *manycores* leves, além de um benchmark sintético. Os resultados obtidos no processador Kalray MPPA-256 revelaram que a LWMPI atinge uma performance e uma escalabilidade de desempenho melhor do que uma solução feita especificamente para essa análise e que se utiliza puramente das abstrações de IPC do Nanvix, ao mesmo tempo em que oferece uma interface de programação mais rica.

Palavras-chave: Manycores Leves. Sistemas de Execução. MPI. Computação de Alto Desempenho.

ABSTRACT

In the last decades, improving the performance of individual cores and increasing the number of high power cores per chip were the main trends in the construction of processors. However, this combination led not only to an increase in the computing capacity, but also to a considerable growth in energy consumption. There is a crescent concern among the scientific community about the energy efficiency of modern supercomputers. In the last years, many efforts have been made in research, searching for alternative solutions capable of solving this problem of scalability and energy efficiency. The performance and energy efficiency provided by lightweight manycores is undeniable. Although, the lack of rich and portable support for these processors, such as high-performance standard interfaces that deliver portable source codes, makes software development a challenging task. Currently, two approaches are employed trying to improve programmability in lightweight manycores: Operating Systems (OSes) and baremetal runtime systems. The former provides portability but exposes complex OS-level programming interfaces to developers. The latter focuses on providing rich and high performance interfaces, which are vendor-specific and yield to non-portable software. Thus, the existing solutions force software engineers to choose between software portability or a faster development process. To address this dilemma, we propose a portable and lightweight MPI library (LWMPI) designed from scratch to cope with restrictions and intricacies of lightweight manycores. We integrated LWMPI into a distributed OS that targets these processors, thus featuring better programmability and implicit portability for lightweight manycores, without incurring excessive performance overheads that could hinder its use. To deliver a comprehensive evaluation of LWMPI, we relied on three applications from a representative benchmark suite used to assess the performance of lightweight manycores, and a synthetic benchmark. Our results obtained on the Kalray MPPA-256 processor unveiled that LWMPI present better performance and scalability when compared with a specifically made solution that uses the raw Nanvix Inter-Process Communication (IPC) abstractions, while exposing a richer programming interface.

Keywords: Lightweight Manycores. Runtime Systems. MPI. High Performance Computing.

LIST OF FIGURES

Figure 1 – Multiprocessor conceptual view. 30
Figure 2 – Multicomputer conceptual view. 31
Figure 3 – Examples of network topologies. 32
Figure 4 – Overview of the Kalray MPPA-256 lightweight manycore processor. . . 33
Figure 5 – The OpenMP fork-join model. 36
Figure 6 – Conceptual view of the Nanvix microkernel. 42
Figure 7 – Portal and Mailbox conceptual views. 44
Figure 8 – Overview of a distributed OS. 45
Figure 9 – Virtualization conceptual view. 46
Figure 10 – Conceptual view of IKC resource multiplexing. 47
Figure 11 – Architectural overview of LWMPI. 51
Figure 12 – Overview Message Passing Interface (MPI) process management in LWMPI. 60
Figure 13 – Protocol for address lookup and internal structures. 63
Figure 14 – Interactions between LWMPI and Nanvix. 65
Figure 15 – Communication protocol. 67
Figure 16 – Interactions between LWMPI and Nanvix in local communications. . . 68
Figure 17 – Example of *compact* (left) and *scatter* (right) policies. 69
Figure 18 – Execution times obtained with different MPI process mapping policies in a scenario with 12 MPI processes and the optimized version of LWMPI. 75
Figure 19 – FN application results. 76
Figure 20 – GF application results. 77
Figure 21 – KM application results. 78
Figure 22 – Power consumption for K-Means (KM) when varying the number of clusters/problem sizes. 78
Figure 23 – Energy consumption for Friendly Numbers (FN), Gaussian Filter (GF) and KM when varying the exp. scenarios. 79

LIST OF TABLES

Table 1 – Predefined MPI Groups. 55
Table 2 – Predefined MPI Communicators. 56
Table 3 – Predefined MPI Error Handlers. 57
Table 4 – Predefined C Datatypes. 59

Table 5 – Parameters of synthetic and CAP Bench applications. 73

LIST OF ALGORITHMS

1 MPI_Comm_rank entry point. 112
2 mpi_comm_rank underlying function. 112
3 mpi_group_rank implementation. 113
4 Runtime initialization implementation. 114
5 Synchronous send implementation. 116
6 Receive function implementation. 117

LIST OF ABBREVIATIONS AND ACRONYMS

AOS	Asynchronous One-Sided. 35
API	Application Programming Interface. 35, 36, 39, 50, 59, 73, 81
C-NoC	Control NoC. 33
CPU	Central Processing Unit. 29, 30, 31, 80
DMA	Direct Memory Access. 33, 42
D-NoC	Data NoC. 33
DRAM	Dynamic Random Access Memory. 33
DSM	Distributed Shared Memory. 32
FN	Friendly Numbers. 15, 78, 79, 80
GF	Gaussian Filter. 15, 78, 79, 80
GPU	Graphics Processing Unit. 29, 34
HAL	Hardware Abstraction Layer. 34, 41, 45, 46, 47, 48
HPC	High Performance Computing. 29, 35, 37
IKC	Inter-Kernel Communication. 42, 45, 81
IPC	Inter-Process Communication. . . 13, 27, 39, 41, 45, 46, 51, 61, 65, 67, 68, 73, 75, 79, 81, 111, 115
KM	K-Means. 15, 78, 79, 80
MIMD	Multiple Instruction Multiple Data. 25, 29, 32, 34, 52, 61
MISD	Multiple Instruction Single Data. 29
MPB	Message Passing Buffer. 39
MPI	Message Passing Interface. . . 15, 23, 26, 27, 37, 39, 40, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 71, 72, 73, 75, 76, 77, 78, 80, 81, 111, 112, 113, 115
MPSoC	Multiprocessor System-on-Chip. 25
MW	Megawatt. 25
NoC	Network-on-Chip. 25, 26, 32, 33, 46, 47, 61, 67, 69, 72, 73, 76
NUMA	Non-Uniform Memory Access. 30, 31, 34

OpenMP ARB	OpenMP Architecture Review Board. . . . 36
OS	Operating System. . . 13, 26, 27, 34, 35, 40, 41, 43, 44, 45, 46, 49, 50, 51, 53, 61, 62, 69, 81, 111, 113, 115
PE	Processing Element. . . . 25, 33, 53, 59, 60, 61, 62, 69, 72, 73
PGAS	Partitioned Global Address Space. . . . 39
POSIX	Portable Operating System Interface. . . 26, 34, 35, 39, 41, 43, 44, 45, 49, 59, 69, 81
PUC Minas	Pontifícia Universidade Católica de Minas Gerais. . . . 26
RAM	Random Access Memory. . . . 30
RM	Resource Manager. . . . 33
RPC	Remote Procedure Call. . . . 32
SIMD	Single Instruction Multiple Data. . . . 29
SISD	Single Instruction Single Data. . . . 29
SMP	Symmetric Multiprocessing. . . . 34
SPM	Scratchpad Memory. . . . 25
SRAM	Static Random Access Memory. . . . 33
UFSC	Universidade Federal de Santa Catarina. . . . 26
UGA	Université Grenoble Alpes. . . . 26
UMA	Uniform Memory Access. . . . 30
UPC	Unified Parallel C. . . . 39

CONTENTS

1 INTRODUCTION **25**
1.1 RESEARCH GOAL 26
1.2 CONTRIBUTIONS 26
1.3 WORK STRUCTURE 27

2 BACKGROUND **29**
2.1 MULTIPLE PROCESSORS COMPUTER ARCHITECTURES 29
2.1.1 Multiprocessors **30**
2.1.2 Multicomputers **31**
2.2 LIGHTWEIGHT MANYCORE PROCESSORS 32
2.2.1 Software Development Support **34**
2.3 PARALLEL PROGRAMMING ENVIRONMENTS 35
2.3.1 OpenMP **35**
2.3.2 MPI **37**

3 RELATED WORK **39**
3.1 DISCUSSION 40

4 NANVIX OS **41**
4.1 NANVIX MICROKERNEL 41
4.1.1 IPC Abstractions **42**
4.2 NANVIX MULTIKERNEL 44
4.3 ENHANCEMENTS IN NANVIX IKC 45
4.3.1 Virtualization **45**
4.3.2 Resource Multiplexing **47**

5 LWMPI: A MPI LIBRARY FOR LIGHTWEIGHT MANY-CORES **49**
5.1 DESIGN GOALS 49
5.2 OVERVIEW 50
5.3 MPIUTIL 51
5.3.1 Objects **52**
5.3.2 Processes **52**
5.4 LIBMPI 53
5.4.1 Runtime Management **54**
5.4.2 Communication Groups **54**
5.4.3 Communicators **55**
5.4.4 Error Handlers **56**

5.4.5 Datatypes . . . 57
5.5 MPI PROCESS MANAGEMENT . . . 58
5.6 THREAD ADDRESSING SCHEME . . . 61
5.7 POINT-TO-POINT COMMUNICATION . . . 63
5.7.1 Send and Receive Operations . . . 63
5.7.2 Request Cycle . . . 65
5.7.3 Communication Protocol . . . 66
5.7.4 Local Communication Optimization via Shared Memory . . . 67
5.8 PROCESS MAPPING POLICIES . . . 68
5.9 ADDITIONAL CONSIDERATIONS . . . 69

6 EVALUATION METHODOLOGY . . . 71
6.1 APPLICATIONS . . . 71
6.2 EXPERIMENTAL DESIGN . . . 72

7 EXPERIMENTAL RESULTS . . . 75
7.1 IMPACTS OF MPI PROCESS MAPPING POLICIES . . . 75
7.2 PERFORMANCE EVALUATION WITH CAP BENCH APPLICATIONS . . . 76
7.2.1 FN Application . . . 76
7.2.2 GF Application . . . 77
7.2.3 KM Application . . . 77
7.2.4 Energy Efficiency Evaluation . . . 78
7.3 ADDITIONAL CONSIDERATIONS . . . 79

8 CONCLUSIONS AND FUTURE WORK . . . 81

BIBLIOGRAPHY . . . 83

APPENDIX A – SCIENTIFIC ARTICLE . . . 89

APPENDIX B – LIST OF IMPLEMENTED FUNCTIONS . . . 109

APPENDIX C – LWMPI SOURCE CODE EXAMPLES . . . 111
C.1 MPI_COMM_RANK . . . 111
C.2 MPI_INIT . . . 113
C.3 MPI_SEND AND MPI_RECV . . . 115

1 INTRODUCTION

For many years, the advances in semiconductors technology and computer architecture were enough to meet the growing demands for computational power (LARUS; KOZYRAKIS, 2008). In the last decades, the increase in clock frequency of individual cores or, more recently, in the number of high power cores per chip, were the main trends in the construction of processors. However, the combination of these two approaches led not only to an increase in the architectural complexity of these processors, but also to a considerable growth in their power consumption.

There is a crescent concern among the scientific community about the energy efficiency of modern supercomputers. Kogge et al. (2008) emphasizes that the acceptable power consumption of a supercomputer to reach the *Exascale* is around 20 Megawatt (MW). This would lead to a minimum efficiency of 50 GFlops/W, much more than the most power-efficient supercomputer currently in use. According to Green500 (2020), Preferred Networks MN-3 (PFN, 2020) is the number one in this list performing 21.1 GFlops/W.

In the last years, many efforts have been made in order to study, evaluate and develop different solutions trying to solve this problem of scalability and energy efficiency. In this context, lightweight manycore processors surged to address demands on high performance and energy efficiency (FRANCESQUINI et al., 2015). Processors belonging to this class are classified as Multiprocessor System-on-Chips (MPSoCs), as they group all the components of a computer in a single chip. On the one hand, to deliver high performance and scalability, these processors rely on a distributed memory architecture and interconnections based on rich Networks-on-Chip (NoCs). On the other hand, to achieve energy efficiency, they are built with simple low-power Multiple Instruction Multiple Data (MIMD) cores (ROSSI et al., 2017), also known as Processing Elements (PEs), and Scratchpad Memories (SPMs) (MELPIGNANO et al., 2012) with no hardware coherency support. Moreover, these processors may exploit hardware heterogeneity by featuring PEs (or entire clusters) with different capabilities (DAVIDSON et al., 2018). Some industry-successful examples of lightweight manycores are the Kalray MPPA-256 (DINECHIN et al., 2013a), PULP (ROSSI et al., 2017) and the Sunway SW26010 (FU et al., 2016), being the latter employed in the fourth most powerful commercially available supercomputer to date according to TOP500[1] (Sunway TaihuLight).

While the aforementioned architectural features make lightweight manycores more scalable than other parallel processors in both performance and energy efficiency, they introduce several challenges in software programmability. For instance, the *distributed memory architecture* requires a non-trivial software design to handle data access across multiple physical address spaces. Hence, software should explicitly fetch data from remote

[1] https://www.top500.org

memories to local ones to be manipulated (FRANCESQUINI et al., 2015). Furthermore, the *small amount of on-chip memory* demands software to explicitly tile the working data set into chunks and locally manipulate them one at a time (SOUZA et al., 2017). Additionally, it is up to the software to take care of data caching and replication to boost performance. Finally, the rich NoC exposes mechanisms for asynchronous programming to overlap communication with computation (HASCOËT et al., 2017); and hand-operated routing to guarantee uniform communication latencies.

Currently, two approaches are employed to address programmability challenges in lightweight manycores: Operating Systems (KLUGE; GERDES; UNGERER, 2014; ASMUSSEN et al., 2016; PENNA et al., 2019) and baremetal runtime systems (DINECHIN et al., 2013b; VARGHESE et al., 2014; RICHIE; ROSS; INFANTOLINO, 2017). The former is meant to bridge critical programmability gaps imposed by hardware intricacies. The latter aims to expose a rich, performance-oriented programming environment, narrowed to the underlying architecture. While these two approaches are effective for some use cases, they have a significant duality drawback. Application development directly on top of OS interfaces yields to software portability across architectures, but the actual programming interface provided is complex and delays the software development process. In contrast, baremetal and vendor-specific runtime systems expose richer interfaces that accelerate the development process, but they exclusively concern to the software stack ecosystem of a specific lightweight manycore. As an immediate consequence, software written on top of these higher-level interfaces end up to be non-portable. This way, the software stack for lightweight manycores lacks in programmability, once it fails to provide support for both fast development process and software portability.

1.1 RESEARCH GOAL

Based on the aforementioned motivations, the main goal of this undergraduate dissertation is to propose LWMPI: a lightweight and portable MPI library that targets lightweight manycores. We integrated it on top of Nanvix, a Portable Operating System Interface (POSIX)-compliant OS that targets these processors. We believe that combining both of the aforementioned approaches would make possible to our library to provide better programmability and implicit portability for lightweight manycores, without incurring excessive performance overheads that could hinder its use. This work is part of the Nanvix research project, a collaborative project between *Universidade Federal de Santa Catarina (UFSC)*, *Pontifícia Universidade Católica de Minas Gerais (PUC Minas)* and *Université Grenoble Alpes (UGA)*, that aims at the design and implementation of a POSIX-compliant OS for lightweight manycore processors.

1.2 CONTRIBUTIONS

This work delivers the following contributions to the state of the art in portable communication libraries for lightweight manycore processors:

1. We propose new enhancements to the Nanvix low-level communication primitives to better support the proposed library;
2. We propose a simple yet performant lightweight MPI library that leverages the IPC abstractions of Nanvix and includes some of the optimizations of the underlying communication system to reduce the overhead imposed by having an additional software layer;
3. We show that the proposed approach has a very low overhead compared to the low-level communication primitives when running representative benchmarks on Kalray MPPA-256.

Part of the contributions presented in this work have been published in *Escola Regional de Alto Desempenho da Região Sul (ERAD/RS)* (ULLER et al., 2020b) and in *Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD)* (ULLER et al., 2020a). The authors of the paper published in WSCAD have been invited to submit an extended version of the paper to the journal *Concurrency and Computation: Practice and Experience*.

1.3 WORK STRUCTURE

The remainder of this work is organized as follows. In Chapter 2, we cover the theoretical background needed for the discussion of the present work. In Chapter 3, we discuss the related work. In Chapter 4, we present Nanvix, an open-source distributed OS that targets lightweight manycores, which serves as base for the development of the proposed library. In Chapter 5, we present the developed library and discuss its design and implementation details. In Chapter 6, we detail our evaluation method, while in Section 7 we analyze our experimental results. Finally, we draw our conclusions and discuss our future work in Chapter 8.

2 BACKGROUND

In this chapter, we uncover the fundamental concepts related to the present work. In Section 2.1, we discuss concepts that are related to parallel architectures. Next, in Section 2.2, we present the lightweight manycore processors and discuss some of their architectural features, as well as the software development support currently available for them. Finally, in Section 2.3, we present some of the runtime systems that are applicable to lightweight manycores, and that have potential to increase software programmability for this class of processors.

2.1 MULTIPLE PROCESSORS COMPUTER ARCHITECTURES

The ever increasing necessity for computational power always pushed the technological advances of modern computers. Nowadays, there are a countless applications that involve a colossal number of operations to be realized, specially those involved in High Performance Computing (HPC). For this reason, modern architectures focus not only on creating faster computers by adding more powerful components, but also on how to organize these resources to better utilize them. This way, they try to achieve the best performance with better efficiency. To understand the variety of parallel architectures available, Flynn (1972) proposed a taxonomy that classifies parallel architectures in four classes, based in their data and instruction flows.

In the Single Instruction Single Data (SISD) class we have sequential machines that operate over a single execution flow, executing at most one instruction per clock cycle and operating over a single data stream. This way, these machines exploit no parallelism in any of these streams, like single core machines. Next, in the Multiple Instruction Single Data (MISD) class we have machines that execute multiple instructions flows over a single data stream. This is the most uncommon class, and it is difficult to find examples of real systems that belong to this class, but critical systems that need fault tolerance may implement this characteristic. The third class is the Single Instruction Multiple Data (SIMD), that describes computers that are able to apply a single instruction over multiple data streams simultaneously. These computers use hardware replication to achieve this characteristic, and a good example of such machines are the Graphics Processing Units (GPUs). Finally, the MIMD class describes machines that have multiple Central Processing Units (CPUs), where each CPU is capable to execute multiple instructions over multiple data streams, simultaneously, i.e., multiple processes may run on these processors independently and in parallel. Actually, this is the class where most of the modern multiprocessors are included, like the CPUs from Intel and AMD.

In the present work, we are particularly interested in architectures that are capable to support MIMD workloads. There are two main groups that fall into this class, which we present in the next subsections: *multiprocessors* and *multicomputers*.

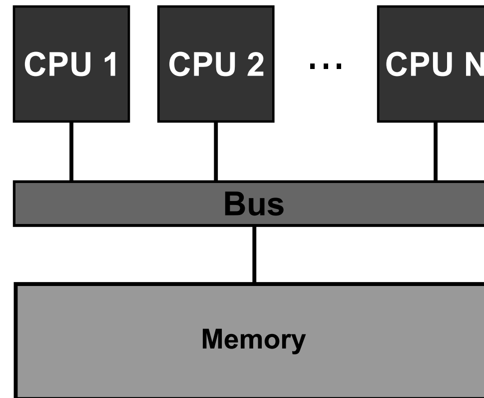


Figure 1 – Multiprocessor conceptual view.
Source: Adapted from Tanenbaum & Bos (2014).

2.1.1 Multiprocessors

Shared-memory multiprocessors are computing systems where two or more CPUs share full access to a common Random Access Memory (RAM) (TANENBAUM; BOS, 2014). Figure 1 illustrates this type of system, in which all CPUs share the same interconnection to access the shared memory module. Additionally to the shared memory, each CPU may have its local cache to reduce access contention to the main memory.

In general, this type of system takes advantage of high rates of parallelism having multiple execution streams. However, since this multiple processors may access the same memory regions, this parallelism may lead to concurrency issues and race conditions, if no synchronization mechanisms are provided. The communication between the different processes is achieved using shared memory regions. For that, the communicant sides agree in a common structure landed in a common memory region, that is read/written when they want to communicate with one another.

According to memory accesses, multiprocessed architectures can be classified into two main classes: Uniform Memory Access (UMA) architectures, where all CPUs have the same latency in memory accesses; and Non-Uniform Memory Access (NUMA) architectures, at which processors may experience different latencies when accessing memory.

In the case of UMA architectures, the memory is centralized and shared across all CPUs, with the characteristic of having the same access latencies for all processors, independently of the interconnection type. The first UMA architectures were bus-based. However, the bus becomes a bottleneck as the number of interconnected processors grows up. This way, crossbar switches and multistage switching networks have become other possibilities for interconnection in these architectures.

Even so, the number of CPUs in UMA multiprocessors is limited to a few dozens. NUMA multiprocessors, on the other hand, offer much more scalability in terms of the number of processors in the same architecture. This type of system stills being characterized by a single address space that is common to all CPUs, but these CPUs may

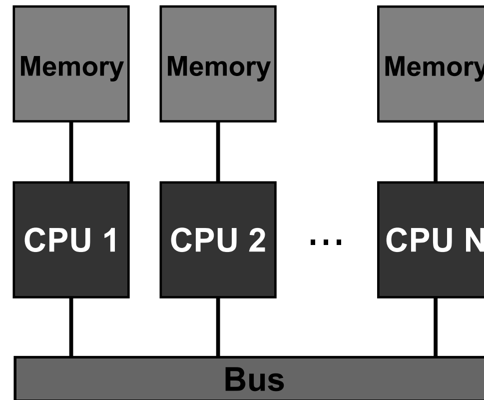


Figure 2 – Multicomputer conceptual view.
Source: Adapted from Tanenbaum & Bos (2014).

have different access latencies, depending on which memory bank is accessed. NUMA architectures are constructed on top of an interconnection that connects multiple NUMA nodes, where each node has its CPUs and close memory bank that is still accessible to CPUs that are external to the node, at the cost of higher access latencies.

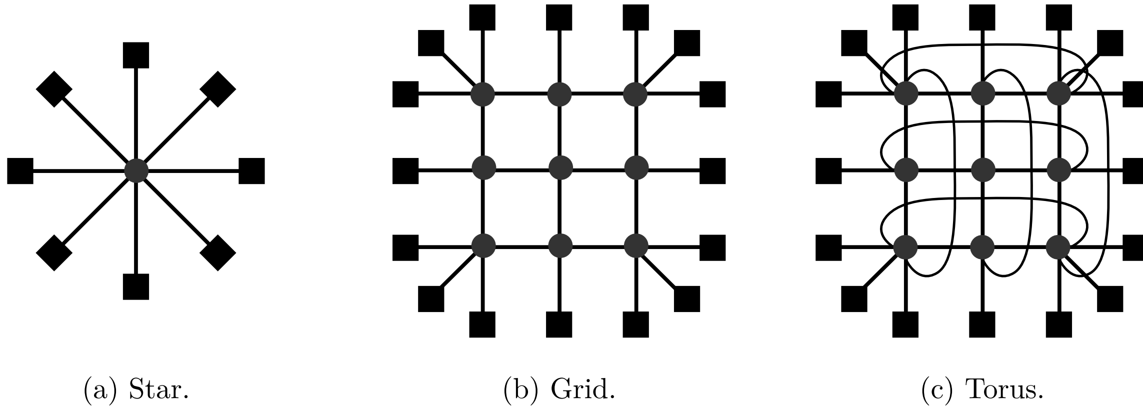
Multiprocessors are popular since they are easier to program. The shared memory makes communication and synchronization much simpler, since any process may achieve this functionalities by simply using shared memory regions. However, building large multiprocessors is difficult and may be very expensive, once the architecture becomes much more complex (TANENBAUM; BOS, 2014). The alternative to keep increasing the number of cores of an architecture is to construct it as a multicomputer.

2.1.2 Multicomputers

Multicomputers are computational systems where the basic components are elementary computers. Each node of this type of system consists in one or more multiprocessors, a local memory module and at least one interface that is used to communicate with the rest of the system. In this type of architecture, a processor has access only to its local memory. To access data that is not locally present, it needs some message passing mechanisms to explicitly communicate with the other nodes. Figure 2 illustrates this type of architecture, in which each CPU has access to its local memory module and a high-performance network interconnects the CPUs.

This approach makes it possible to integrate up to thousands of nodes under the vision of a single clustered architecture. This type of system relieves the burden of providing a single address space and cache coherency across all CPUs in exchange for the problem of providing fast networks and high-speed communication interfaces. However, the goal now is to provide message passing in the microsecond scale, instead of nanosecond in the case of shared memory, being much simpler and cheaper (TANENBAUM; BOS, 2014).

Figure 3 – Examples of network topologies.



(a) Star. (b) Grid. (c) Torus.

Source: Adapted from Tanenbaum & Bos (2014).

To interconnect the nodes of a multicomputer, a switch set of a high-performance network may be organized into a variety of topologies, according to the intended characteristics. For small systems, a single switch in a star topology may be cheaper to implement and also fulfill the system necessities. Commercially used multicomputers, however, generally adopt bi- or even tri-dimensional topologies like mesh, torus or cube that have better scalability and regular behavior to interconnect multiprocessor systems. Figure 3 illustrates some examples of traditional network topologies.

In software, there is the necessity of message passing mechanisms to enable explicit communication between the nodes. There is a variety of options to implement such mechanisms: a *send/receive* library that can be synchronous or asynchronous, depending on the hardware available; Remote Procedure Call (RPC) calls (NELSON, 1981); or even some mechanism that implements Distributed Shared Memory (DSM) notions (NURNBERGER et al., 2014; CHEN et al., 2011), which offers the illusion of shared memory on top of the distributed memory system implemented by the architecture.

2.2 LIGHTWEIGHT MANYCORE PROCESSORS

Lightweight manycores are a new class of manycore processors that have an endeavor to deliver high performance and energy efficiency in a single die. Considering the aforementioned classification, lightweight manycores can be seen as multicomputer systems on a chip. To achieve this, they rely on specific architectural features such as:

i thousands of low-power cores;
ii MIMD capability;
iii tightly-coupled groups of cores (aka *clusters*);
iv distributed memory architecture and small local memories;
v reliable and fast NoCs for message-passing; and
vi heterogeneous processing capabilities in I/O and computing clusters.

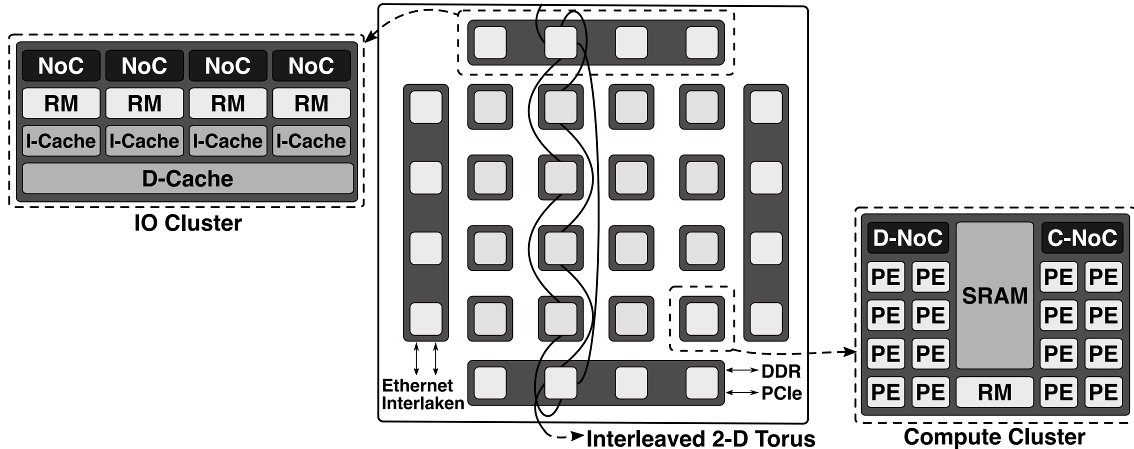


Figure 4 – Overview of the Kalray MPPA-256 lightweight manycore processor.
Source: (PENNA; FRANCIS; SOUTO, 2019)

To provide substantial insight on lightweight manycores, we consider in the present work an industry-successful example of such type of processor: the Kalray MPPA-256 (DINECHIN et al., 2013a). Notwithstanding, the following discussion extends to other lightweight manycores (ROSSI et al., 2017; FU et al., 2016). Figure 4 presents an overview of this processor. Overall, Kalray MPPA-256 integrates 256 general-purpose cores and 32 firmware cores, called PEs and Resource Managers (RMs), respectively, totalizing 288 cores. These cores are disposed into 20 clusters, being 16 Compute Clusters, intended for general-purpose processing, and 4 I/O Clusters, intended for general I/O connectivity, being two of these clusters connected to the Dynamic Random Access Memory (DRAM), and two with PCI/Ethernet controllers. Each cluster is composed of heterogeneous and limited hardware capabilities to perform different roles. For instance, I/O Clusters have four RMs, four NoC interfaces, and 4 MB local Static Random Access Memory (SRAM) to exchange data with external resources and internal clusters. Differently, Compute Clusters have one RM, 16 PEs, one NoC interface, and only 2 MB local SRAM to run user workloads. Cores within the cluster share and have uniform access to hardware resources.

Communication between clusters is exclusively achieved by explicitly exchanging hardware-level messages through two NoCs. Specifically, the Control NoC (C-NoC) enables synchronization and small control messages handover, whereas the Data NoC (D-NoC) supports arbitrary-sized data exchanges. At this point, the I/O heterogeneity among clusters becomes more evident. I/O Clusters have direct access to the attached DRAM or a device, while Compute Clusters must tile their data into messages and send them through the NoC using an I/O Cluster as an intermediary to access these resources. To improve communication performance, Kalray MPPA-256 features a built-in Direct Memory Access (DMA) engine in its NoC interfaces to enable asynchronous communications and higher bandwidth for dense data transfers.

To summarize, the aforementioned set of architectural features grants important distinctions between lightweight manycores and other well-known manycore processors:

- Manycore processors such as Intel Xeon Phi, Tilera TILE-Gx100 and Intel Single-Cloud Computer do not have a constrained memory system, with a distributed architecture and small local memories;
- Symmetric Multiprocessing (SMP) architectures based on NUMA design are built with multiple CPU packages interconnected by a dedicated hardware outside of the processor chips (e.g., NUMAlink); and
- GPUs do not cope efficiently with MIMD workloads.

2.2.1 Software Development Support

The paradigm breakthrough brought by lightweight manycores allows computer systems to scale their performance and energy efficiency. However, challenges introduced by their architectural intricacies to software programmability impact from low- to user-level applications. Examples of these challenges are dark silicon (HAGHBAYAN et al., 2017), data prefetching and tiling (FRANCESQUINI et al., 2015), asynchronous communication (HASCOËT et al., 2017), non-coherent caches (DINECHIN et al., 2013a) and application deployment (SOUZA et al., 2017).

The challenges that arise from the architectural characteristics of lightweight manycores make better development environments an important requirement to porting software for these processors. To do that, there are two approaches currently employed to provide better programmability in lightweight manycores: OSes and baremetal runtime systems.

OSes are meant to bridge critical programmability gaps in architectures. To this end, they provide resource sharing and multiplexing mechanisms, as well as they expose rich abstractions to user-level applications. Inherently due to the architectural features of lightweight manycores, OSes for these processors embrace a distributed design to achieve scalability (BOYD-WICKIZER et al., 2010). In this approach, the OS is factored in a set of services, each of which is deployed on a core of the parallel architecture. Cores that do not run OS services are made available to user-level applications.

Multiple architectures and implementations for a distributed OS are possible, each one targeting a specific set of design goals and constraints. However, a three-tier approach is commonly adopted by distributed OSes for lightweight manycores such as MOSSCA (KLUGE; GERDES; UNGERER, 2014), M^3 (ASMUSSEN et al., 2016) and Nanvix (PENNA et al., 2019). In the bottom layer, a generic and flexible Hardware Abstraction Layer (HAL) enables portability across different processor architectures. A *microkernel* lies in the middle layer and provides minimum system abstractions, handles local resource multiplexing and ensures security policies. Finally, in the top layer, runtime OS libraries expose a standard interface to user-level applications such as POSIX.

In contrast to OSes, baremetal runtime systems aim at exposing a rich programming environment that is narrowed for the underlying architecture. In general, they

implement only essential primitives that manage the hardware to avoid unnecessary overheads to the application or fit a specific programming model design. Usually, they are provided on top of the hardware as libraries and are directly linked with applications. As an immediate consequence, baremetal runtime systems may not hide low-level aspects of the underlying architecture. Moreover, they do not provide important abstractions that are usually implemented by OSes, such as virtual memory, resource sharing, core multiplexing and others. For instance, they do not enable multiple applications to be concurrently deployed in the processor nor provide mechanisms to time-share the hardware between different applications.

Overall, runtime systems are usually shipped by manufacturers of lightweight manycore processors as a cutting-edge performant programming environment. Programming models or well-known standards often guide the Application Programming Interface (API) of the runtimes to benefit a specific set of applications. For instance, NodeOS (DINECHIN et al., 2013b) uses the *pipe-and-filter* programming model to allow processes to communicate on Kalray MPPA-256. The primitives exported by NodeOS resemble the classical POSIX pipes interface, but they require specific knowledge from developers. Differently, libasync (HASCOËT et al., 2017) implements the Asynchronous One-Sided (AOS) communication and synchronization model for Kalray MPPA-256 inspired by libraries used in the HPC domain. The AOS layer defines put/get and atomic operations over requisition queues, allowing applications to read/write data from/to remote memory segments. This model mitigates the problems derived from small local memories in Kalray MPPA-256. However, this approach is focused on enhancing the overall performance of applications, putting aside all programmability and portability issues in lightweight manycores.

2.3 PARALLEL PROGRAMMING ENVIRONMENTS

As most of the modern parallel architectures have been evolving in recent years, better programming environments are needed to help the programmer to explore all the possibilities of parallelism that are offered by these architectures. To ease the software development and to reduce the time needed to develop parallel programs, an increasing effort has been made to make available ports of standard APIs for the most varied parallel architectures.

These programming environments not only make the software development an easier task, but also help the programmer to extract better performance from the underlying hardware. In the next subsections, we describe some of these standard programming environments that are used in the development of HPC applications.

2.3.1 OpenMP

OpenMP is a multi-platform API that intends to provide an easy way to take advantage of parallelism and multiprocessing in shared memory and distributed shared memory architectures. It consists in a set of compiler directives, library routines and environment variables for parallelism in C, C++ and Fortran, that influence the runtime behavior of an application (BOARD, 2020). Its specification is defined and maintained by the OpenMP Architecture Review Board (OpenMP ARB), a consortium composed mostly by hardware and software vendors.

To perform a program parallelization, the OpenMP API implements a *fork-join* parallel model. All OpenMP programs start execution with a single thread (the *master thread*), which sequentially executes the program flow until the definition of a parallel region. When a definition of such type of region is encountered, the *master thread* will fork and create a team, that includes itself and other *worker threads* to execute in parallel the code inside the parallel region. When this team of threads finishes its execution at the end of the parallel region, all the *worker threads* are synchronized and rejoined by the *master thread*, that continues sequentially its execution. Figure 5 illustrates the described process, in which the *master thread* and `N-1` other *worker threads* execute a parallel region. Then, all *worker threads* are joined at the end of the parallel region.

To define a parallel region, OpenMP makes use of a set of compiler directives, where the programmer specifies which regions will be executed in parallel by simply adding these directives in the sequential code. This way, it is easy to parallelize a sequential program in an incremental way (CHAPMAN; JOST; PAS, 2007), inserting compiler directives in small portions of the sequential code, testing if the execution still correct and repeating this process until the desired performance is achieved. Additionally to the compiler directives, OpenMP also defines library routines and environment variables that may be used by the programmer to specify how the parallel execution must behave. This gives a finer control and a better view to the programmer about the parallelism being exploited in the parallel execution.

2.3.2 MPI

The MPI is a portable standard specification for libraries that implement the message passing programming model. It is the *de facto* standard for message passing in HPC, and broadly used in supercomputing and parallel programming. The MPI standard is defined and maintained by the MPI-Forum[1], a group of researchers that includes both people from academia and from industry in its lines. Currently, the most recent official release of the specification is the 3.1 version[2], from June, 2015, with drafts of a possible 4.0

[1] MPI-Forum website: https://www.mpi-forum.org
[2] MPI 3.1 Spec available at: https://www.mpi-forum.org/mpi-31/

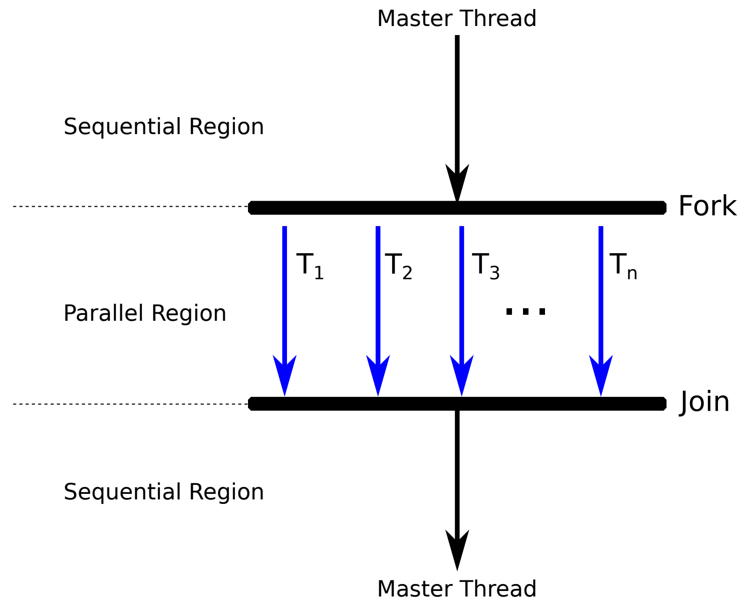


Figure 5 – The OpenMP fork-join model.
Source: Developed by the author.

version being released for discussion in the last few years. Also, there are some open-source implementations of MPI available in public domains, such as MPICH[3] and OpenMPI.[4]

One of the strong points of MPI is its flexibility. It can be applied to various supercomputers and parallel architectures, ranging from shared memory multiprocessors to distributed memory multicomputers (WU et al., 2013). Programs in MPI run as multiple processes, where each process has its own address space and communication occurs through an interconnection network. At the same time, it is important to note that it is flexible enough to be used in shared memory architectures handling the communications as read/write operations in shared memory. Another strong point from MPI is its standardization. A programmer that writes an MPI application may run it in any computer that has the MPI library installed, without making any additional changes to the source code (MUTTIL; LIONG; NESTEROV, 2007). So, MPI is also a way of providing portability across architectures, since there have been existing efficient MPI implementations on a wide range of platforms. Additionally, it presents good scalability in heterogeneous systems, as the processes offer a standardized and homogeneous view of the nodes in the system. This way, communication can be handled efficiently, regardless of the underlying hardware implemented.

The MPI standard includes in its specification functions for point-to-point communication, collective operations that are made on top of groups of processes specified by the user, communication domains to specify different universes of communication, virtual topologies to establish different patterns of communication in collective calls, environmental management, a profiling interface and code bindings for Fortran and C.

[3] MPICH Website: https://www.mpich.org/
[4] OpenMPI Website: https://www.open-mpi.org/

Point-to-point communications are made directly between the involved processes. It can be both synchronous or asynchronous, blocking or non-blocking, and it may use different communication modes, based on the resources exposed by the underlying hardware. These advanced topics, involving the semantics and configuration of point-to-point communication in MPI will be discussed in more depth in Section 5.7, when we discuss the implementation details of point-to-point communication in the proposed library.

3 RELATED WORK

Software development for lightweight manycores is challenging because it strives in finding the balance between performance and programmability. On the one hand, each new layer inserted in the software stack consumes a small part of the hardware resources of the architecture. In lightweight manycores, where some of these resources, such as local memory, are scarce and need to be used wisely, onerous development environments may result in significant performance degradation. In extreme situations, this degradation may even lead to the infeasibility of the solution. On the other hand, programmability is also an important requirement for lightweight manycores, due to all of their intricacies. Otherwise, porting applications to this class of processors may become a painful task.

In this context, and specifically concerning communication, there are two approaches currently employed that try to alleviate this problem:

i vendor-specific communication libraries, which expose a performance-oriented interface for the underlying architecture; and
ii industry-standard communication libraries, which provide a richer communication interface, in exchange for some performance penalty.

Vendor-specific solutions mostly rely on specific features of the underlying hardware to achieve high performance. For instance, synchronous (WIJNGAART; MATTSON; HAAS, 2011) and asynchronous (CLAUSS et al., 2011) interfaces are provided on top of Message Passing Buffer (MPB) for the Intel Single-Cloud Computer. On the other hand, Kalray MPPA-256 features both a communication library that shares some similarity with POSIX (DINECHIN et al., 2013b) and a specific interface for one-sided communications (HASCOËT et al., 2017). A high-level message-oriented parallel programming model is provided for the IMAPCAR (KELLY; GARDNER; KYO, 2013). Finally, a specific communication API is provided for the Adapteva Epiphany processor (VARGHESE et al., 2014).

In contrast, standard communication interfaces benefit from extensive improvements and optimizations, making them a solid choice for programming lightweight manycores. However, to the best of our knowledge, all standard communication interfaces ports are built on top of low-level primitives and libraries provided by the vendors of these processors, making it difficult to adapt them to other manycore processors. Examples of such solutions are those based on the Partitioned Global Address Space (PGAS) programming model, such as the Unified Parallel C (UPC) port for the Intel Single-Cloud Computer (GAMELL et al., 2012) and Tilera TILE64 (SERRES et al., 2011) processors as well as the OpenSHMEM implementation (ROSS; RICHIE, 2016) for the Adapteva Epiphany processor. Moreover, there have been some efforts on providing an MPI port for Kalray MPPA-256 (HO et al., 2015) and Adapteva Epiphany (RICHIE; ROSS; INFANTOLINO, 2017). The former is the closest work to the present one, also presenting

an implementation from scratch to cope with the restrictions of lightweight manycores, having similar concepts to those adopted in the present work. The main difference, however, is the fact that it is implemented on top of a vendor-specific IPC library, and so, being not portable to other processors/architectures. The latter, in addition, does not conform with the MPI standard.

3.1 DISCUSSION

Overall, both of the aforementioned approaches lack application portability. On the one hand, there are very efficient solutions (i.e., vendor-specific libraries) that perfectly adhere to the design purposes of lightweight manycores, but require a greater effort in learning and software design time. On the other hand, there are well-known and widely used standards that alleviate portability problems and improve project development. However, implementations of these interfaces use baremetal facilities, making the entire standard stack architecture-dependent and difficult to be adapted to other lightweight manycores.

For this reason, this work takes a step further on providing a flexible and extendable implementation of a well-known parallel programming standard (MPI) on top of an open-source OS for lightweight manycores (Nanvix). We rely on an OS to provide rich hardware management, sharing and multiplexing and we implement and deploy a high-level, industry-standard library on top of this OS. We believe that the proposed solution brings the best of the aforementioned approaches, since it offers a standard high performance solution with implicit portability, that can be used in a broad range of lightweight manycores or easily adapted to be implemented on top of other runtime systems.

4 NANVIX OS

Nanvix[1] is an open-source research OS that aims at addressing the intricacies of lightweight manycores (PENNA et al., 2019). It is a POSIX-compliant OS designed from scratch to be compatible with this new class of processors, seeking for the balance between performance, portability and programmability. Nanvix adopts a distributed multikernel structure that consists in multiple instances of asymmetric microkernels, i.e., each cluster has a single core dedicated exclusively to executing the kernel, leaving the other cores for general-purpose computing.

In this chapter, we highlight some concepts and details of the Nanvix structure that are important to the present work, specially some related to the Nanvix Microkernel (Section 4.1) and the Multikernel (Section 4.2). Additionally, in Section 4.3 we present some enhancements that were made by us into Nanvix, in the context of the present work, in order to make it possible to implement LWMPI on top of the Nanvix IPC module.

4.1 NANVIX MICROKERNEL

The Nanvix OS implements a microkernel design, where a minimal version of a basic OS kernel provides resource management, sharing and multiplexing, security, minimum OS services, as well as resource abstractions and primitives at a cluster level (PENNA et al., 2019). This way, it provides the basic functionality to offer support for the implementation of more complex OS services in the user-level space, while keeping a small memory footprint for the kernel itself.

Figure 6 shows an overview of the layers structure adopted by the Nanvix Microkernel. In the lower level, we have the baremetal architectures supported by Nanvix, i.e., Kalray MPPA-256 (DINECHIN et al., 2013a), OpTiMSoC (WALLENTOWITZ et al., 2012), x86 and RISC-V emulated in QEMU, and a special platform for development that is a virtualization on top of Unix64. On top of them, a HAL is the responsible for abstracting the underlying hardware and providing an uniform interface for the above kernel layers. This way, this layer enables portability for the entire kernel, making it compatible with all the abstracted architectures. On top of the HAL, we have the Modules Layer that represents the core of the Nanvix Microkernel, in which are implemented the kernel functionalities and capabilities. Currently, four modules are implemented in the Nanvix microkernel:

- **Thread System**: provides a kernel thread abstraction. Kernel threads run in uninterruptible mode and have exclusive access to a core. It features scheduling, multiplexing and management functionalities to cooperative user threads that are implemented on top of these kernel threads.

[1] Publicly Available at: https://github.com/nanvix

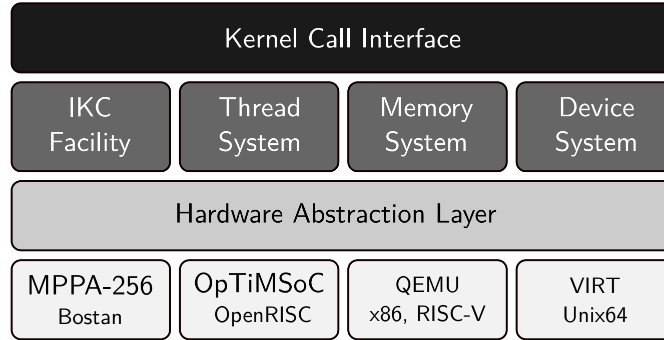


Figure 6 – Conceptual view of the Nanvix microkernel.
Source: (PENNA et al., 2019)

- **Memory System**: provides rich memory management in a cluster, as well as a virtual memory extension that supports different page sizes and permissions tracking.
- **Device System**: controls the access permissions to memory and port mapped devices. Moreover, it exports routines for reading/writing data from/to these devices and for implementing device drivers at user space.
- **Inter-Kernel Communication (IKC)**: provides simple abstractions and primitives to carry out inter-cluster communication. All the abstractions provided operate in a synchronous behavior, but may also include an asynchronous mode in lightweight manycores that feature a DMA engine.

On top of the Modules Layer, the Kernel Call Interface exposes the functionalities implemented by each module to the user space. This layer represents the entry point to the Nanvix kernel from the user perspective, and at this level are performed the verifications and parameters checking that help to provide security in the kernel space. Also, it is in this level where the control flow of the system calls is defined, and where the asymmetric characteristic of the Nanvix kernel is handled: complex routines of the kernel modules are executed exclusively by the master core, while simpler ones can be handled locally by the core that called the specific routine.

4.1.1 IPC Abstractions

To enable inter-cluster communication, the Nanvix IKC exposes three basic message passing abstractions, to explicitly control the data flow across the kernel: *Mailbox*, *Portal* and *Sync*. These abstractions are designed to generalize three common behaviors that are observed in distributed systems: transferring small fixed-size control messages (Mailbox); handling dense data transfers (Portal) and building synchronization points (Sync) (SOUTO et al., 2020). Also, they can be used together, as building blocks, to compose more complex protocols and to construct runtime services. In this section, we give an overview on these abstractions, since they are the base to implementing the communication protocols on the upper layers, including those used by the proposed library.

Mailbox

The Mailbox abstraction (similar to POSIX *Message Queue*), is intended to enable the exchange of small fixed-size control messages with low latency. These messages have their size fixed, generally, in a few hundreds of bytes, and may be handled asynchronously, depending on the abstraction usage.

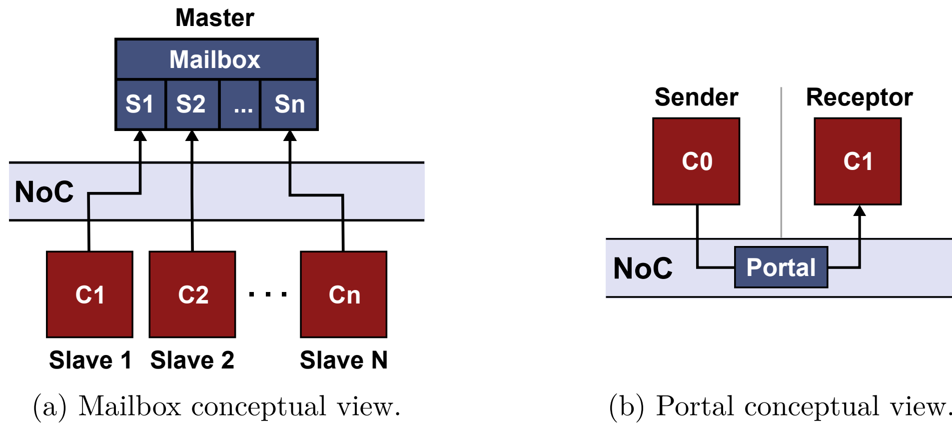
To receive incoming messages, the receiver allocates a message queue with sufficient space to receive exactly one message from each possible sender. This way, each sender have its predefined space to communicate with an input mailbox, as we can see in Figure 7a. When this buffer is already occupied by a previous message, the sender must wait for the first one to be consumed by the receiver before transferring the new message, revealing the synchronous behavior of the abstraction. When the receiver wants to read a message from its incoming mailbox, it may specify the sender from which it wants to receive, or read any available message on the allocated buffer. If no message is available, it blocks waiting for a message to arrive.

In a distributed OS context, this abstraction can be used to implement the agreement phase between client and server of a given OS service, where the client makes a requisition encoding the desired operation and necessary parameters, for example.

Portal

The Portal abstraction (similar to POSIX *Pipe*) provides arbitrarily-sized messages to be transferred with high bandwidth between two nodes. This abstraction is intended for dense data transfers from one cluster to another in an unidirectional channel, as we can see in Figure 7b. When a remote process wants to write data to an input portal, it needs to receive explicit authorization from the receiver. When the receiver wants to read a message from the channel, it specifies from which node it wants to receive data, and then, it grants the permission to write to that node using a special kernel call. If the remote node is not ready to send, the receiver blocks until it starts to receive data through the channel. The sender, in its side, may block in a write call until it receives the authorization from the receiver in the form of an allow signal, that signalizes it can start to send the data. Since this abstraction is totally synchronous, there is no necessity for intermediary buffering. The communication is configured using memory buffers passed as parameters by the user, consuming no additional memory to carry out the dense data transfers.

In the context of a distributed OS, this abstraction may be used as the main building block to carry out the data transfers across the system, since it is designed to handle the dense data transfers with the biggest bandwidth.



(a) Mailbox conceptual view.

(b) Portal conceptual view.

Figure 7 – Portal and Mailbox conceptual views.
Source: (SOUTO; PENNA; CASTRO, 2019)

Sync

The Sync abstraction (similar to POSIX *Signals*) is intended to establish synchronization points among the clusters and to construct distributed barriers. It can be used in one of the two distinct modes available: `ONE_TO_ALL`, in which `N` slaves wait to be unblocked by a single master node; and `ALL_TO_ONE`, where the single leader wait for notifications of the `N` slaves. This way, different types of barriers may be created, giving more flexibility to the programmer of the applications.

4.2 NANVIX MULTIKERNEL

The multikernel OS design was introduced to address the intrinsic characteristics of lightweight manycores (WENTZLAFF; AGARWAL, 2009). The Nanvix Multikernel adopts this type of design, where it is implemented as a distributed system. In this model, each cluster of the distributed architecture executes an instance of the microkernel, and to communicate with one another, these instances use message passing mechanisms. Additionally, traditional OS services are also implemented in this distributed way. In the case of Nanvix, the OS services are modeled in a Client-Server fashion, where the servers are deployed in specific clusters (I/O Clusters), and attend to requisitions that arrive from the user processes, that are deployed in the general-purpose clusters of the architecture (Compute Clusters).

Figure 8 shows a snapshot of a distributed system running in a lightweight manycore in this programming model. The cores of a cluster may be either Idle (black squares), one core of each cluster will be running the kernel instance (dark grey squares), some cores will be executing OS services routines (grey squares), and the remaining may be executing user's space applications (white squares). To be POSIX-compliant, the Nanvix OS services implement standardized interfaces, keeping the client-server model transparent

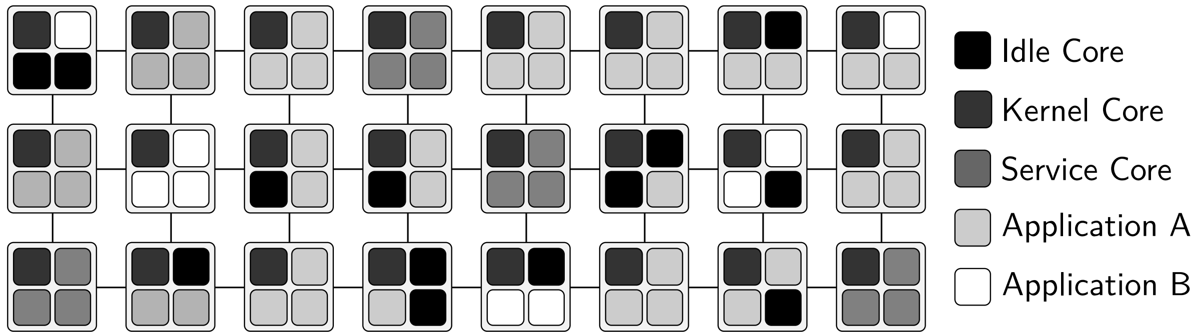


Figure 8 – Overview of a distributed OS.
Source: (PENNA et al., 2019)

to the user. The functionalities are provided in a transparent way, while the underlying implementation of these services perform the computing and communications needed.

At this point, we can notice the versatility given by this model, in which any core of the processor may be used to any of the aforementioned purposes. Also, the set of available services may be adapted for each use-case scenario. At the startup, the user may define where and which kernel instances to deploy across the processor, defining only those services that will be needed, or even defining multiple instances of a single service to enable higher availability, for example. Some of the services currently supported by the Nanvix Multikernel are: (i) a Name Service that provides naming linking and resolution, similar to a DNS service, that translates processes logic names to their logic identifier in the distributed system; (ii) POSIX-compliant Named Semaphores; (iii) POSIX-compliant Shared Memory Regions abstraction; and (iv) a Virtual File System service.

4.3 ENHANCEMENTS IN NANVIX IKC

In this section, we discuss some of the improvements that have been made to the Nanvix IKC subsystem, in order to provide needed features to offer support for LWMPI and that were not yet available in Nanvix. In the next few sections, we discuss the proposed enhancements and explain why they were necessary to LWMPI development, emphasizing their development in the context of the present work.

4.3.1 Virtualization

In computing, the concept of virtualization corresponds to the act of creating a virtual version of a physical computational resource. This virtual resource may be used as a simplified form to generalize a complex component, or to permit a component to be efficiently shared and utilized (LI et al., 2013). In the context of the present work, it is precisely to permit that scarce hardware resources may be managed by the OS, removing this responsibility from the programmer.

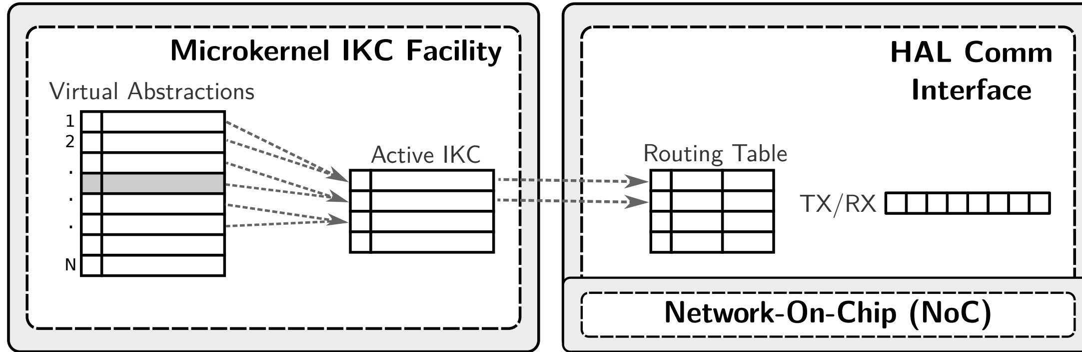


Figure 9 – Virtualization conceptual view.
Source: Developed by the author.

The Nanvix HAL exposes a standard interface of IPC abstractions that enable communication between distinct processes. The number of available abstractions that are exported by the HAL, however, is architecture-dependent, and is limited by the underlying hardware. In the case of lightweight manycores, where most of these hardware resources are scarce, the number of available abstractions is very limited. To cite an example, the Nanvix HAL permits the user in the Kalray MPPA-256 to create a single input portal and a single input mailbox, using the available NoC interfaces in a regular Compute Cluster. This means that there is a single resource abstraction that needs to be shared by the OS and by all user applications that run in a specific cluster.

Because of that, the implementation of mechanisms that enable the virtualization of the IPC abstractions exported by the HAL becomes an important requirement before providing a communication library on top of the Nanvix IKC facility. Not only to make the resource management easier and to permit more peers to keep an active connection simultaneously, but also to permit a finer extraction of statistics related to communication of the kernel. The existence of virtual abstractions, in this case, would permit the programmer and system developers to distinguish communication statistics that are related to kernel communications and those that are respective to the user. Additionally, to LWMPI, it permits the library implementation to establish multiple universes of communication, since different contexts are allowed to use distinct virtual abstractions to carry out the communications, giving much more flexibility for the programmer.

Figure 9 illustrates the virtualization scheme adopted in Nanvix that was implemented as part of the present work. In this implementation, the Nanvix kernel stores two control tables for each IPC abstraction: the first, the *Active IKC Table*, is directly mapped to the hardware resources exposed by the HAL and keeps track of the active connections configured in the underlying interfaces; the second, the *Virtual Abstractions Table*, keeps a mapping scheme of the virtual abstractions exposed by the kernel to their respective active connection. An `N:1` communication pattern is adopted, in which a maximum number of virtual abstractions (`N`) is allowed to be mapped to the same physical abstraction, representing the logic ports controlled by the kernel.

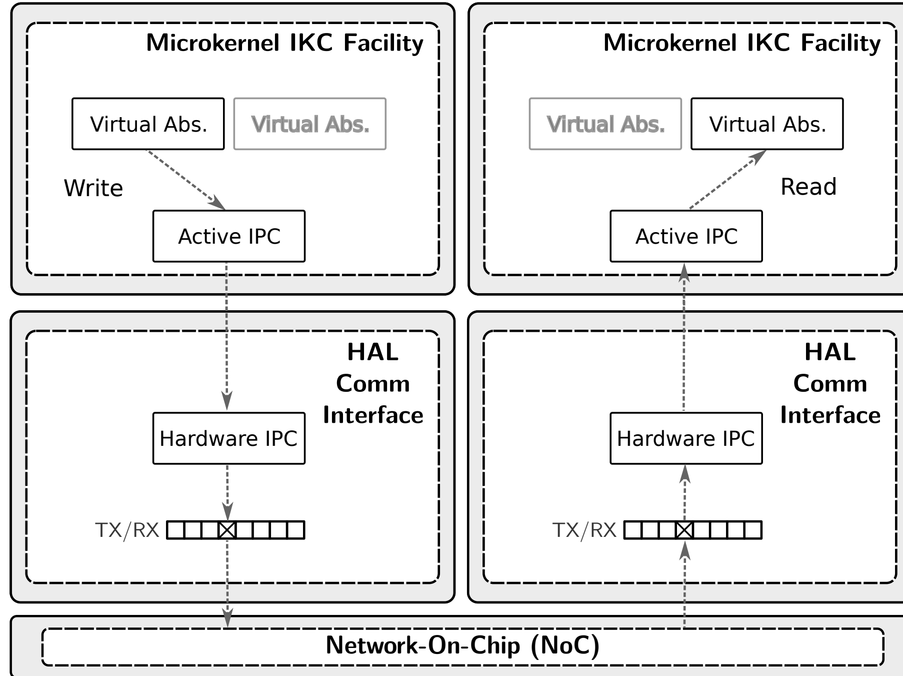


Figure 10 – Conceptual view of IKC resource multiplexing.
Source: Developed by the author.

4.3.2 Resource Multiplexing

The concept of multiplexing consists in combining multiple signals into a single one over a shared medium. In the present work, this concept is adapted and put altogether with the virtualization idea, in which multiple virtual abstractions are multiplexed over a single hardware resource. The importance of this feature is analogous to the virtualization one. In fact, these concepts are tightly linked in the context in which they are employed in this work. Virtualizing hardware resources is not sufficient to provide the intended functionality for the user. It is also necessary to provide means of resource sharing, in which multiple virtual abstractions are capable of using the hardware resources concurrently.

In Nanvix, we implemented a logic port-based scheme in which a limited number of virtual abstractions are mapped to abstractions that represent the hardware resources exposed by the HAL. After that, all of these mapped virtual abstractions may be capable of using the underlying resource, one at a time, while the others wait for it to become free. Figure 10 illustrates the expected data flow when one output abstraction (left) writes a simple message to be read by an input abstraction (right). When an abstraction receives a request from the user space to execute a communication call, it first looks to the lock of the active connection which it is mapped. If the underlying resource is already in use attending another request, it blocks waiting for an opportunity to handle its request. Otherwise, it acquires the lock and proceeds with the request to be processed by the HAL and transmitted through the NoC.

Originally, the communication calls in the Nanvix microkernel did not use any type of intermediary buffering. The input/output buffers used to carry on the communication operations simply came from the user space and were directly transmitted by the HAL routines. However, intermediary buffering schemes must be implemented to allow resource multiplexing, since the kernel needs to have means of storing messages that are read by a virtual abstraction and that are not addressed to it. More than specifying the remote node when writing a message, it is now necessary to address the target port which we want to communicate. This way, an additional *Buffers Table* was added to the kernel to permit this temporary storage of messages. When reading a message, a virtual abstraction may identify that the message that arrived through the interconnection is not addressed to it. This way, it reserves an entry in the *Buffers Table* and stores the arriving message there, while it keeps monitoring the hardware interface waiting for a message containing its logic address. Similarly, when a virtual abstraction wants to read an input message, it first traverses the *Buffers Table* to see if previous messages already arrived for it before trying to obtain access to the underlying hardware interfaces.

5 LWMPI: A MPI LIBRARY FOR LIGHTWEIGHT MANYCORES

In order to improve programmability and portability in lightweight manycores, we propose the **L**ight**w**eight **M**essage **P**assing **I**nterface library (LWMPI). In contrast to alternative solutions (HO et al., 2015; RICHIE; ROSS; INFANTOLINO, 2017; DINECHIN et al., 2013b), more than developing an efficient solution in terms of performance and programmability, we intended to make LWMPI portable across different architectures, and achieved this thanks to a design and implementation that relies on top of a POSIX-compliant distributed OS for lightweight manycores. Before presenting the details and the internals of our solution, we first elucidate the reasons behind some of the design decisions that guided our library implementation and give the reasons why we believe that these decisions are important to achieve our main goal of providing a lightweight MPI-compliant library for lightweight manycores.

5.1 DESIGN GOALS

Lightweight manycores bring several challenges to software development, thereby making easy-to-use interfaces an important requirement for this class of processors. These challenges are not restricted to user-level programming, but also to basic software development. Thus, solutions must meet users demands while dealing with strict architectural constraints, especially memory issues. Hence, the main design goals of LWMPI are:

1. **Portability** The library should be portable and applicable to various lightweight manycores.
2. **Compatibility** The implementation must comply with the MPI specification.
3. **Extendability** It should be possible to add new functions or submodules to the implementation with little effort.
4. **Lightness** The implementation should be simple and lightweight to cope with restrictive resources of lightweight manycores.

To achieve these goals, we rely on important design decisions to cope with the aforementioned challenges:

(i) design our library on top of an OS to enable *portability* across different architectures;
(ii) adhere to the MPI standard to deliver *compatibility*;
(iii) follow a tier-based approach to keep encapsulation and maintain the top-level library isolated from OS-dependent code, thereby enabling *extendability* without incurring excessive overheads that can arise from using more complex software patterns; and
(iv) implement the library from scratch, rather than adapting an existing heavy-weight solution like OpenMPI (SPI, 2020) or MPICH (MPICH, 2020) to keep our solution *light* and suitable for lightweight manycores (HO et al., 2015).

The reason behind the choice of Nanvix to be the base OS for our library is that, to the best of our knowledge, it is the only open-source distributed OS that runs on commercially available baremetal lightweight manycores, like the Kalray MPPA-256 (DINECHIN et al., 2013a) and OpTiMSoC (WALLENTOWITZ et al., 2012). This way, it permits us to develop and test our solution directly on top of a real lightweight manycore, instead of an emulated machine or some other option based in virtualization, as it would have been if we had chosen another candidate OS like M^3 or MOSSCA.

Currently, LWMPI implements an initial subset of the MPI specification and the reason behind this partial support is twofold: first, fully implementing the entire standard would demand much more time to be implemented and tested satisfactorily than the time available for the current work; and second, the complete implementation of the standard would also result in a much larger memory footprint, what violates our fourth design goal (*lightness*).

At this moment, an attentive reader may notice how difficult it is to find a balance between the four proposed design goals, especially considering the *lightness* goal, which may be the most difficult to achieve, while being one of the most important for lightweight manycores. It limits how much we can make the library extensible and how far we can go in the compliance with the full standard. Therefore, we give ourselves a concession in the second goal (*compatibility*) of understanding it in the sense of not diverting our implementation from what is specified by the standard, rather than completely supporting it. This way, we change our commitment from implementing the complete standard to implementing its essential parts, which are applicable for lightweight manycores, as they are specified.

5.2 OVERVIEW

As already mentioned, LWMPI implements an initial subset of the MPI specification (version 3.1), as we can see in Appendix B. Our library is open-source and its source code is available in Github.[1] Figure 11 presents an architectural overview of Nanvix and how LWMPI was introduced in this design. In this figure, we consider a conceptual lightweight manycore composed of one I/O Cluster and two Compute Clusters. Although Nanvix has several OS services and modules, we only present those that are used by our LWMPI implementation. LWMPI has two logical tiers to isolate the MPI API from OS-dependent software: `LibMPI` and `MPIUtil`. In the next sections, we detail the implementation and the functionalities exposed by each one of these tiers. In Section 5.3, we present the concepts and abstractions provided by `MPIUtil`. Next, in Section 5.4, we show the actual interface and the implemented subset of functions exposed by LWMPI. In Section 5.5, we present how LWMPI manages its communicating processes. In Sec-

[1] LWMPI is available at: https://github.com/nanvix/libmpi

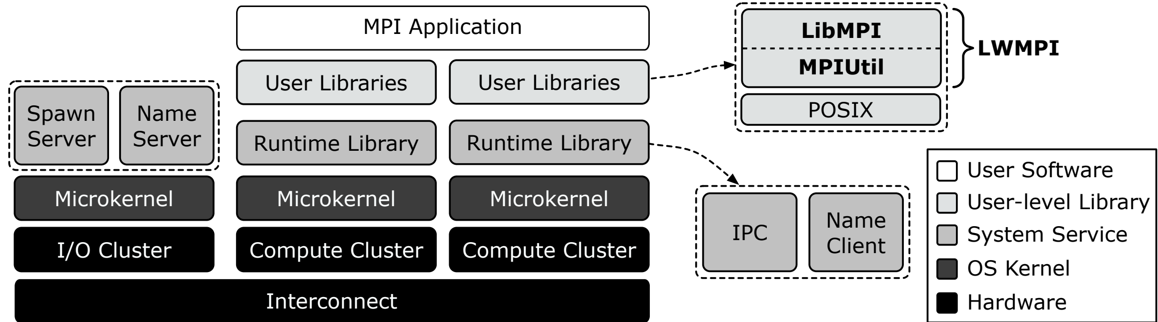


Figure 11 – Architectural overview of LWMPI.
Source: Developed by the author.

tion 5.6, we detail the thread addressing scheme adopted by LWMPI to address multiple MPI processes per cluster. In Section 5.7, we present some internal details about the point-to-point communication implementation in LWMPI, revealing some of its internal structures and the employed protocols. Finally, in Section 5.8, we present the mapping policies available in LWMPI.

5.3 MPIUTIL

The `MPIUtil` tier is the middle layer between the overlying library and the base OS. Precisely, it is responsible for translating the requests from `LibMPI` to the Nanvix interface. `MPIUtil` exposes elementary abstractions that support the top-level implementation of MPI, aiming at keeping the library implementation decoupled from the OS interface. In other words, `MPIUtil` aggregates all OS dependent code, keeping the top tier totally independent of the base OS interface. This enables better extendability to our library, since extensions can be made in the upper layer entirely apart from the OS, if the needed support is already implemented in `MPIUtil`. Additionally, it gives the possibility of easily porting LWMPI to a new OS, or to use another runtime system as base, just adapting this underlying layer to the new candidate interface. Although `MPIUtil` was developed specifically to offer support to `LibMPI`, these layers are designed to be loosely coupled, reinforcing the idea of keeping the top-level library portable and easily extensible.

The `MPIUtil` layer also implements the communication protocols employed by the `LibMPI` calls, e.g., synchronous point-to-point sends and receives. To perform these protocols, `MPIUtil` relies on some important components of Nanvix. Specifically, LWMPI uses the *Spawn Server* to spawn a *system process* on each Compute Cluster. As we show later in Section 5.5, this process may spawn user-level threads within the Compute Cluster. The *Name Service*, composed of a *Name Server* and *Name Clients*, is used by `MPIUtil` to address MPI processes. Finally, it relies on the IPC abstractions to implement the adopted protocols. The IPC abstractions exposed by the Nanvix multikernel, that are the base to compose the communication protocols, are those presented in the Section 4.1.1, i.e.,

mailbox for fine-grain fixed-size transfers, *portal* for coarse-grain fixed-size transfers, and *sync* for building synchronization points and distributed barriers (SOUTO et al., 2020).

In the next sections, we give more details about the main concepts and abstractions implemented in this layer that give support to the `LibMPI` layer. Some of these abstractions are: (i) *Objects* applied in all MPI structures (Section 5.3.1), and (ii) *Processes* for establishing communication groups (Section 5.3.2).

5.3.1 Objects

The first basic abstraction provided by `MPIUtil` is the *Object* abstraction. The standard specification defines that MPI manages the system memory and stores the internal structures representations as opaque pointers. They are so-called because the memory is not directly accessible to the user. However, they may exist in the user space and may be used via specific handles. This way, the `MPIUtil` exposes a generic object abstraction, which defines basic operations such as allocation, deallocation, management and a reference counter to control multiple references, in a common structure that is then implemented by all MPI objects in the above layer.

The idea behind this concept is that one object is separated from another and can be manipulated individually. In our implementation, we use referencing schemes instead of copying these objects. This way, we reduce the amount of instantiated objects and the memory footprint of our library. To efficiently handle deallocations and verifications about a handle validity, each class of complex objects defined in the `LibMPI` tier includes an opaque object that represents a *null* instance. This way, when an object is freed, if it is the last reference for a specific object, it is marked for deallocation and the user opaque pointer references this *null* instance, signalizing that this pointer is no more valid to be used in subsequent MPI operations.

5.3.2 Processes

An MPI application consists in a group of autonomous processes executing in a MIMD style, each one having its own execution flow, that does not need to be identical between distinct processes, and having its own address space (MPI-FORUM, 2020). This way, `MPIUtil` exposes a *Process* abstraction, that represents a simple process from the perspective of the MPI runtime system. Processes in LWMPI are nothing more than an opaque object with a `process name`, a `pid` and the associated thread id (`tid`) attributes, and represent a communicant entity inside the MPI environment.

Using the processes names to represent the entities together with the Nanvix *Name Service*, allows the system to be more flexible. Moreover, this approach allows a process to migrate from one node to another, while being addressable in the MPI environment. At the same time, the usage of opaque pointers and a `pid` facilitates the

internal management of the active processes, since internal structures may use these friendlier representations instead of using the processes names to manipulate them over the system.

In the current version of LWMPI, MPI processes are single threaded from the runtime perspective. However, they may be multithreaded by using solutions like OpenMP or by managing themselves their own threads. However, MPI functions in our current implementation of the LWMPI library are not thread safe. Thus, LWMPI currently expects only one thread per MPI process doing MPI calls to the runtime system. We intend to bring thread safeness to LWMPI in the future. An easier and securer way of using more PEs in LWMPI is to use the scheme of emulated MPI processes that we present in Section 5.5.

5.4 LIBMPI

The `LibMPI` tier is the top-level library and represents the entry point for user applications, encapsulating the MPI standard specification. This layer exposes the library interface and implements the back-end functions on top of the `MPIUtil` tier. This way, its implementation is totally independent from the base OS interface, since all functions are implemented on top of the `MPIUtil` exposed abstractions. At this level, we focus on filtering the input parameters given by the user, performing the runtime management and correctly choosing the protocols employed by each MPI call in the underlying layer. In fact, this layer acts just like a shell that exposes the MPI interface to the user, while it encapsulates and acts as a wrapper for the underlying implementations exposed by the `MPIUtil`.

In this section, we look into the details of the interface exposed by LWMPI, i.e., the currently supported subset of the MPI specification implemented by LWMPI. In the current version, our library implements:

(i) functions for *runtime management*, such as `MPI_Init` and `MPI_Finalize`;
(ii) support for *communicators* and information retrieving, such as `MPI_Comm_rank` and `MPI_Comm_size`;
(iii) support for *groups* of communications with functions that are similar to those related with communicators;
(iv) *error handlers*; and
(v) *point-to-point communication* via `MPI_Send` and `MPI_Recv` using the synchronous mode and carrying any of the predefined *data types* for the C language.

Due to the complexity involved in the implementation of *point-to-point communication* functions (`MPI_Send` and `MPI_Recv`), they will be presented later on in Section 5.7.

5.4.1 Runtime Management

One of the goals of MPI is source code portability, which means that a program written using MPI must not require any source code change when moved from one system to another (MPI-FORUM, 2020). However, an implementation may require an additional setup to be performed before the MPI environment becomes ready. For this, the `MPI_Init` routine is meant to be the responsible for this initialization of the runtime. In the same way, the `MPI_Finalize` is intended to perform all the routines that are necessary for the runtime clean-up, and all processes must call `MPI_Finalize` before finishing. This means that an MPI application must call `MPI_Init` and `MPI_Finalize` once and no other MPI call can be made before `MPI_Init` or after `MPI_Finalize`.

LWMPI implements both `MPI_Init` and `MPI_Finalize`, and these functions are enough to setup and to clean-up the MPI runtime, respectively. Additionally to these functions, LWMPI also implements the functions `MPI_Initialized` and `MPI_Finalized`, which are exceptions to the last mentioned rule and may be called anytime in the execution flow, since they return to the user the current state of the runtime life cycle.

5.4.2 Communication Groups

Communication groups are defined in MPI by a special datatype `MPI_Group`, which defines ordered collections of processes and is the basic structure that defines the scope of the communicators (MPI-FORUM, 2020). The groups define which processes are involved in the context of a communicator, and consequently, the available scope for point-to-point communication and collective operations. The rank of a process inside a communicator is given by its order number inside the group associated to the respective communicator, and the groups may be manipulated separately from communicators. However, only communicators can be used in communication operations.

In LWMPI, groups are implemented as a fixed size array of processes, in which the group size is defined during the group instantiation, as specified by the MPI specification. An example of an important group is the MPI_COMM_WORLD associated group. It is not directly accessible to the user as a predefined group, but it is the one that contains all active MPI processes connected with the MPI_COMM_WORLD communicator, and may be accessed by using the `MPI_Comm_group` function, passing MPI_COMM_WORLD as the communicator input parameter.

Table 1 presents the predefined groups described in the MPI standard. Next, we present the MPI_GROUP_EMPTY and the MPI_GROUP_NULL handles, and what they represent inside the LWMPI.

MPI_GROUP_EMPTY This is a special predefined `MPI_Group` that is characterized by representing an empty group, i.e., with no processes associated to it. It is

Table 1 – Predefined MPI Groups.

Group
MPI_GROUP_EMPTY
MPI_GROUP_NULL

Source: Adapted from MPI-Forum (2020).

important to note that this constant is different from the MPI_GROUP_NULL pointer. The former is a valid pointer for a group of size zero, and consequently may be passed as a valid argument for any function that requires an `MPI_Group` handle. The latter is an invalid handle that is returned when a valid group is freed.

MPI_GROUP_NULL This is not a valid group pointer. It is the *null* handler associated to the `MPI_Group` class, and may be used only to evaluate whether or not a `MPI_Group` object is valid. Passing MPI_GROUP_NULL as a parameter for any function that needs a valid `MPI_Group` will raise an MPI_ERR_GROUP error.

5.4.3 Communicators

Communicators are defined in MPI by a special datatype called `MPI_Comm`. They encapsulate all of the previously mentioned ideas, in order to define the scope of all communication operations in MPI. They aggregate the concepts of group and communication contexts in a single structure. Every process associated to a communicator is able to communicate with the other processes that lay in the same communicator group, which always includes the local process in it. The communication is made using the processes ranks inside the communicator.

Communicators can be of two different types: (i) *intracommunicators*, that are the most common type, which are composed by a single communication group, and distinct contexts for point-to-point and collective communications; and (ii) *intercommunicators*, that are composed by a single communication context and two non-overlapping groups, in which the communications are made from one group to another, i.e., a *send* in the local group is always a *receive* in the remote group. In the current version of LWMPI, only intracommunicators are implemented, since they are the most commonly used communicators in most MPI applications. However, as there is no additional complexity in the implementation of intercommunicators other than preparing the environment to support this functionality, we intend to include it in LWMPI in the future.

Besides the communicators themselves, there are also some functions already implemented in LWMPI that manage and retrieve information from them. For instance, LWMPI provides `MPI_Comm_rank` to extract the local process rank in the given communicator, `MPI_Comm_size` to retrieve the number of processes involved in the given communicator and `MPI_Comm_group` to retrieve the `MPI_Group` associated with the given communicator. All these functions, and some more, are listed and detailed in the

Appendix B. Table 2 also presents the predefined communicators described in the MPI standard and implemented in LWMPI, which we describe next.

Table 2 – Predefined MPI Communicators.

Communicator
MPI_COMM_WORLD
MPI_COMM_SELF
MPI_COMM_NULL

Source: Adapted from MPI-Forum (2020).

MPI_COMM_WORLD This special communicator contains all the connected processes, in the case of a static-processes model, which is the case for LWMPI. This means that the processes that are initialized in the startup of the runtime system are all the available processes during the life cycle of the runtime, with no processes dynamically joining the execution. This communicator cannot be deallocated by the user, and it may be the only communicator used by an application during its lifetime, once it is capable of providing communication between all the available processes.

MPI_COMM_SELF This special communicator contains only the local process itself. It is a valid communicator, liable to be passed as an argument in functions that require a valid `MPI_Comm`. Its importance, however, resides in the fact that it is the default communicator in which errors are raised when an error occurs without having a communicator, window or file associated. The only exception is when an error occurs before `MPI_Init` or after `MPI_Finalize`, in which cases the default error handler is raised with no object associated.

MPI_COMM_NULL This is not a valid MPI_Communicator. It is the *null* handler associated to the `MPI_Comm` class, and may be used only to evaluate whether or not a `MPI_Comm` object is valid. Passing MPI_COMM_NULL as a parameter for any function that needs a valid `MPI_Comm` will raise an MPI_ERR_COMM error.

5.4.4 Error Handlers

MPI applications are so susceptible to runtime errors during MPI calls as any other type of runtime systems. These errors may generate exceptions that need to be treated, and an MPI implementation may choose which errors it handles and which not. In the case where some errors are not treated, at least they need to be handled by generic error handlers, or even better, permit the user to define personalized error handlers. MPI defines mechanisms for both of these solutions. In the case of LWMPI, we implemented

the three predefined error handlers defined in the specification, which we present next, and that can be seen in Table 3.

Table 3 – Predefined MPI Error Handlers.

Error handlers
MPI_ERRORS_ARE_FATAL
MPI_ERRORS_ABORT
MPI_ERRORS_RETURN
MPI_ERRHANDLER_NULL

Source: Adapted from MPI-Forum (2020).

MPI_ERRORS_ARE_FATAL When called, it causes the program to abort all the connected MPI processes, independently on their execution state. It is the default error handler associated with the predefined communicators, but may be changed using the appropriate functions.

MPI_ERRORS_ABORT When called in a communicator, it aborts all processes associated to that communicator. The difference from this error handler to the previous one is that it provides a finer control of which processes are aborted. Calling MPI_ERRORS_ABORT on top of MPI_COMM_WORLD, however, is similar to calling MPI_ERRORS_ARE_FATAL since MPI_COMM_WORLD aggregates all active MPI processes in the application.

MPI_ERRORS_RETURN This is the less drastic of the predefined error handlers specified by the MPI standard. This handler, when called, does nothing more than simply returning the error code to the user. This error code must, then, represent a significant value to the user, such as an error class.

MPI_ERRHANDLER_NULL This is not a valid error handler. It is the *null* handler associated with `MPI_Errhandler`, and may be used only to evaluate if a `MPI_Errhandler` object is valid. Passing MPI_ERRHANDLER_NULL for any function that needs a valid `MPI_Errhandler` will raise an `MPI_ERR_ARG` error.

5.4.5 Datatypes

Datatypes are MPI objects that are included as parameters that specify the data being transferred in communications, in which a transfer buffer is defined as a `count` number of successive entries of a datatype object. It helps the programmer to count the number of elements transferred in a message, and the runtime to manage if both the sender and the receiver are transmitting the data in similar formats. Since MPI is designed to be implemented as a library with no additional needs of preprocessing or compilation, the datatype of a communication should be explicitly supplied as an argument to be verified in a send/receive matching (MPI-FORUM, 2020).

Datatypes in an MPI application may be of two types: *basic* predefined datatypes, that are the most common, or *derived* datatypes that are combinations of multiple basic datatypes. Currently, LWMPI implements only the predefined datatypes for the C language, defined in the 4.0 specification. Table 4 shows the list of supported datatypes available to the user, and any of these may be used in send/receive operations as opaque pointers. In the current version of LWMPI, datatypes involved in a communication must be exactly the same between the sender and the receiver to be considered valid, or involve MPI_PACKED or MPI_BYTE in either the sender or the receiver process. Next, we highlight some special datatypes that may not be trivially understood.

MPI_BYTE and MPI_PACKED They can match any byte of data in the MPI standard. This is why these types are the only ones that can carry any type of data in a communication and will avoid type matching verifications. As one can see in Table 4, these types are not defined as a primitive type in C, since they represent raw bunches of bytes. The difference between these two datatypes is that MPI_PACKED is used to send data that was explicitly packed, and consequently, need to be explicitly unpacked, while MPI_BYTE is used to transfer the binary value of a byte as it was stored in the memory. In the current version of LWMPI, these types have no difference between them and both are implemented as raw C `void` pointers.

MPI_DATATYPE_NULL This is not a valid type to be used for transferring data. It is the *null* handler associated with `MPI_Datatype`, and may be used only to evaluate if a `MPI_Datatype` object is valid or not. Passing MPI_DATATYPE_NULL as a parameter for any function that needs a valid `MPI_Datatype` as input will raise an MPI_ERR_TYPE error.

5.5 MPI PROCESS MANAGEMENT

Even though the MPI standard neither describes the *MPI process abstraction* in detail nor how *MPI processes* are managed, most of current MPI implementations provide default startup mechanisms that define how the MPI environment should behave. The idea of separating the program startup from the application itself provides not only more flexibility for heterogeneous environments but also gives more usability to the implementation while offering different possibilities for developers (MPI-FORUM, 2020).

In order to provide an easy way for users to exploit all the features of lightweight manycores, LWMPI takes advantage of this flexibility given by the MPI specification and implements an *MPI process management module*. Our module provides a homogeneous view of the environment while keeping the intrinsic architectural details of the hardware hidden from users, taking the portability of LWMPI to a new level. Since the Nanvix microkernel is intended to be lightweight and to consume a minimum amount of resources

Table 4 – Predefined C Datatypes.

Datatype Name	**C Type**
MPI_CHAR	char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG_INT	signed long long
MPI_LONG_LONG	signed long long
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_LONG_LONG	unsigned long long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wchar_t
MPI_C_BOOL	_Bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_C_COMPLEX	float _Complex
MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_BYTE	(any C type)
MPI_PACKED	(any C type)
MPI_AINT	MPI_Aint
MPI_OFFSET	MPI_Offset
MPI_COUNT	MPI_Count
MPI_DATATYPE_NULL	(no type defined)

Source: Adapted from MPI-Forum (2020).

of the Compute Clusters, it was designed to allow a single system process per Compute Cluster to be spawned. To use the remaining PEs, applications must employ threads. Fortunately, Nanvix implements POSIX Threads (*pthreads*), which is a well-known API defined by the standard *POSIX.1c*.

LWMPI leverages the *thread abstraction* implemented by *pthreads* in Nanvix to allow MPI applications to make use of all PEs available in a Compute Cluster. For that,

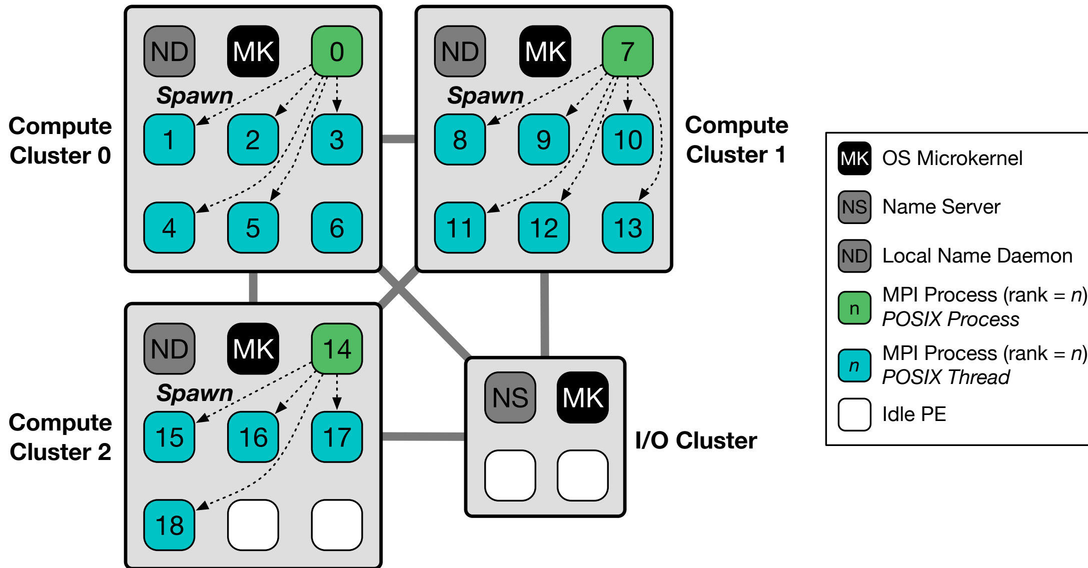


Figure 12 – Overview MPI process management in LWMPI.

LWMPI spawns and manages its own *user-level threads*, exposing them to the user as *MPI processes with distinct MPI ranks*. This allows developers to use more PEs of the architecture in a transparent way and avoids the need of a hybrid programming model (shared-memory + distributed memory). A similar approach was also successfully employed in other MPI implementations such as Adaptive MPI (AMPI) (HUANG; LAWLOR; KALÉ, 2004) and MPC (PÉRACHE; JOURDREN; NAMYST, 2008). From now onward, we will use the term *MPI process* to refer to an *MPI flow of execution that has its own MPI rank, which can either be a system process or a user-level thread in LWMPI*.

Figure 12 pictures how LWMPI manages MPI processes. For the sake of simplicity, let us consider a conceptual lightweight manycore that features three Compute Clusters (composed of nine PEs each) and one I/O Cluster (composed of four PEs). In this example, a PE in each cluster is reserved for the Nanvix microkernel (black box). Moreover, a PE in each Compute Cluster is reserved for the *Local Name Daemon* (gray box), which will be further be discussed in Section 5.6.

Let us now consider an MPI application composed of 19 MPI processes running with LWMPI on top of Nanvix. The first MPI process in each Compute Cluster is a *system process* (green box), which may spawn multiple user-level threads, each one representing a new MPI process with its own MPI rank [2] (blue boxes). In this specific scenario, there are seven MPI processes in Compute Clusters 0–1 and five MPI processes in Compute Cluster 2. In the I/O Cluster, however, there is a *Name Service* that resolves logical process names into logical Compute Cluster identifiers (more details in Section 5.6).

From the MPI application point of view, there is no distinction between a real *system process* (*POSIX Process*) and *user-level threads* (*POSIX threads*) in LWMPI, i.e.,

[2] Since threads cannot share the same PE in the current version of Nanvix, we can only spawn one MPI process per PE.

they are all exposed as MPI processes to developers. Naturally, each MPI process has its own MPI rank and all MPI processes execute the same application code in a MIMD style. Overall, this approach brings the following advantages to lightweight manycores:

Scalability The possibility of using more MPI processes per Compute Cluster allows developers to make use of all PEs of a lightweight manycore.

Lightness Using *pthreads* to implement MPI processes improves the memory consumption in Compute Clusters and allows optimizations in communications between MPI processes that run in the same Compute Cluster via shared memory.

Programmability It improves programmability, since LWMPI manages system processes and user-level threads transparently. As a result, developers do not need to explicitly employ a hybrid programming model such as MPI + *pthreads* to use all PEs of the lightweight manycore.

Although not specified by the MPI standard, many actual MPI applications assume that global variables can be used independently in each MPI process. This is especially true for most of the existing MPI implementations since they leverage the OS process abstraction to implement MPI processes (i.e., each MPI process has its own address space). Implementing MPI processes with user-level threads allows LWMPI to exploit all PEs in a Compute Cluster with a lower memory footprint. However, this approach prevents MPI processes that are running within the same Compute Cluster to be completely isolated from one another. This means that all MPI processes in a Compute Cluster will inevitably share the same address space. Similarly to AMPI (HUANG; LAWLOR; KALÉ, 2004), MPC (PÉRACHE; JOURDREN; NAMYST, 2008), and other MPI implementations that leverage user-level threads to implement MPI processes, having global variables in MPI applications is disallowed in LWMPI.

5.6 THREAD ADDRESSING SCHEME

In Nanvix, the *Name Service* is responsible for linking a logical system process name to the logical Compute Cluster identifier where it is running. Since the OS was designed to allow a single system process per Compute Cluster, any means of intra-cluster addressing was unnecessary. However, we had to overcome this limitation when designing LWMPI, since each MPI process must be addressed individually.

Fortunately, the design of Nanvix IPC abstractions already supports *thread addressing*. Essentially, virtual communicators are linked to physical NoC connectors through logical port identifiers. Then, each thread in a Compute Cluster can reserve a port identifier that, when used in conjunction with its Compute Cluster identifier, represents its logical address within the system. Therefore, peers may use this address to communicate with a specific thread by sending messages to its respective virtual communicator.

Since we attach virtual OS-level communicators to distinct threads, the *Name Service* had to reflect this identification mapping. Specifically, this service must recognize several process names per Compute Cluster with different addresses among them instead of resolving to an unique Compute Cluster identifier. A possible solution to address MPI processes individually would be to simply add the logical port that corresponds to the inbox of a given MPI process rank in the same name entry that already stores its associated Compute Cluster identifier. However, modifying a consolidated OS service would either break its entire interface or overload it with features that only concern LWMPI.

To overcome this problem, we designed an extension to the traditional *Name Service* that can be enabled when using LWMPI. To do so, we kept the original *Name Server* centralized to handle name queries while opting for a distributed scheme to resolve address lookups. When the proposed extension is enabled, a *Local Name Daemon* is spawned in each Compute Cluster of the lightweight manycore. In particular, this daemon uses a *Local Name Table* that contains the logical addresses of all MPI process names associated with the Compute Cluster to respond name lookup requests related to local MPI process names.

Figure 13 illustrates the protocol for an address lookup operation as well as the internal structures involved in this operation on the aforementioned conceptual lightweight manycore. To improve visibility, we omitted all PEs that are not relevant for this example. In this scenario, an MPI process with rank 8 running on Compute Cluster 1 (*source MPI process*) wants to send a message via `MPI_Send` to the MPI process with rank 1 running on Compute Cluster 0 (*destination MPI process*). The protocol works as follows. First, the centralized *Name Server* is inquired for the number of the Compute Cluster associated with the destination MPI process (①). When the response arrives in the source MPI process (②), it discovers that the destination MPI process resides in Compute Cluster 0. Then, the *Local Name Daemon* running in Compute Cluster 0 is inquired to determine the specific logical address of the destination MPI process (③). When the response arrives in the source MPI process (④), it finally finds the complete logical address of the destination MPI process and can now send a message to it via `MPI_Send` (⑤).

To improve the overall performance of the thread addressing scheme, we implemented in software a small cache of names in each *Local Name Daemon*. This cache reduces the volume of address translation requests that need to be resolved in remote Compute Clusters, especially when multiple MPI processes repeatedly communicate with the same one (e.g., in master/slave models). This optimization drastically reduces the intensity of communications and allows for lookups to be resolved very quickly.

While this distributed approach gives more flexibility, it adds more pressure on the communication subsystem than a centralized approach, since it adds at least an extra pair of messages exchanged between MPI processes and remote *Local Name Daemons*. However, we believe that the benefits of not drastically changing an existing OS service surpass the small overhead introduced by it.

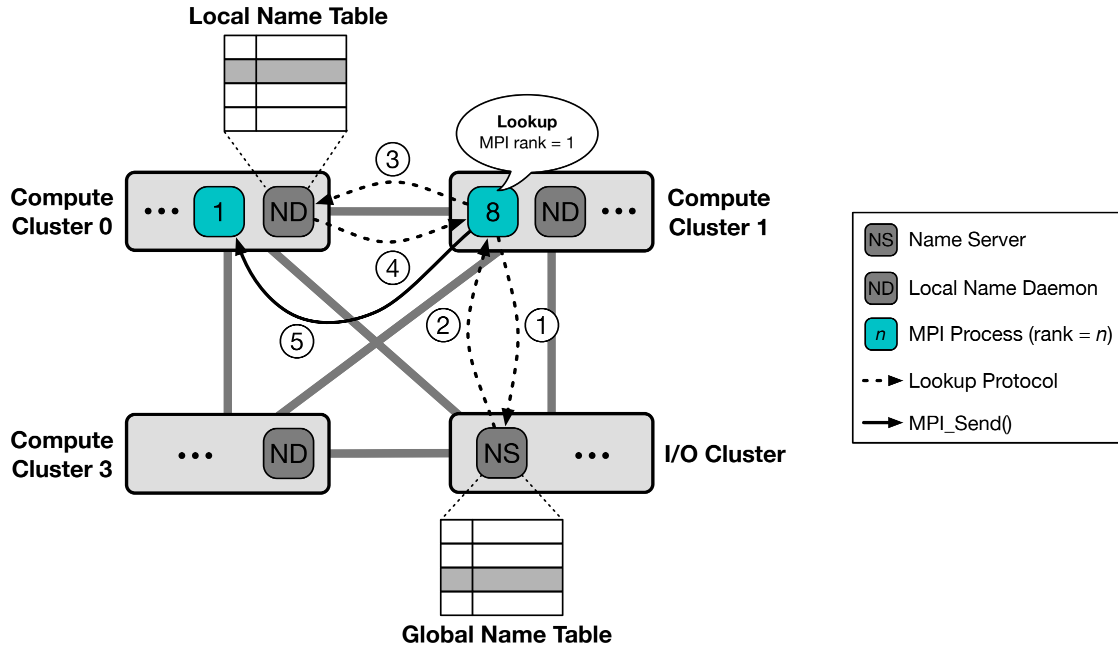


Figure 13 – Protocol for address lookup and internal structures.

5.7 POINT-TO-POINT COMMUNICATION

Sending and receiving messages between the processes is the basic mechanism utilized by MPI. Communication operations may be of two distinct types: (i) point-to-point communication, in which two processes exchange messages directly; and (ii) collective operations, in which a group of processes perform operations together in a coordinated way. Currently, LWMPI only supports point-to-point communication, although implementing collective operations is just a question of implementing the protocols employed and providing optimizations for these collective operations. However, in the present work, we focused on offering support for the point-to-point communication, letting the collective operations as a future work. The basic point-to-point operations in MPI are **send** and **receive**, which are represented by the `MPI_Send` and `MPI_Recv` functions, respectively.

5.7.1 Send and Receive Operations

To match a send operation with its respective receive, there are some attributes that need to be matched between the two operations. These attributes are called the message envelope, which consists in the *source*, *destination*, *tag* and *communicator*. Additionally, the datatypes involved in the communication need to be compatible. In the case of LWMPI, the datatype must be the same for both sender and receiver. The only exception is when at least one of the sides uses the MPI_PACKED or the MPI_BYTE types, that match all the other datatypes defined in the MPI specification.

The MPI standard defines three basic communication modes for the send operation, which are detailed bellow:

Buffered Mode A send operation can be started whether or not a matching receive has been posted. This mode uses intermediary buffering, in which the MPI runtime stores the user message in an internal buffer to be sent when a matching receive is posted. This is a local operation, and the sender may return from the `MPI_Send` function as soon as the message was buffered by the runtime. When a matching receive comes, the runtime transfers the buffered message asynchronously. Although this mode may be the most efficient by blocking the sender for the minimum amount of time, it can be very costly from the memory consumption point of view. In general, only small messages may be buffered to avoid memory exhaustion.

Synchronous Mode A send operation also can be started whether or not a matching receive has been posted. However, it will only complete successfully when the receive operation has started to receive the data through the channel. In general, this mode does not need any type of intermediary buffering, but it may be very "costly", since the sender may be blocked for a long time while waiting for a matching receive to be posted.

Ready Mode The communication can be started only if the matching receive is already posted. Otherwise, the operation is erroneous and may result in undefined behaviors. In general, the idea behind this mode is that it may allow the removal of a handshake, that otherwise is required, in the beginning of the communication protocol. Because of that, this mode may significantly improve the performance, as long as the programmer guarantees program correctness.

It is allowed for the user to specify what mode it wants to use for the send operation. To specify that, it may use the `MPI_Bsend`, `MPI_Ssend` or `MPI_Rsend` functions to use buffered, synchronous or ready modes, respectively. If the user does not want to specify one of these modes, it may use the regular `MPI_Send` function. In this case, the runtime system tries to choose the mode that delivers the best results based on the resource consumption and performance needs. In the current version, LWMPI only implements the *synchronous mode* to carry out the communications to avoid extra memory usage, which is inherent to the buffered mode, and keep the library thin, since memory is a very scarce resource in lightweight manycores. Additionally, the ready mode is not implemented because in the case of the Kalray MPPA-256, specifically, both the sender and the receiver need to know the amount of data that will be transferred, and both sides need to specify the same amount. This way, the handshake in the beginning of the communication also serves for this agreement phase, and may not be removed unless a better protocol becomes available in the Nanvix microkernel. The implementation of these two additional modes is stated as future work.

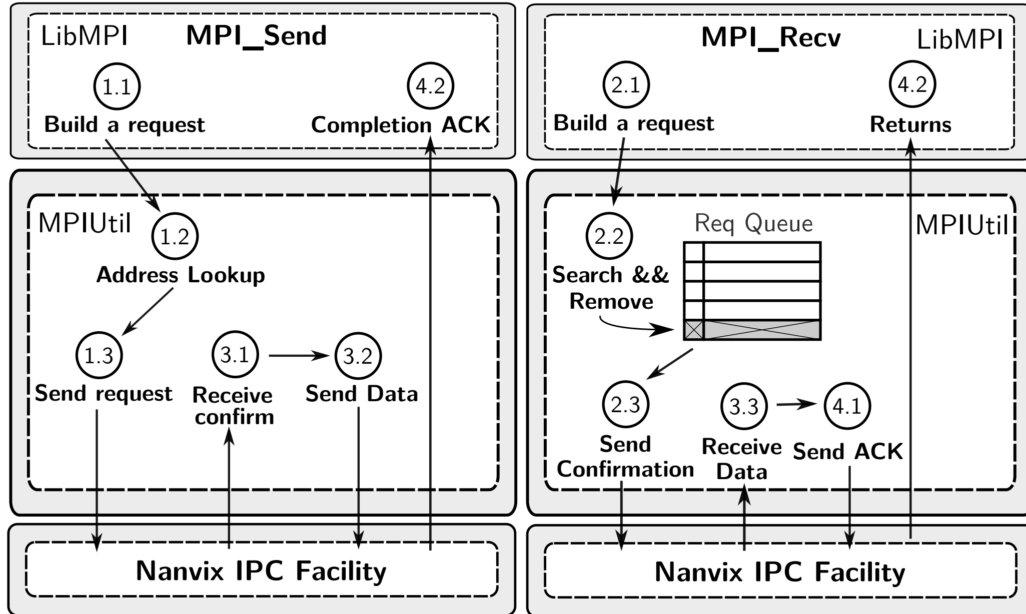


Figure 14 – Interactions between LWMPI and Nanvix.
Source: The author.

5.7.2 Request Cycle

Now, we describe in more details the interactions between LWMPI layers and Nanvix IPC. Figure 14 illustrates how `LibMPI` and `MPIUtil` tiers interact on the sender (left) and receiver (right) sides.

`LibMPI` is responsible for checking the input parameters and creating the communication requests that will be used by `MPIUtil` (steps (1.1) and (2.1)). Requests include the information to be matched between `MPI_Send` and `MPI_Recv`, such as communicator, context, tag, source/destination and, the memory address where the IPC call will use to place/retrieve data to be received/sent. Consequently, this implementation avoids any temporary buffers.

Proceeding with the `MPI_Send` operation, the sender inquires an address lookup to the Nanvix *Name Server* (step (1.2)) if the corresponding translation is not present in the name cache of the *Local Name Daemon* as explained in Section 5.6. With the target MPI process address, the sender submits a *request-to-send* message to the receiver (step (1.3)) through the *mailbox* abstraction and blocks waiting for a confirmation message to arrive from the receiver when a matching `MPI_Recv` is posted. This additional step in the handshake is required to confirm from which port of the remote Compute Cluster the acknowledgment message will come in the completion stage, since we may have more than one MPI process per Compute Cluster.

When an `MPI_Recv` call is issued, the receiver first constructs the communication request. Then, it searches in an internal FIFO queue (step (2.2)) for a send request that matches the received request built in step (2.1). If the queue is empty or no match is found, the receiver waits for a matching request to arrive from the interconnection.

Any other requests that arrive in the meantime are placed at the end of the queue to be fulfilled later.

In Nanvix, threads allocated in the same Compute Cluster share the same physical communication resources, which are distinguished only by their logical addresses (PENNA et al., 2021). Since we do not know in advance which thread will check the underlying buffers when receiving a requisition, all threads in the same Compute Cluster need to agree on a common address from which they can all consume and store messages to unlock the communication mechanisms. Thus, all requests arrive at a common address that is prefixed and known by all threads in the system, and only from step (2.3) onward those communications use the specific addresses of the communicating MPI processes ranks.

When a matching request is found, the receiver consumes and handles it promptly as follows. First, the receiver identifies the sender logical address that comes in the *request-to-send* message, and then sends its own address in the confirmation message (step (2.3)). Along with this confirmation, the receiver grants permission for the data transfer using the *portal* abstraction, allowing the sender to proceed with the communication in steps (3.1) and (3.2). When the receiver starts to receive data through its input *portal*, it sends an *acknowledge* message to the sender via *mailbox* (step (4.1)), indicating to the sender that it can successfully return. Finally, the sender returns from `MPI_Send` when it has sent all of its data and has received the *ack* from the receiver (step (4.2)). The receiver returns from `MPI_Recv` when it has read all the data from the channel or when it has read the amount of data equivalent to the local buffer size.

5.7.3 Communication Protocol

Figure 15 shows the inter-process interaction from the perspective of message exchanges, and gives a bit more of understanding about the communication protocol employed in LWMPI.

The first part of the communication corresponds to an agreement phase that is implemented as a two-step handshake. In the first part of the agreement, the sender submits the *request-to-send* message to the receiver using the *mailbox* abstraction. This message contains all the information necessary for the receiver to build a complete request, that will then be compared and matched with the candidate receive requests.

When the receiver makes a total match, it completes the handshake by sending a confirmation message and emitting an *allow* to the output *portal* of the sender, authorizing it to initiate the transmission through the high bandwidth channel. The receiver will then wait for the data to start to arrive, and will emit a *started-to-receive* message, using *mailbox* again, to the sender, signalizing that it started to receive the data. This *started-to-receive* message is the acknowledgment message that tells to the sender that it can successfully return when it has transmitted all the data through the channel.

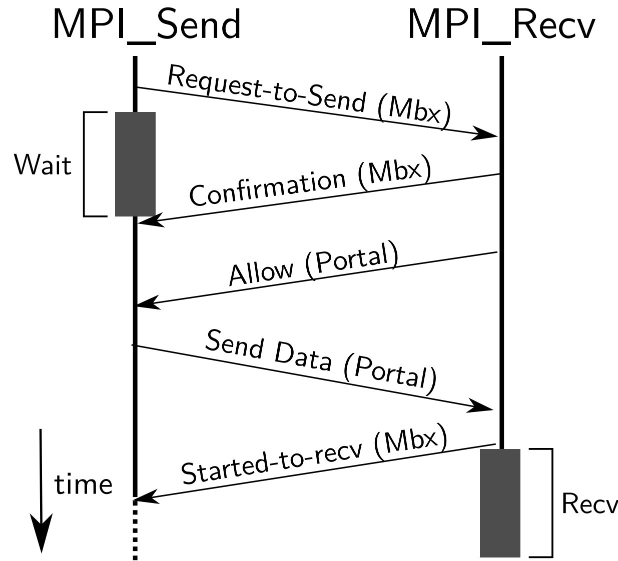


Figure 15 – Communication protocol.
Source: The author.

5.7.4 Local Communication Optimization via Shared Memory

It is important to note that the protocol presented in Figures 14 and 15 is generic enough to carry out both local and remote communications by taking advantage of the transparency given by the Nanvix IPC abstractions on handling specificities of each type of communication. The Nanvix IPC module itself leverages the shared memory in a Compute Cluster to provide faster local communications that do not use the NoC. However, the IPC module still interacts with the Nanvix asymmetric microkernel, resulting in undesired overheads when several MPI processes are running in parallel.

To avoid the aforementioned problem, and trying to handle local communications even faster, we propose a new communication protocol in LWMPI that is especially designed to handle local communications almost completely in user space. The main advantages of this new communication protocol are the following:

(i) it considerably reduces the number of system calls invocations, thus minimizing the pressure over the Nanvix asymmetric microkernel; and
(ii) it reduces the number of intermediate copies of internal buffers, thereby enabling much faster communications for all MPI processes within the Compute Cluster.

Figure 16 presents the new protocol to handle intra-cluster communications. Similar to the non-optimized version, sender and receiver peers first build requests that will be matched to establish the communication (steps (1.1) and (2.1)). The difference for this new version is that when the sender dispatches an address lookup request (step (1.2)), it will receive a local address as a response and will proceed with the new part of the protocol. First, it reserves a buffer slot (step (1.3)) in a new data structure that associates a pointer in the local memory of the Compute Cluster (i.e., the pointer to the user-level

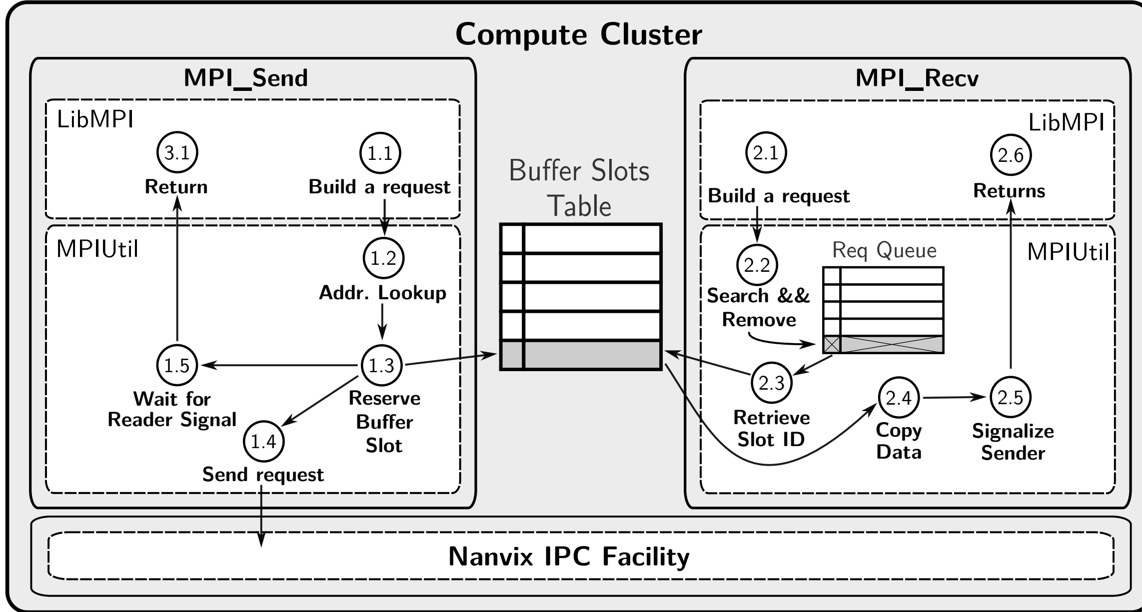


Figure 16 – Interactions between LWMPI and Nanvix in local communications.

buffer), with an identifier that represents the buffer slot inside this structure. The sender then adds the reserved buffer identifier in its request and sends it to the receiver using the *mailbox* abstraction (step (1.4)). After that, it blocks waiting for a signal from the receiver to indicate that the buffer slot is free (step (1.5)).

At the receiver side, after having its request built, it searches for an already received request in the requisitions queue (step (2.2)), or waits for a new request to arrive like in the original protocol. When a matching request is found and the communication is local, it retrieves the buffer slot identifier (step (2.3)) associated with the received request and copies the data directly from the memory address linked in the respective buffer slot (step (2.4)). When all data were copied from the sender's buffer to the receiver's buffer, the receiver sends a signal to the sender, allowing the sender to safely reuse that buffer (step (2.5)). At this point, both receiver and sender are ready to return (steps (2.6) and (3.1), respectively).

Overall, with this new protocol we reduce the number of messages exchanged using the IPC module (from 5 messages to a single message), significantly reducing the protocol complexity and the quantity of system calls invoked to carry out the communication. In Chapter 7, we evaluate the benefits of this new optimization when compared to the standard non-optimized communication.

5.8 PROCESS MAPPING POLICIES

Finally, another important feature of LWMPI is the *process mapping policies*, which define how MPI processes with consecutive MPI ranks are assigned to Compute Clusters of a lightweight manycore. Currently, LWMPI supports the following policies:

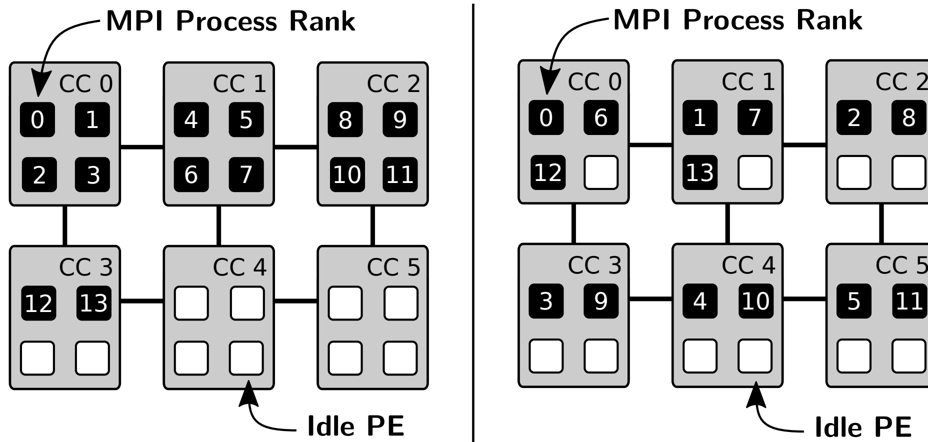


Figure 17 – Example of *compact* (left) and *scatter* (right) policies.

Compact Policy Each MPI process is assigned to a free PE within the same Compute Cluster c. When there is no more free PEs in c, the remaining MPI processes are assigned to a neighbor Compute Cluster according to the NoC topology $(c + 1)$. This procedure is repeated until all MPI processes are assigned to PEs. Overall, this policy concentrates MPI processes in less Compute Clusters, improving resource sharing.

Scatter Policy Each MPI process is assigned to a different Compute Cluster in a round-robin fashion. Overall, this policy distributes MPI processes across the Compute Clusters, reducing local resource contention. Moreover, this policy allows MPI processes to allocate more memory in Compute Clusters, since the number of MPI processes per Compute Cluster is reduced.

Figure 17 illustrates how 14 MPI processes (ranks 0 to 13) are assigned to PEs in a conceptual lightweight manycore with six Compute Clusters (0 to 5) and four PEs per Compute Cluster. As it can be noticed, the *compact* policy assigns all MPI processes to Compute Clusters 0–3, whereas the *scatter* policy spreads MPI processes across all Compute Clusters in a balanced way.

5.9 ADDITIONAL CONSIDERATIONS

In this chapter, we saw the implementation details of LWMPI and how it leverages from an underlying POSIX-compliant OS to deliver programmability and an implicit portability for lightweight manycores. More than that, we saw that LWMPI not only focus in exposing a richer programming interface, but also on extracting the best performance by providing feasible optimizations in its communication protocols and implemented functionalities. One may reason that exploiting these optimizations in user-made solutions is also possible, but, here we argue that it is much more simple to provide these optimiza-

tions and rely in a solidified library rather than reimplementing them each time a new solution is needed.

This way, with LWMPI we take a step ahead when talking about programmability in lightweight manycores. More than a standard and portable interface for message passing communications, we also deliver an easy way for the user to better use the resources of a lightweight manycore, abstracting the majority of the hardware intricacies of these processors, offering them in a transparent way in the application level.

6 EVALUATION METHODOLOGY

In this chapter, we first give a brief description of the applications that we used to evaluate LWMPI. Then, we describe the experimental design employed in this undergraduation dissertation.

6.1 APPLICATIONS

To deliver a comprehensive assessment of LWMPI, we relied on two distinct types of applications:

(i) a synthetic application that stresses the all-to-all communication pattern; and
(ii) three applications extracted from CAP Bench (SOUZA et al., 2017), a benchmark suite designed to assess the performance of lightweight manycores.

All applications were implemented in C language with MPI.[1] An overview of each application is given bellow.

All-to-All (A2A) is a synthetic application that executes a sequence of *supersteps* $s = 0, 1, \ldots, n$ in a Bulk Synchronous Parallel (BSP) scheme but with no actual computation. In a superstep s, each MPI process sends a fixed number of messages carrying a payload of size p bytes to all other MPI processes (`N:N` communication pattern) and blocks on a global barrier. Then, p is exponentially increased before the next superstep ($p = 2^{s+7}$ bytes). The application stops when the last superstep is finished. This application is communication-bound and is employed to stress intra-and inter-cluster communications.

Friendly Numbers (FN) is an application that finds all subsets of numbers in a range $[n, m]$ that share the same *abundance*. The abundance of n is the ratio between the sum of divisors of n by n itself. FN implements the *MapReduce* parallel pattern and has tasks with regular loads. The problem is predominantly CPU-bound.

Gaussian Filter (GF) is a filter that reduces the noise of an image by applying a matrix convolution operation with a special two-dimensional Gaussian mask to the image pixels. GF performs the *Stencil* parallel pattern to equal-sized parts of the image, thus being CPU-intensive and having a medium communication intensity.

K-Means (KM) is a clustering technique employed in data analysis. KM gets a set of n points in real d-dimensional space and randomly split them into k partitions. Then, it applies the *Map* parallel pattern to distribute points and replicate data centroids between the Compute Clusters. The irregular workload is both CPU- and memory-bound. Since each iteration must update data centroids, this kernel operates with high communication intensity.

[1] Publicly available at: https://github.com/nanvix/benchmarks.

The standard CAP Bench applications follow a master/slave model, where a *global master* distributes tasks to *slaves* to be computed. We kept the same approach when implementing the MPI versions of FN, GF and GM applications in our previous work (ULLER et al., 2020a), since we were restricted to a single MPI process per Compute Cluster (maximum of 15 *workers* on Kalray MPPA-256) due to the limitations of both Nanvix and LWMPI. However, this simple model clearly prevents applications to scale to hundreds of *workers*. Since now LWMPI can exploit all PEs in Compute Clusters, we modified the applications to include a *local master* on each Compute Cluster, which is responsible for making the bridge between the *global master* and *slaves*. This modification greatly improved the overall scalability of the applications, since *slaves* running on the same Compute Cluster can communicate locally with their corresponding *local master*, thus reducing the amount of messages transferred through the NoC. Because of that, we adopted this new version in all experiments discussed in this undergraduation dissertation.

6.2 EXPERIMENTAL DESIGN

We carried out all experiments on the baremetal lightweight manycore presented in Section 2.2 (Kalray MPPA-256). In all experiments, we were restricted to 12 MPI processes per Compute Cluster on Kalray MPPA-256 (192 MPI processes when using all Compute Clusters), because:

(i) Nanvix can only spawn a single thread per PE to keep the system simple and to avoid managing thread preemption and scheduling at kernel level;
(ii) one PE is reserved for the Nanvix asymmetric microkernel;
(iii) two PEs are reserved for Nanvix services; and
(iv) one PE is reserved for the *Local Name Daemon* proposed in this paper (Section 5.6).

We conducted two sets of experiments to asses LWMPI. First, we employed the synthetic application (A2A) to evaluate the impacts of the local communication optimization via shared memory presented in Section 5.7.4. In this experiment, we considered scenarios with 12 MPI processes running with different MPI process mapping policies (*compact* and *scatter*) presented in Section 5.8. We employed the optimized version of LWMPI (`LWMPI-opt`) in these experiments.

Second, we carried out weak scaling (GUSTAFSON, 1988) experiments with the aforementioned CAP Bench applications, in which each time we double the number of processes involved, we also double the input problem size. We varied the number of MPI processes from 1 to a maximum of 192, which corresponds to 16 Compute Clusters running 12 MPI processes each. All applications from CAP Bench were executed with the *compact* MPI process mapping policy, since it delivers better performance than *scatter*. We contrasted the results obtained with the applications running with the optimized (`LWMPI-opt`) and unoptimized (`LWMPI-unopt`) versions of LWMPI with their corre-

Type	Name	Abstractions	Parameters	Trials
Synthetic	A2A	`LWMPI-opt`	Payload sizes from 128 B to 32,768 B	10
CAP B.	FN	`LWMPI-unopt/opt`, `IPC`	Numbers in [1000001; 1000001 + N], $N = 1536 * nclusters$	10
	GF	`LWMPI-unopt/opt`, `IPC`	N images, 256×256 pixels, 7×7 mask, $N = 1200 * nclusters$	10
	KM	`LWMPI-unopt/opt`, `IPC`	N 2D points, 128 centroids, $N = 13440 * nclusters$	10

Table 5 – Parameters of synthetic and CAP Bench applications.

sponding implementations using only Nanvix IPC abstractions (`IPC`). The main goal was to evaluate the overhead introduced by LWMPI when compared to the Nanvix low-level programming API for parallel programming.

We collected the following metrics from applications to evaluate LWMPI: execution time and energy consumption. All time measurements were performed using hardware performance counters to enable monitoring with minimum interference. On the other hand, to retrieve energy consumption statistics, we relied on a device that is externally attached to the board of the processor. This device measures power dissipation on the board and comprises statistics for all PEs, NoCs and other on-chip resources. All results are based on a confidence interval threshold of 95% (significance of 5%). The maximum coefficients of variation observed with A2A and CAP Bench applications were 7% and 3%, respectively.

Table 5 presents the parameters used in each application. For CAP Bench applications, the input problem size (N) increases with respect to the number of Compute Clusters running MPI processes ($nclusters$).

7 EXPERIMENTAL RESULTS

In this chapter, we present and discuss our experimental results. First, we evaluate the impacts of MPI process mapping policies when running a synthetic communication-bound application (A2A) with the optimized version of LWMPI (`LWMPI-opt`). Then, we examine the performance and energy consumption of MPI-based implementations of CAP Bench applications when running with the optimized (`LWMPI-opt`) and unoptimized (`LWMPI-unopt`) versions of LWMPI. A comparison with results obtained from IPC-based implementations of these applications (`IPC`) is also presented.

7.1 IMPACTS OF MPI PROCESS MAPPING POLICIES

Figure 18 presents the execution times obtained with the A2A application when executed with 12 MPI processes and with different MPI process mapping policies (*scatter* and *compact*). As expected, *compact* delivered the best execution times, since in this scenario all MPI processes are carrying out local communications. The execution time was nearly constant, no matter the message size. The rationale behind this result is that the time spent in synchronizations among communicating peers dominates the time spent in local data copies from source to destination buffers, making the size of messages involved in local communications irrelevant for determining the overall transfer time.

Scatter, however, achieved a nearly constant execution time with up to 4096-byte messages. After that, the execution times increased significantly along with the size of messages. The nearly constant execution time with up to 4096-byte messages is due to the granularity of data transfers used by the Nanvix IPC *portal* abstraction, which is 4096 bytes (one memory page). This means that any message carrying a payload smaller than 4096 bytes will be transferred in a packet of size 4096 bytes, resulting in a constant transfer

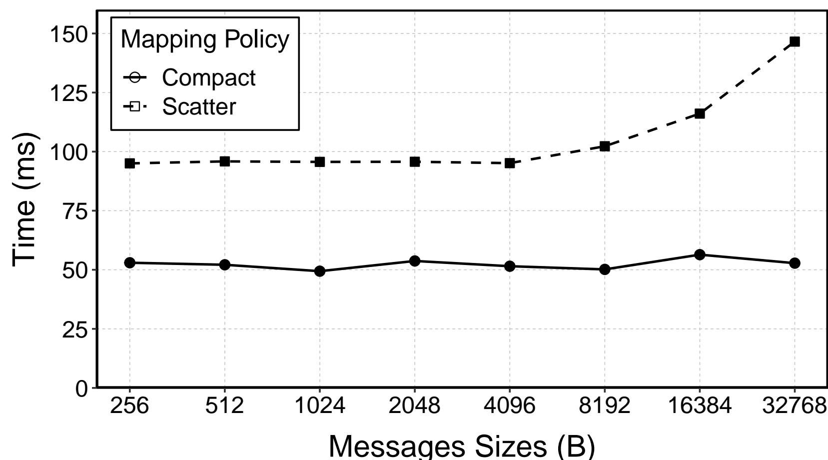


Figure 18 – Execution times obtained with different MPI process mapping policies in a scenario with 12 MPI processes and the optimized version of LWMPI.

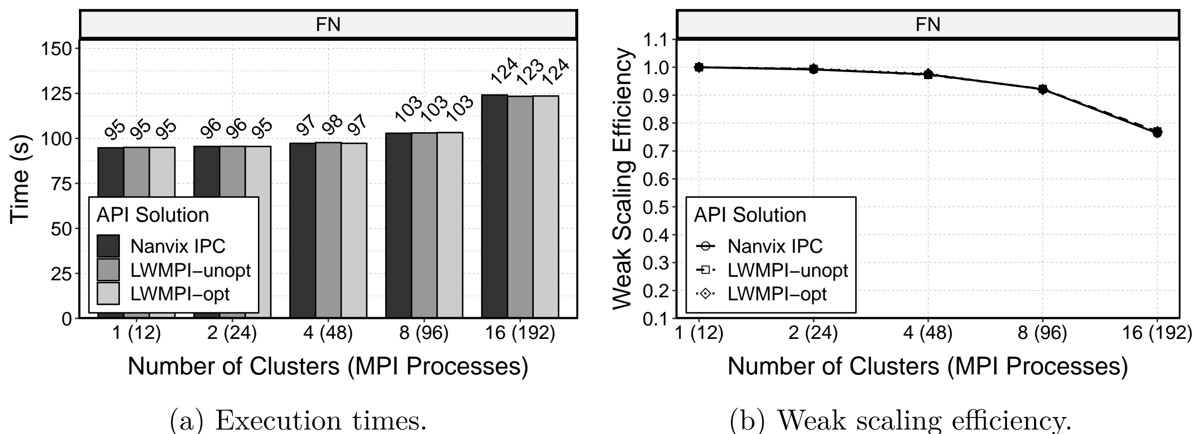
time. Messages carrying payloads greater than 4096 bytes will require more packets to be transferred through the NoC, resulting in higher transfer times.

This experiment allowed us to conclude that the performance gains obtained with intra-cluster communications (*compact* policy) surpass the costs involved in synchronizations to access shared data structures in local memory. Thus, *compact* should be used for communication-bound applications whereas *scatter* is preferable to CPU-bound applications (or to those that make a moderate amount of communications), thus allowing MPI processes to allocate more memory in Compute Clusters.

7.2 PERFORMANCE EVALUATION WITH CAP BENCH APPLICATIONS

7.2.1 FN Application

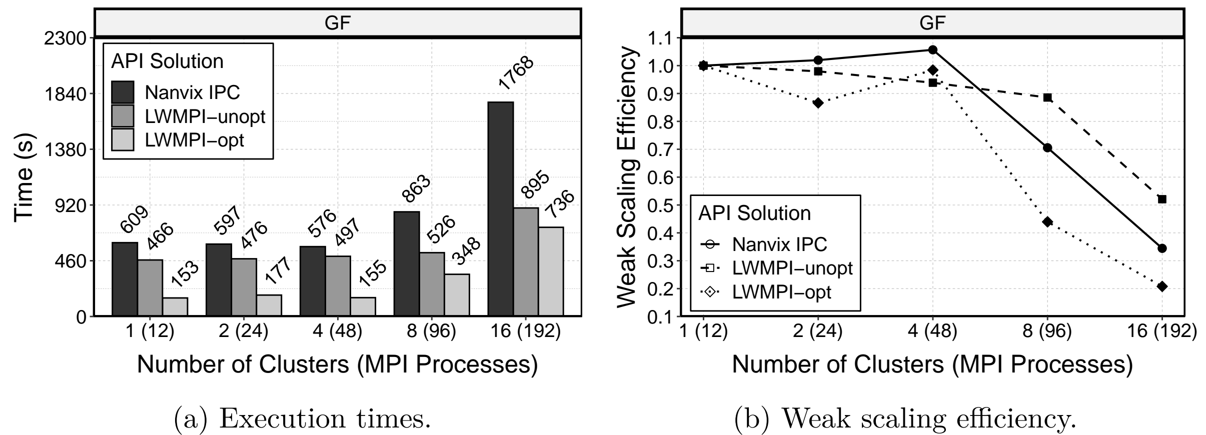
Figure 19a and Figure 19b present execution times and weak scaling results obtained with FN, respectively. In this application, the *global master* distributes equal-sized ranges of numbers to *local masters*, which in turn divide these ranges equally among its associated *slaves* to compute the abundance values. Since FN is a CPU-bound application and has a low communication demand, the results obtained with `IPC`, `LWMPI-unopt` and `LWMPI-opt` solutions are fairly similar. This result is expected, since most of the differences between these solutions come from the way they manage communications. Moreover, such low communication demand in FN is not sufficient to highlight the benefits of `LWMPI-opt` over `LWMPI-unopt`. The efficiency of 77% achieved with 16 Compute Clusters (192 MPI processes in total) shows that this application is able to scale to hundreds of MPI processes.



(a) Execution times.

(b) Weak scaling efficiency.

Figure 19 – FN application results.



(a) Execution times.

(b) Weak scaling efficiency.

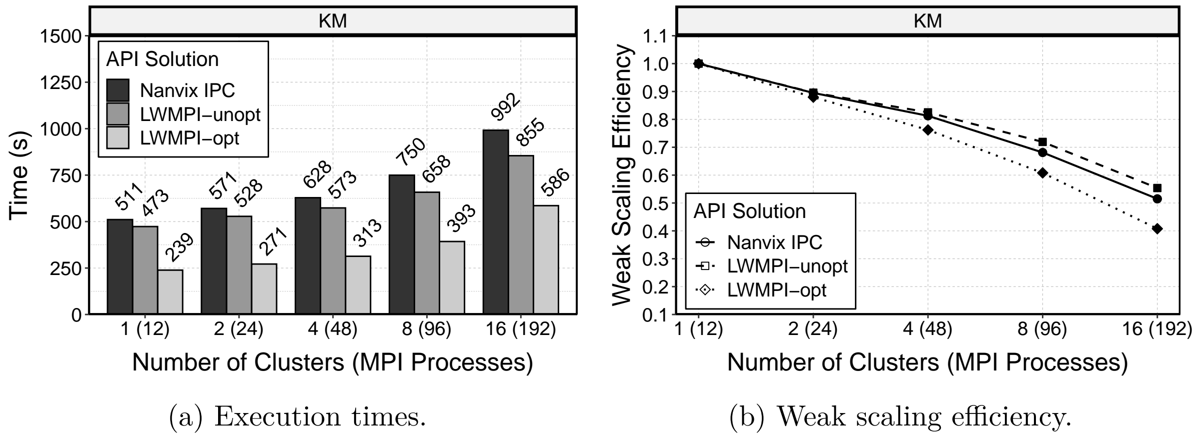
Figure 20 – GF application results.

7.2.2 GF Application

Figure 20a and Figure 20b present execution times and weak scaling results obtained with GF, respectively. In this application, the *global master* distributes images to be processed by *local masters*. Then, each *local master* splits the image into equal-sized chunks and distributes them to its associated *slaves* to perform matrix computations using a Gaussian mask. Overall, `LWMPI-opt` achieved the best execution times, followed by `LWMPI-unopt` and `IPC`. Since GF has a medium communication intensity, `LWMPI-opt` clearly presents a significant improvement when compared to `LWMPI-unopt`, achieving performance gains of up to 3.2× (1.8× on average). The results show that the performance gains achieved by `LWMPI-opt` in comparison to `LWMPI-unopt` tend to decrease as we increase the number of MPI processes. We believe that this performance degradation observed with `LWMPI-opt` is related to the communications between the *global master* and the *local masters*. This completely synchronous communication tends to hide the benefits of the optimized local communications in Compute Clusters, resulting in *local masters* waiting for their turn to communicate with the *global master*.

7.2.3 KM Application

Figure 21a and Figure 21b present execution times and weak scaling results obtained with KM, respectively. In this application, the *global master* iteratively orchestrates the parallel execution by gathering and broadcasting centroids to *local masters*, which then forward the data to *slaves* to perform the actual computation. Again, `LWMPI-opt` achieved the best execution times, followed by `LWMPI-unopt` and `IPC`. We observed a fairly consistent growth in execution times of all solutions as we increased the number of MPI processes. Since KM is a communication-bound application, it is ideal for evaluating the performance gains that can be achieved with the local communication optimization implemented in `LWMPI-opt`. Overall, the lowest performance improvement achieved by



(a) Execution times. (b) Weak scaling efficiency.

Figure 21 – KM application results.

`LWMPI-opt` was 1.4× (scenario with 192 MPI processes), being up to 2× faster than `LWMPI-unopt` in a scenario with 12 MPI processes.

7.2.4 Energy Efficiency Evaluation

Figure 22 shows the power consumption when running KM with 12, 48, and 192 MPI processes. As it can be noticed, the power consumption of Kalray MPPA-256 when running KM with `LWMPI-opt` is slightly higher than with `LWMPI-unopt` and `IPC`. This increase in power consumption is due to the optimizations in local memory communications presented in Section 5.7.4. A similar behavior was also observed on GF. Since execution times of GF and KM are drastically reduced with `LWMPI-opt`, their overall energy consumption is also reduced as shown in Figure 23. As expected, the energy consumption of FN was the same for all solutions because it has very few communications. Overall, execution times and energy consumption follow the same trend on all applications considered in this study.

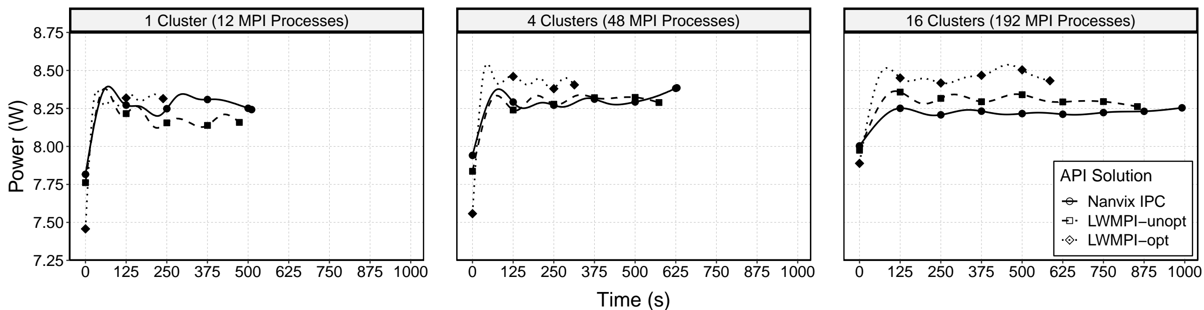


Figure 22 – Power consumption for KM when varying the number of clusters/problem sizes.

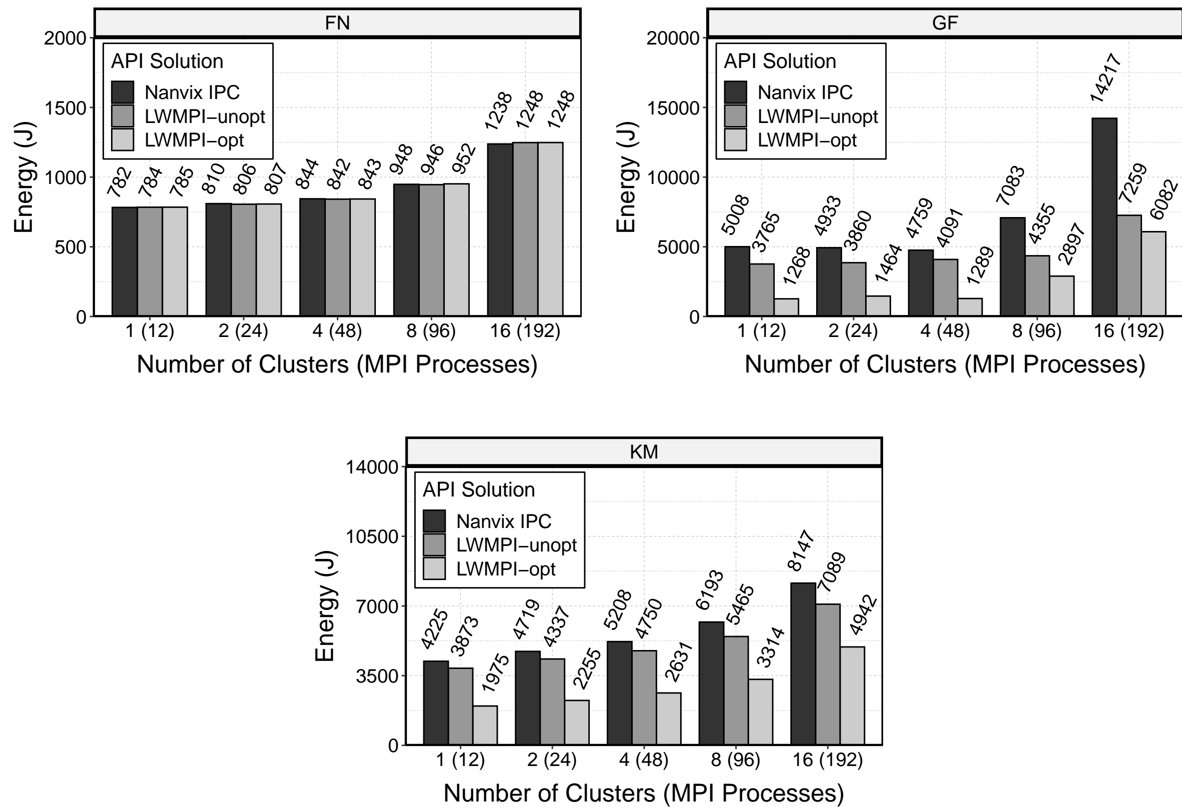


Figure 23 – Energy consumption for FN, GF and KM when varying the exp. scenarios.

7.3 ADDITIONAL CONSIDERATIONS

Overall, LWMPI presented better results, both in performance and energy efficiency, when compared with the Nanvix IPC abstractions solution, especially when considering the optimized version of LWMPI for local communications. Despite unexpected, since an additional overhead instead of direct gains would be natural, this is possible due to the the efforts of LWMPI to present better optimizations for its protocols, like the optimization for local communications discussed in Section 5.7.4. In this case, we saw that the optimized version of LWMPI presents significant gains in terms of performance when compared to its non-optimized version, showing the importance of providing optimizations to extract the better results from both the underlying hardware and software.

Comparing the performance of the optimized LWMPI and the Nanvix IPC solution, it presented the same results in the FN application, and gains ranging from 4× to 2.4× (3.2× on average) for the GF application, and from 2.1× to 1.7× (2× on average) for KM. In terms of power and energy consumption, LWMPI presented a slightly higher power consumption than the Nanvix IPC version, presenting peaks of 8.5 W against 8 W, respectively, exactly because it increases the parallelism in communications. However, as this increase in parallelism results in shorter execution times, LWMPI drastically reduces the final energy consumption of these applications, almost in the same rates that it reduces the total execution times.

At the same time, when we look to the weak scaling speedups obtained with these three applications, we conclude that CPU-bound applications like FN scale better for hundreds of processes (77% efficiency for 192 MPI processes) than those that present more significant communication workloads like GF and KM (52% and 57% efficiency, respectively). This is a totally expected result since the overheads imposed by additional communication demmands in these applications tends to be compensated by the gains in computational power delivered by lightweight manycores.

8 CONCLUSIONS AND FUTURE WORK

Lightweight manycores brought together concepts of parallel and distributed systems into a single die to deliver high performance and energy efficiency. Nevertheless, architectural intricacies and the absence of APIs that embrace programmability and portability make software development an arduous task, specifically because current solutions rely on hardware-dependent and/or vendor-specific APIs.

To unite programmability and portability for lightweight manycores, we proposed LWMPI, a lightweight and portable MPI implementation on top of a POSIX-compliant distributed OS that targets this class of processors (Nanvix). LWMPI is designed from scratch and follows a two-tier approach to separate and self-contain the MPI interface from the OS-dependent layer.

The results obtained with a subset of the CAP Benchmarks applications on the Kalray MPPA-256 processor unveil that LWMPI not only delivers a lightweight and richer programming interface, but also, it presents good performance and scalability for parallel and distributed problems. Instead of an expected overhead, the optimized version of LWMPI even achieved better results in the comparison with a baseline solution that uses the raw Nanvix IPC abstractions.

Overall, LWMPI improved programmability and delivered implicit portability for lightweight manycores without introducing significant overheads that could hinder its adoption, posing itself as a good and viable solution for lightweight manycores.

As future work, some improvements in the LWMPI design and implementation remain as open oportunities:

Adaptative Communications: implementing a mechanism that is capable of dynamically choosing which IPC abstraction fits better the data granularity to be sent, instead of using fixed *portals* to handle the data transfers, would make it possible to send fine-grain messages with low latency and coarse-grain data with high bandwidth;

Messages Forwarding in Nanvix IKC: implementing a message forwarding scheme in the Nanvix IKC facility would permit much faster Address Resolutions for the Nanvix Name Service, and consequently, for LWMPI, since the Name Server would be capable of forwarding an address query directly to the correct Name Daemon, which in turn, answers it directly to the requesting process;

Collective Communications: implementing collective communication operations would not only improve programmability with LWMPI, but also, it would permit better optimizations to be made for this type of communications, giving even more possibilities of improvement for LWMPI.

BIBLIOGRAPHY

ASMUSSEN, N. et al. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In: **International Conference on Architectural Support for Programming Languages and Operating Systems**. Atlanta, Georgia: ACM, 2016. (ASPLOS ‘16), p. 189–203. ISBN 9781450340915. Disponível em: http://dl.acm.org/citation.cfm?doid=2954680.2872371.

BOARD, O. A. R. **OpenMP Application Program Interface Specification - Version 5.1**. 2020. Disponível em: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf.

BOYD-WICKIZER, S. et al. An analysis of linux scalability to many cores. In: **USENIX Conference on Operating Systems Design and Implementation**. Vancouver, Canada: [s.n.], 2010. (OSDI ‘10), p. 1–16.

CHAPMAN, B.; JOST, G.; PAS, R. Using openmp: Portable shared memory parallel programming (scientific and engineering computation). jan 2007.

CHEN, X. et al. Hybrid distributed shared memory space in multi-core processors. **JSW**, v. 6, p. 2369–2378, 12 2011.

CLAUSS, C. et al. Evaluation and improvements of programming models for the Intel SCC many-core processor. In: **International Conference on High Performance Computing & Simulation (HPCS)**. IEEE, 2011. p. 525–532. ISBN 978-1-61284-380-3. Disponível em: http://ieeexplore.ieee.org/document/5999870/.

DAVIDSON, S. et al. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. **IEEE Micro**, IEEE, v. 38, n. 2, p. 30–41, mar 2018. Disponível em: https://ieeexplore.ieee.org/document/8344478/.

DINECHIN, B. D. de et al. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In: **IEEE High Performance Extreme Computing Conference**. Waltham, USA: IEEE, 2013. (HPEC ‘13), p. 1–6. ISBN 978-1-4799-1365-7. Disponível em: http://ieeexplore.ieee.org/document/6670342/.

DINECHIN, B. D. de et al. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. **Procedia Computer Science**, Elsevier, v. 18, n. International Conference on Computational Science, p. 1654–1663, jan 2013. ISSN 1877-0509. Disponível em: https://www.sciencedirect.com/science/article/pii/S1877050913004766?via%3Dihub.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, C-21, n. 9, p. 948–960, 1972.

FRANCESQUINI, E. et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. **Journal of Parallel and Distributed Computing (JPDC)**, Elsevier - Academic Press, Orlando, v. 76, n. C, p. 32–48, february 2015. ISSN 0743-7315. Disponível em: http://linkinghub.elsevier.com/retrieve/pii/S0743731514002093.

FU, H. et al. The Sunway TaihuLight Supercomputer: System and Applications. **Science China Information Sciences**, Science China Press, v. 59, n. 7, p. 072001–0720016, jul 2016. ISSN 1674-733X. Disponível em: http://link.springer.com/10.1007/s11432-016-5588-7.

GAMELL, M. et al. Exploring cross-layer power management for PGAS applications on the SCC platform. In: **International Symposium on High-Performance Parallel and Distributed Computing (HPDC)**. New York, USA: ACM Press, 2012. p. 235. ISBN 9781450308052. Disponível em: http://dl.acm.org/citation.cfm?doid=2287076.2287113.

GREEN500. **Green500 Release**. 2020. Disponível em: https://www.top500.org/lists/green500/2020/06/.

GUSTAFSON, J. L. Reevaluating amdahl's law. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 5, p. 532–533, 5 1988. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/42411.42415.

HAGHBAYAN, M.-H. et al. Performance/reliability-aware resource management for many-cores in dark silicon era. **IEEE Transactions on Computers (TC)**, v. 66, n. 9, p. 1599–1612, sep 2017. Disponível em: http://ieeexplore.ieee.org/document/7892847/.

HASCOËT, J. et al. Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor. In: **Symposium on Embedded Systems for Real-Time Multimedia**. Seoul: ACM Press, 2017. (ESTIMedia '17), p. 51–60. ISBN 9781450351171. Disponível em: http://dl.acm.org/citation.cfm?doid=3139315.3139318.

HO, M. Q. et al. MPI communication on MPPA many-core NoC: Design, modeling and performance issues. In: **International Conference on Parallel Computing**. Edinburgh, UK: IOS Press, 2015. (ParCo '15, v. 27), p. 113–122. Disponível em: https://doi.org/10.3233/978-1-61499-621-7-113.

HUANG, C.; LAWLOR, O.; KALÉ, L. V. Adaptive mpi. In: RAUCHWERGER, L. (Ed.). **Languages and Compilers for Parallel Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 306–322. ISBN 978-3-540-24644-2.

KELLY, B.; GARDNER, W. B.; KYO, S. Autopilot: Message passing parallel programming for a cache incoherent embedded manycore processor. In: **International Workshop on Many-Core Embedded Systems**. New York, NY, USA: Association for Computing Machinery, 2013. (MES '13), p. 62–65. ISBN 9781450320634. Disponível em: https://doi.org/10.1145/2489068.2491624.

KLUGE, F.; GERDES, M.; UNGERER, T. An Operating System for Safety-Critical Applications on Manycore Processors. In: **International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing**. Reno, Nevada: IEEE, 2014. (ISORC '14), p. 238–245. ISBN 978-1-4799-4430-9. Disponível em: http://ieeexplore.ieee.org/document/6899155/.

KOGGE, P. et al. **ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor and Study Lead**. 2008.

LARUS, J.; KOZYRAKIS, C. Transactional memory. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 7, p. 80–88, jul 2008. ISSN 0001-0782. Disponível em: https://doi.org/10.1145/1364782.1364800.

LI, W. et al. Resource virtualization and service selection in cloud logistics. **Journal of Network and Computer Applications**, v. 36, n. 6, p. 1696 – 1704, 2013. ISSN 1084-8045. Disponível em: http://www.sciencedirect.com/science/article/pii/S108480451300057X.

MELPIGNANO, D. et al. Platform 2012, a many-core computing accelerator for embedded socs. In: **Design Automation Conference**. New York, USA: ACM Press, 2012. (DAC '12), p. 1137–1142. ISBN 9781450311991. Disponível em: http://dl.acm.org/citation.cfm?doid=2228360.2228568.

MPI-FORUM. **MPI: A Message-Passing Interface Standard Version 4.0**. 2020. Disponível em: https://www.mpi-forum.org/docs/drafts/mpi-2020-draft-report.pdf.

MPICH. **MPICH: High-Performance Portable MPI**. 2020. Disponível em: https://www.mpich.org.

MUTTIL, N.; LIONG, S.-Y.; NESTEROV, O. A parallel shuffled complex evolution model calibrating algorithm to reduce computational time. **MODSIM07 - Land, Water and Environmental Management: Integrated Systems for Sustainability, Proceedings**, jan 2007.

NELSON, B. Remote procedure call. In: . [S.l.: s.n.], 1981.

NURNBERGER, S. et al. Shared memory in the many-core age. In: LOPES, L. et al. (Ed.). **Euro-Par 2014: Parallel Processing Workshops**. Cham: Springer International Publishing, 2014. p. 351–362. ISBN 978-3-319-14313-2.

PENNA, P. H.; FRANCIS, D.; SOUTO, J. The hardware abstraction layer of nanvix for the kalray mppa-256 lightweight manycore processor. In: **Conférence d'Informatique en Parallélisme, Architecture et Système**. Anglet, France: [s.n.], 2019. p. 1–11.

PENNA, P. H. et al. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In: **Brazilian Symposium on Computing Systems Engineering**. Natal, Brazil: SBC, 2019. (SBESC '19), p. 1–8. ISSN 2324-7894. Disponível em: https://hal.archives-ouvertes.fr/hal-02297637.

PENNA, P. H. et al. Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores. **Journal of Parallel and Distributed Computing (JPDC)**, 2021.

PÉRACHE, M.; JOURDREN, H.; NAMYST, R. Mpc: A unified parallel runtime for clusters of numa machines. In: LUQUE, E.; MARGALEF, T.; BENÍTEZ, D. (Ed.). **Euro-Par 2008 – Parallel Processing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 78–88. ISBN 978-3-540-85451-7.

PFN, I. **MN-3 Supercomputer**. 2020. Disponível em: https://projects.preferred.jp/supercomputers/en/.

RICHIE, D.; ROSS, J.; INFANTOLINO, J. A Distributed Shared Memory Model and C++ Templated Meta-Programming Interface for the Epiphany RISC Array Processor. **Procedia Computer Science**, Elsevier, v. 108, p. 1093–1102, jan 2017. ISSN 1877-0509. Disponível em: http://www.sciencedirect.com/science/article/pii/S1877050917308293.

ROSS, J.; RICHIE, D. Implementing openshmem for the adapteva epiphany risc array processor. **Procedia Computer Science**, Elsevier, v. 80, n. C, p. 2353–2356, jan 2016. Disponível em: http://www.sciencedirect.com/science/article/pii/S1877050916309206.

ROSSI, D. et al. Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster. **IEEE Micro**, IEEE, v. 37, n. 5, p. 20–31, sep 2017. Disponível em: http://ieeexplore.ieee.org/document/8065010/.

SERRES, O. et al. Experiences with UPC on TILE-64 processor. In: **Aerospace Conference**. IEEE, 2011. p. 1–9. ISBN 978-1-4244-7350-2. Disponível em: http://ieeexplore.ieee.org/document/5747452/.

SOUTO, J. V. et al. Mecanismos de comunicação entre clusters para lightweight manycores no nanvix os. In: **Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2020. (ERAD/RS '20), p. 1–4. ISSN 2595-4164. Disponível em: https://sol.sbc.org.br/index.php/eradrs/article/view/10741.

SOUTO, J. V.; PENNA, P. H.; CASTRO, M. **An Inter-Cluster Communication Facility for Lightweight Manycore Processors in the Nanvix OS**. 2019. Trabalho de Conclusao, UFSC. Disponível em: https://repositorio.ufsc.br/bitstream/handle/123456789/202469/joaovicentesouto-tcc.pdf.

SOUZA, M. et al. Cap bench: A benchmark suite for performance and energy evaluation of low-power many-core processors. **Concurrency and Computation: Practice and Experience (CCPE)**, Wiley Online Library, v. 29, n. 4, p. 1–18, february 2017. ISSN 1532-0626. Disponível em: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3892.

SPI. **Open MPI: Open Source High Performance Computing**. 2020. Disponível em: https://www.open-mpi.org.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 4th. ed. USA: Prentice Hall Press, 2014. ISBN 013359162X.

ULLER, J. F. et al. Enhancing programmability in noc-based lightweight manycore processors with a portable mpi library. In: **Simpósio em Sistemas Computacionais de Alto Desempenho**. Online: SBC, 2020. (WSCAD '20), p. 1–12. ISSN 2358-6613.

ULLER, J. F. et al. Proposta de suporte ao padrão mpi sobre infraestrutura de comunicação de baixo nível no nanvix. In: **Escola Regional de Alto Desempenho da Região Sul**. Porto Alegre, RS, Brasil: SBC, 2020. (ERAD/RS '20), p. 121–124. ISSN 2595-4164. Disponível em: https://ojs.sbc.org.br/index.php/eradrs/article/view/10771.

VARGHESE, A. et al. Programming the adapteva epiphany 64-core network-on-chip coprocessor. In: **International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. Phoenix, USA: IEEE, 2014. (IPDPSW '14), p. 984–992. ISBN 978-1-4799-4116-2. Disponível em: http://ieeexplore.ieee.org/document/6969488/.

WALLENTOWITZ, S. et al. A Framework for Open Tiled Manycore System-On-Chip. In: **International Conference on Field Programmable Logic and Applications**. Oslo: IEEE, 2012. (FPL '2012), p. 535–538. ISBN 978-1-4673-2256-0. Disponível em: http://ieeexplore.ieee.org/document/6339273/.

WENTZLAFF, D.; AGARWAL, A. Factored operating systems (fos): The case for a scalable operating system for multicores. **ACM SIGOPS Operating Systems Review**, ACM, v. 43, n. 2, p. 76–85, apr 2009. ISSN 0163-5980. Disponível em: http://portal.acm.org/citation.cfm?doid=1531793.1531805.

WIJNGAART, R. F. van der; MATTSON, T. G.; HAAS, W. Light-weight communications on intel's single-chip cloud computer processor. **SIGOPS Operating Systems Review (OSR)**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 1, p. 73–83, feb 2011. ISSN 0163-5980. Disponível em: https://doi.org/10.1145/1945023.1945033.

WU, Y. et al. Parallelization of a hydrological model using the message passing interface. **Environmental Modelling and Software**, v. 43, p. 124–132, 05 2013.

APPENDIX A – SCIENTIFIC ARTICLE

Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

ARTICLE TYPE

LWMPI: An MPI Library for NoC-Based Lightweight Manycore Processors with On-Chip Memory Constraints

João Fellipe Uller[1] | João Vicente Souto[1] | Pedro Henrique Penna[2,3] | Márcio Castro[1] | Henrique Freitas[2] | Jean-François Méhaut[3]

[1]Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD), Universidade Federal de Santa Catarina (UFSC), Santa Catarina, Brazil
[2]Computer Architecture and Parallel Processing Team (CArT), Pontifícia Universidade Católica de Minas Gerais (PUC Minas), Minas Gerais, Brazil
[3]Laboratoire d'Informatique de Grenoble (LIG), Université Grenoble Alpes (UGA), Auvergne-Rhône-Alpes, France

Correspondence
Email: joao.f.uller@grad.ufsc.br, joao.vicente.souto@posgrad.ufsc.br, pedro.penna@sga.pucminas.br, marcio.castro@ufsc.br, cota@pucminas.br, jean-francois.mehaut@univ-grenoble-alpes.fr

Summary

Lightweight manycore processors deliver high performance and energy efficiency by bundling hundreds of low-power cores, a distributed memory architecture with small local memories and Networks-on-Chip (NoCs) in a single die. However, the lack of rich and portable programming models for these processors makes software development a challenging task. Currently, two approaches are employed to address programmability in lightweight manycores: Operating Systems (OSes) and baremetal runtime libraries. The former provides portability but exposes complex OS-level programming interfaces to developers. The latter focuses on providing rich and high performance interfaces, which are vendor-specific and yield to non-portable software. In this work, we address these programmability and portability challenges by combining a rich OS with a well-known standard for parallel programming. We propose a portable and lightweight Message Passing Interface (MPI) library (LWMPI) designed from scratch to cope with restrictions and intricacies of lightweight manycores. We integrated LWMPI into Nanvix, an open-source distributed OS that runs on silicon lightweight manycores. To deliver a comprehensive assessment of LWMPI, we relied on a synthetic benchmark and three applications extracted from a representative benchmark suite specifically designed for this class of processors. Our results obtained on the Kalray MPPA-256 processor unveiled that applications running with LWMPI achieve better performance and energy efficiency than those implemented with low-level Inter-Process Communication (IPC) abstractions of Nanvix.

KEYWORDS:
lightweight manycores, MPI, runtime systems, memory constraints

1 | INTRODUCTION

Lightweight manycore processors emerged to address demands on high performance and energy efficiency [1]. They feature in a single chip hundreds of low-power Multiple Instruction Multiple Data (MIMD) cores [2], also known as Processing Elements (PEs), and a distributed memory system based on Scratchpad Memories (SPMs) [3]. Usually, PEs are grouped in so-called *clusters*, which are interconnected by rich Networks-on-Chip (NoCs). Moreover, these processors may exploit hardware heterogeneity

by featuring PEs (or entire clusters) with different capabilities[4]. Some industry-successful examples of lightweight manycores are Kalray MPPA-256[5], PULP[2], Adapteva Epiphany[6] and Sunway SW26010[7], being the latter employed in the fourth most powerful commercially available supercomputer to date according to TOP500[†] (Sunway TaihuLight).

While the aforementioned architectural features make lightweight manycore processors more scalable in both performance and energy efficiency, they introduce several challenges in software programmability. For instance, their *distributed memory architecture* requires a non-trivial software design to handle data accesses across multiple physical address spaces. Hence, parallel applications should explicitly fetch data from remote memories to local ones to manipulate them[1]. Furthermore, the *small amount of on-chip memory* demands parallel applications to explicitly tile the working data set into chunks and locally manipulate them[8]. Additionally, it is up to the application to take care of data caching and replication to boost performance[9].

Currently, two approaches are employed to address programmability challenges in lightweight manycores: Operating Systems (OSes)[10,11,12] and baremetal runtime systems[13,6,14]. The former is meant to bridge critical programmability gaps imposed by hardware intricacies. The latter, on the other hand, aims at exposing a rich, performance-oriented programming environment narrowed to the underlying architecture. While these two approaches are effective for some use cases, they have a significant duality drawback. Application development directly on top of OSes yields to software portability across architectures, but programming interfaces provided by existing OSes for lightweight manycores are complex and delay the software development process. In contrast, baremetal and vendor-specific runtime systems expose richer interfaces that accelerate the development process, but they exclusively concern to the software stack ecosystem of a specific lightweight manycore. As an immediate consequence, software written on top of these higher-level interfaces end up to be non-portable.

The software stack for lightweight manycores lacks in programmability, once it fails to provide support for both fast development process and software portability. In this work, we address the programmability and portability challenges in lightweight manycores by combining a rich OS with a well-known standard for parallel programming. We propose a lightweight implementation of the Message Passing Interface (MPI) standard (named LWMPI) on top of Nanvix, an open-source distributed OS that targets lightweight manycores[12]. LWMPI is compatible with the MPI specification v3.1 and can be extended to support new features and OSes thanks to its multitier-based design that keeps encapsulation and maintains the top-level library isolated from OS-dependent code. The LWMPI design and implementation proposed in this paper stems from our previous work on improving the programmability of lightweight manycores[15]. We extended our previous work by adding the following new contributions:

MPI Process Management Module In the original implementation of LWMPI[15], we were restricted to a single MPI process per Compute Cluster. Indeed, this limitation came from Nanvix, which was designed to allow a single system process to be spawned per Compute Cluster. To make use of all PEs in a Compute Cluster, developers must rely on the Portable Operating System Interface (POSIX) for threads (*pthreads*). To prevent developers from implementing MPI applications using a hybrid programming model (MPI + *pthreads*), we designed a new module in LWMPI that leverages the thread abstraction in Nanvix and exposes a *unique "MPI process abstraction"* to developers. This allows developers to run existing parallel applications implemented with MPI on Nanvix using all PEs without any source-code changes.

Thread Addressing Considering that MPI processes can either be system processes or threads, an improved addressing scheme had to be designed to allow each MPI process to be addressed individually. Our solution relies on a distributed scheme to resolve address lookups. An LWMPI *daemon* running in each Compute Cluster is responsible for answering name requests related to MPI process names associated to the Compute Cluster. The daemon leverages a small cache implemented in software to make MPI process name resolutions faster, improving the overall performance of the system.

Local Communication Optimization via Shared Memory LWMPI relies on Nanvix Inter-Process Communication (IPC) abstractions to carry out communications. We implemented an optimization that simplifies the communication protocols, leverages the local memory of Compute Clusters and avoids unnecessary interactions with Nanvix for intra-cluster communications.

MPI Process Mapping Policies We designed two different mapping policies that define how MPI processes with consecutive MPI ranks are assigned to Compute Clusters in a lightweight manycore: (i) *compact*, which concentrates MPI processes in less Compute Clusters to improve resource sharing; and (ii) *scatter*, which spreads MPI processes across different Compute Clusters in a round-robin fashion to allow MPI processes to allocate more memory in Compute Clusters.

[†]https://www.top500.org

To assess LWMPI with representative computing workloads, we carried out experiments with a synthetic application that performs all-to-all communications as well as with three applications extracted from the CAP Bench suite[8]: a benchmark designed to evaluate state-of-the-art lightweight manycores. We redesigned the MPI-based implementations of these applications, which were originally presented in our previous work[15], so as to improve their scalability and, ultimately, make stronger performance evaluations with LWMPI. All experiments were executed on the Kalray MPPA-256 processor, a silicon lightweight manycore that features 288 cores in a single chip. Our results unveiled that the local communication optimization, combined with the *compact* mapping policy, improved the time spent in communications in up to 2.8× on the synthetic communication-bound application. Moreover, the results obtained with CAP Bench applications showed that the overhead of LWMPI is negligible when the application is CPU-bound. Applications that featured moderate to high communication intensities were able to benefit from LWMPI, achieving up to 4× better performance than those implemented with OS low-level communication primitives.

The remainder of this work is organized as follows. In Section 2, we cover the background on lightweight manycores. In Section 3, we present our proposal. In Section 4, we detail our evaluation methodology. In Section 5, we discuss our experimental results. In Section 6 we discuss related works. In Section 7, we draw our conclusions.

2 | LIGHTWEIGHT MANYCORE PROCESSORS

In this section, we cover the background on lightweight manycore processors. First, we present an architectural discussion on these processors and how they differ from other parallel architectures. Then, we discuss about the current support for software development in lightweight manycores.

2.1 | Architectural Overview

Lightweight manycores have an endeavor to deliver high performance and energy efficiency in a single die. To achieve this, these processors rely on the following architectural features: (i) thousands of low-power cores; (ii) MIMD capability; (iii) tightly-coupled groups of cores (aka *clusters*); (iv) distributed memory architecture with multiple address spaces and small local memories; (v) reliable and fast NoCs for message-passing; and (vi) heterogeneous processing capabilities in I/O and computing clusters.

To provide substantial insight on lightweight manycores, we consider in this paper an industry-successful tape out of such type of processor: the Kalray MPPA-256[5]. Notwithstanding, the following discussion extends to other lightweight manycores[2,7]. Figure 1 presents an overview of this processor. Overall, Kalray MPPA-256 integrates 288 cores disposed into 20 clusters. Each cluster is composed of heterogeneous and limited hardware capabilities to perform different roles. For instance, I/O Clusters have four Resource Managers (RMs), four NoC interfaces, and 4 MB local Static Random Access Memory (SRAM) to exchange data with external resources and internal clusters. Differently, Compute Clusters have one RM, 16 PEs, one NoC interface, and only 2 MB local SRAM to run user workloads. Cores within the cluster share and have uniform access to hardware resources.

Communication between clusters is exclusively achieved by explicitly exchanging hardware-level messages through two NoCs. Specifically, The Control NoC (C-NoC) enables synchronization and small control messages handover, whereas the Data NoC (D-NoC) supports arbitrary-sized data exchanges. At this point, the I/O heterogeneity among Compute Clusters becomes more evident. On the one hand, I/O Clusters have direct access to a Dynamic Random Access Memory (DRAM) and external

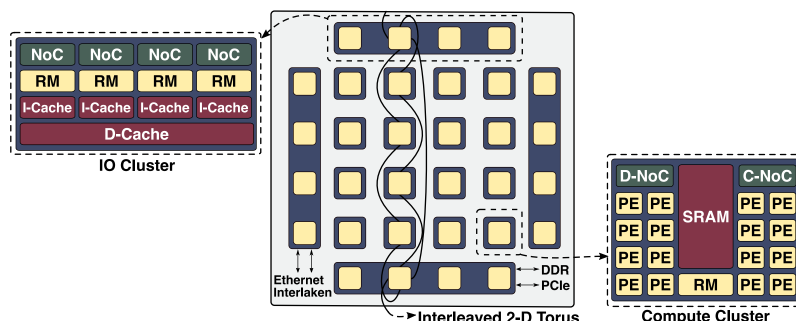


Figure 1 An overview of the Kalray MPPA-256 lightweight manycore processor.

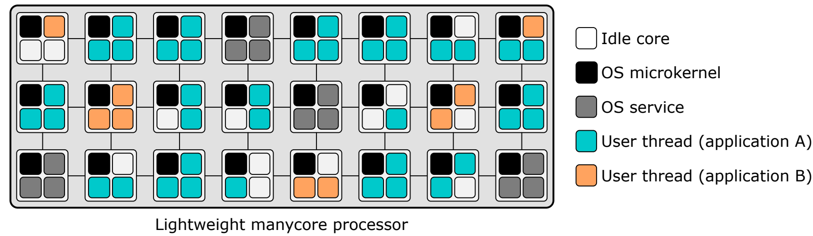


Figure 2 An example of a distributed OS deployed on a hypothetical lightweight manycore.

devices (e.g., Ethernet interface). On the other hand, Compute Clusters must tile their data into messages and send them through the NoC using an I/O Cluster as an intermediary to access these external resources.

The paradigm breakthrough brought by lightweight manycores allows computer systems to scale their performance and energy efficiency. However, challenges introduced by their architectural intricacies to software programmability impact from low- to user-level applications. Examples of these challenges are dark silicon [16], data prefetching and tiling [1], asynchronous communication [9], non-coherent caches [5] and application deployment [8].

2.2 | Software Development Support

There are two approaches currently employed to address programmability challenges in lightweight manycores: OSes and baremetal runtime systems. In the following sections, we examine each of them, and we state where our work is positioned.

2.2.1 | Distributed OSes

OSes are meant to bridge critical programmability gaps in architectures. To this end, they expose rich abstractions to user-level applications, providing resource sharing and multiplexing mechanisms. Inherently due to the architectural features of lightweight manycores, OSes for these processors embrace a distributed design to achieve scalability [17]. In this approach, the OS is factored in a set of services, each of which is deployed on a core of the parallel architecture. Cores that do not run OS services are made available to user-level applications.

Multiple architectures and implementations for a distributed OS are possible, each one targeting a specific set of design goals and constraints. However, a three-tier approach is commonly adopted by distributed OSes for lightweight manycores such as MOSSCA [10], M^3 [11] and Nanvix [12]. In the bottom layer, a generic and flexible Hardware Abstraction Layer (HAL) enables portability across different processor architectures. A *microkernel* lies in the middle layer and provides minimum system abstractions, handles local resource multiplexing and ensures security policies. Finally, in the top layer, runtime OS libraries expose a standard interface to user-level applications such as POSIX.

OS kernel instances may run on all cores of the processor or in a selected set of them (symmetric vs. asymmetric design) [18]. In this work, we are interested on asymmetric multikernel OSes due to their outstanding performance isolation between kernel and user spaces [19]. Figure 2 presents a snapshot of an asymmetric distributed OS running on a lightweight manycore. A unique OS microkernel instance runs on a dedicated core within each cluster. *User threads* and *OS services* run in the remainder cores and request system calls to the kernel through a client-server interface. In this design, performance isolation is delivered because the core where the kernel resides is not time-shared between two execution flows (i.e., kernel and user). Moreover, there is no contention in structures of the OS kernel.

2.2.2 | Baremetal Runtime Systems

In contrast to OSes, baremetal runtime systems export a rich and efficient programming environment narrowed to the underlying architecture. In general, they implement only essential primitives that manage the hardware to avoid unnecessary overheads to the application or fit a specific programming model design. On the one hand, an application can reach optimum performance by utilizing these runtime systems. However, a significant design effort and/or knowledge of the target architecture is usually required. On the other hand, they may not hide low-level aspects of the underlying architecture. Moreover, they do not provide important abstractions that are usually implemented by OSes, such as virtual memory, resource sharing, core multiplexing and others.

Programming models or well-known standards often guide the Application Programming Interface (API) of the runtimes to benefit a specific set of applications. For instance, NodeOS[13] uses the *pipe-and-filter* programming model to allow processes to communicate on Kalray MPPA-256. The primitives exported by NodeOS resemble the classical POSIX pipes interface, but they require specific knowledge from developers. Differently, libasync[9] implements the Asynchronous One-Sided (AOS) communication and synchronization model for Kalray MPPA-256 inspired by libraries used in the High Performance Computing (HPC) domain. The AOS layer defines put/get and atomic operations over requisition queues, allowing applications to read/write data from/to remote memory segments. This model mitigates the problems derived from small local memories in Kalray MPPA-256. However, this approach is focused on enhancing the overall performance of applications, putting aside all programmability and portability issues in lightweight manycores.

To support different programming models for the Adapteva Epiphany lightweight manycore, Epiphany Software Development Kit (eSDK)[6] provides only low-level primitives for communication and synchronization between cores. In this sense, it is up to the developer to manually handle the processor management or resort to external libraries. For instance, CLETE[14] combines meta-programming techniques with data layout and job distribution methodologies to provide a transparent distributed shared memory. Although this additional layer abstracts the peculiarities of the Adapteva Epiphany, this solution is restricted to the chosen programming model and the processor architecture.

2.2.3 | Discussion

In this paper, we focus on implementing and deploying a high-level runtime system on top of a distributed OS for lightweight manycores. We highlight two high-level runtime systems that concern distributed programming and consequently are suitable for lightweight manycores: MPI and Partitioned Global Address Space (PGAS). MPI is an industry-standard interface for message passing programming that exposes one-sided and two-sided communication functions for sending and receiving arbitrary-sized messages. Its communicator abstraction allows multiple logical communication flows within a distributed application. PGAS, on the other hand, is a distributed programming environment that provides a global and shared address spaces over a distributed memory configuration. It relies on a logical partitioning of the address spaces of several processes and provide primitives for reading/writing/synchronizing data from/to these logical partitions. In this paper, we focus on MPI, since it has been the *de facto* standard for parallel programming for more than two decades. It is provided on a wide range of platforms and has been widely used by the scientific computing community.

3 | LWMPI: A MPI LIBRARY FOR LIGHTWEIGHT MANYCORES

Aiming at better programmability in lightweight manycores, we propose LWMPI: an MPI library designed from scratch for these processors. In contrast to alternative solutions for lightweight manycore processors, LWMPI is portable across different architectures, due to its design and implementation on top of a POSIX-compliant distributed OS for lightweight manycores.

3.1 | Design Goals

Lightweight manycores bring several challenges to software development, thereby making easy-to-use interfaces a fundamental requirement for this class of processors. These challenges are not restricted to user-level programming, but also to basic software development. Thus, solutions must meet user demands while dealing with strict architectural constraints, especially those concerning memory-related issues. Hence, we aim at the following requirements for LWMPI:

Portability The library should be portable and applicable to various lightweight manycores.

Compatibility The implementation must comply with the MPI specification.

Extendability It should be possible to add new functions or submodules to the implementation with little effort.

Lightness The implementation should be simple and lightweight to cope with restrictive resources of lightweight manycores.

To satisfy the aforementioned requirements, we rely on important design decisions: (i) design our library on top of an OS to enable *portability* across different architectures; (ii) adhere to the MPI standard to deliver *compatibility*; (iii) follow a tier-based

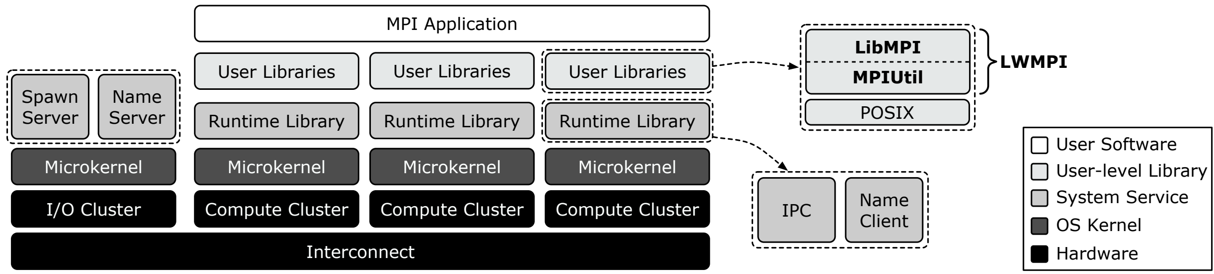


Figure 3 Architectural overview of LWMPI.

approach to keep encapsulation and to maintain the top-level library isolated from OS-dependent implementation, thereby enabling *extendability*; and (iv) implement the library from scratch, rather than adapting an existing heavy-weight solution like OpenMPI[20] or MPICH[21] to keep our solution *light* and suitable for lightweight manycores[22].

We developed our library on top of Nanvix, a POSIX-compliant research OS that targets lightweight manycores[12]. To the best of our knowledge, Nanvix is the only open-source distributed OS that runs on commercially available baremetal lightweight manycores, such as Kalray MPPA-256[5], OpTiMSoC[23] and PULP[2].

3.2 | Overview of LWMPI

LWMPI is an open-source [‡] MPI library for lightweight manycores that follows the MPI specification (version 3.1). Since the implementation of the entire standard would result in a large memory footprint, we decided to implement only a crucial subset of the MPI functionalities, respecting our most important goals and making it suitable for current lightweight manycores.

Figure 3 presents an architectural overview of Nanvix and how LWMPI was introduced in this design. In this figure, we consider a conceptual lightweight manycore composed of one I/O Cluster and three Compute Clusters. Although Nanvix has several OS services and modules, we only present those that are used by our LWMPI implementation.

LWMPI has two logical tiers to isolate the MPI API from OS-dependent software. The `LibMPI` tier is the top-level library and represents the entry point for user applications, encapsulating the standard specification. This layer exposes the library interface and implements the backend functions over the `MPIUtil` tier. At this level, we focus on filtering the input parameters given by the user, performing the runtime management and correctly choosing the protocols employed by each MPI call in the underlying layer. In the current version, our library implements: (i) functions for *runtime management*, such as `MPI_Init` and `MPI_Finalize`; (ii) support for *communicators* and information retrieving, such as `MPI_Comm_rank` and `MPI_Comm_size`; (iii) support for *groups* of communications with functions that are similar to those related with communicators; (iv) *error handlers*; and (v) *point-to-point communication* via `MPI_Send` and `MPI_Recv` using the synchronous mode and carrying any of the predefined *data types* for the C language.

The `MPIUtil` tier is the middle layer between the overlying library and the base OS. It is responsible for translating the requests from `LibMPI` to the Nanvix interface. `MPIUtil` exposes elementary abstractions that support the top-level implementation of the MPI standard, keeping the library implementation decoupled from the OS interface. Some of these abstractions are: (i) *basic objects* applied in all MPI structures; (ii) *processes* for establishing communication groups; and (iii) *communication contexts* that define universes of communication. This layer also implements the MPI communication protocols for point-to-point communications and collective synchronizations.

`MPIUtil` relies on some important components of Nanvix. Specifically, LWMPI uses the *Spawn Server* to spawn a *system process* on each Compute Cluster. As we show later in Section 3.3, this process may spawn user-level threads within the Compute Cluster. The *Name Service*, composed of a *Name Server* and *Name Clients*, is used by `MPIUtil` to address MPI processes. Finally, `MPIUtil` relies on the IPC abstractions to implement all communication protocols. IPC abstractions of Nanvix include primitives for peer synchronization (*sync*), fine-grain fixed-size transfers (*mailbox*) and coarse-grain fixed-size transfers (*portal*). In the next paragraphs, we provide an overview of each one of these abstractions[§].

[‡]LWMPI is available at: https://github.com/nanvix/libmpi

[§]For a more detailed description, please refer to our previous work on Inter-Kernel Communication (IKC) for distributed OSes that target NoC-based lightweight manycores[24].

sync It provides the basis for peer synchronization. It works by having on one side multiple peers (i.e., receivers) to block and wait for peers on the other side (i.e., senders) to issue a notification. The notification itself does not carry any information other than the required to wake up the receivers, thus this abstraction works with fine-grain data.

mailbox It enables peers to exchange fixed-size messages with each other. The size of a message is designed to be small (i.e., hundreds of bytes), so that communication latency is reduced. This abstraction features an `N:1` semantic and works as follows. On the one hand, a receiver owns a *mailbox* from which it reads messages. On the other hand, multiple senders may write messages to this *mailbox*.

portal It allows two peers to exchange arbitrarily large amounts of data (i.e., thousands of bytes) with each other, with built-in support for receiver-side control flow. The granularity of the data is 4 kB in Kalray MPPA-256. This abstraction presents an `1:1` semantic: a receiver owns a *portal*, from which it reads incoming data, and a sender may write data to that *portal*, once a connection with the receiver is established. This connection is explicitly established by the receiver itself by allowing a write on its *portal* from the sender.

3.3 | MPI Process Management

Even though the MPI standard neither describes the *MPI process abstraction* in detail nor how *MPI processes* are managed, most of current MPI implementations provide default startup mechanisms that define how the MPI environment should behave. The idea of separating the program startup from the application itself provides not only more flexibility for heterogeneous environments but also gives more usability to the implementation while offering different possibilities for developers[25].

In order to provide an easy way for users to exploit all the features of lightweight manycores, LWMPI takes advantage of this flexibility given by the MPI specification and implements an *MPI process management module*. Our module provides a homogeneous view of the environment while keeping the intrinsic architectural details of the hardware hidden from users, taking the portability of LWMPI to a new level. Since the Nanvix microkernel is intended to be lightweight and to consume a minimum amount of resources of the Compute Clusters, it was designed to allow a single system process per Compute Cluster to be spawned. To use the remaining PEs, applications must employ threads. Fortunately, Nanvix implements POSIX Threads (*pthreads*), which is a well-known API defined by the standard *POSIX.1c*.

LWMPI leverages the *thread abstraction* implemented by *pthreads* in Nanvix to allow MPI applications to make use of all PEs available in a Compute Cluster. To do so, LWMPI spawns and manages its own *user-level threads*, exposing them to the user as *MPI processes with distinct MPI ranks*. This allows developers to use more PEs of the architecture in a transparent way and avoids the need of a hybrid programming model (shared-memory + distributed memory). A similar approach was also successfully employed in other MPI implementations such as Adaptive MPI (AMPI)[26] and MPC[27]. From now onward, we will use the term *MPI process* to refer to an *MPI flow of execution that has its own MPI rank, which can either be a system process or a user-level thread in LWMPI.*

Figure 4 pictures how LWMPI manages MPI processes. For the sake of simplicity, let us consider a conceptual lightweight manycore that features three Compute Clusters (composed of nine PEs each) and one I/O Cluster (composed of four PEs). In this example, a PE in each cluster is reserved for the Nanvix microkernel (black box). Moreover, a PE in each Compute Cluster is reserved for the *Local Name Daemon* (gray box), which will be further be discussed in Section 3.4.

Let us now consider an MPI application composed of 19 MPI processes running with LWMPI on top of Nanvix. The first MPI process in each Compute Cluster is a *system process* (green box), which may spawn multiple user-level threads, each one representing a new MPI process with its own MPI rank [¶] (blue boxes). In this specific scenario, there are seven MPI processes in Compute Clusters 0–1 and five MPI processes in Compute Cluster 2. In the I/O Cluster, however, there is a *Name Service* that resolves logical process names into logical Compute Cluster identifiers (more details in Section 3.4).

From the MPI application point of view, there is no distinction between a real *system process* (*POSIX Process*) and *user-level threads* (*POSIX threads*) in LWMPI, i.e., they are all exposed as MPI processes to developers. Naturally, each MPI process has its own MPI rank and all MPI processes execute the same application code in a MIMD style. Overall, this approach brings the following advantages to lightweight manycores:

Scalability The possibility of using more MPI processes per Compute Cluster allows developers to make use of all PEs of a lightweight manycore.

[¶] Since threads cannot share the same PE in the current version of Nanvix, we can only spawn one MPI process per PE.

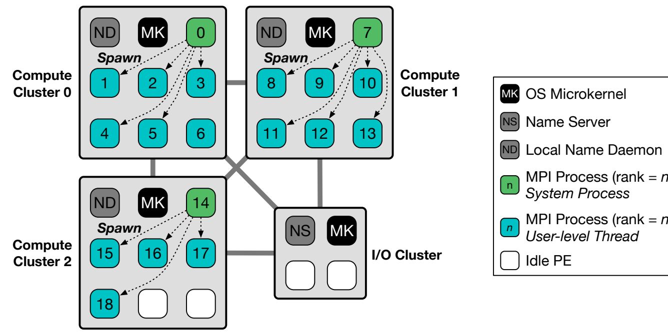


Figure 4 Overview MPI process management in LWMPI.

Lightness Using *pthreads* to implement MPI processes improves the memory consumption in Compute Clusters and allows optimizations in communications between MPI processes that run in the same Compute Cluster via shared memory.

Programmability It improves programmability, since LWMPI manages system processes and user-level threads transparently. As a result, developers do not need to explicitly employ a hybrid programming model such as MPI + *pthreads* to use all PEs of the lightweight manycore.

Although not specified by the MPI standard, many actual MPI applications assume that global variables can be used independently in each MPI process. This is especially true for most of the existing MPI implementations since they leverage the OS process abstraction to implement MPI processes (i.e., each MPI process has its own address space). Implementing MPI processes with user-level threads allows LWMPI to exploit all PEs in a Compute Cluster with a lower memory footprint. However, this approach prevents MPI processes that are running within the same Compute Cluster to be completely isolated from one another. This means that all MPI processes in a Compute Cluster will inevitably share the same address space. Similarly to AMPI[26], MPC[27], and other MPI implementations that leverage user-level threads to implement MPI processes, having global variables in MPI applications is disallowed in LWMPI.

3.4 | Thread Addressing Scheme

In Nanvix, the *Name Service* is responsible for linking a logical system process name to the logical Compute Cluster identifier where it resides. Since the OS was designed to allow a single system process per Compute Cluster, any means of intra-cluster addressing was unnecessary. However, we had to overcome this limitation when designing LWMPI, since each MPI process must be addressed individually. Fortunately, Nanvix IPC abstractions already support *thread addressing*, where virtual communicators are linked to physical NoC connectors through logical port identifiers. Since we now attach virtual OS-level communicators to distinct threads residing in the same Compute Cluster, the *Name Service* had to reflect this identification mapping. Specifically, it must now recognize several MPI process names per Compute Cluster with different addresses.

To overcome this problem, we designed an extension to the traditional *Name Service* that can be enabled when using LWMPI. To do so, we kept the original *Name Server* centralized to handle name queries while opting for a distributed scheme to resolve address lookups. When the proposed extension is enabled, a *Local Name Daemon* is spawned in each Compute Cluster of the lightweight manycore. In particular, this daemon uses a *Local Name Table* that contains the logical addresses of all MPI process names associated with the Compute Cluster to respond name lookup requests related to local MPI process names.

Figure 5 illustrates the protocol for an address lookup operation as well as the internal structures involved in this operation on the aforementioned conceptual lightweight manycore. To improve visibility, we omitted all PEs that are not relevant for this example. In this scenario, an MPI process with rank 8 running on Compute Cluster 1 (*source MPI process*) wants to send a message via `MPI_Send` to the MPI process with rank 1 running on Compute Cluster 0 (*destination MPI process*). The protocol works as follows. First, the centralized *Name Server* is inquired for the number of the Compute Cluster associated with the destination MPI process (①). When the response arrives in the source MPI process (②), it discovers that the destination MPI process resides in Compute Cluster 0. Then, the *Local Name Daemon* running in Compute Cluster 0 is inquired to determine the specific logical address of the destination MPI process (③). When the response arrives in the source MPI process (④), it finally finds the complete logical address of the destination MPI process and can now send a message to it via `MPI_Send` (⑤).

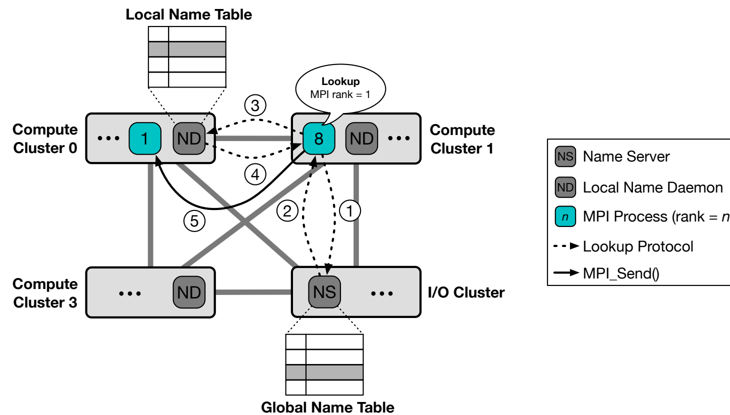


Figure 5 Protocol for address lookup and internal structures.

To improve the overall performance of the thread addressing scheme, we implemented in software a small cache of names in each *Local Name Daemon*. This cache reduces the volume of address translation requests that need to be resolved in remote Compute Clusters, especially when multiple MPI processes repeatedly communicate with the same one (e.g., in master/slave models). This optimization drastically reduces the intensity of communications and allows for lookups to be resolved very quickly.

3.5 | Point-to-Point Communication in LWMPI

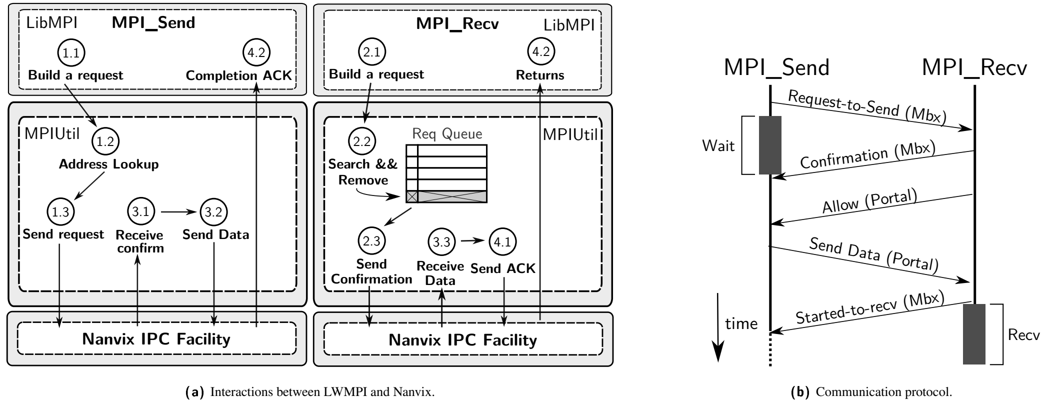
In the current implementation of LWMPI, we only use the *synchronous mode* defined in the MPI specification to perform `MPI_Send` and `MPI_Recv` operations. We decided not to provide the MPI *buffered mode* because:(i) it would be necessary to allocate more memory in LWMPI to store internal buffers, reducing the already constrained memory available for MPI applications; (ii) it would be necessary to introduce an additional daemon to carry out asynchronous operations, since they are not natively supported by the Nanvix IPC module; and (iii) it would need a strict criterion to decide whether an `MPI_Send` will perform a synchronous or a buffered transfer to avoid memory exhaustion in Compute Clusters.

Figure 6a illustrates how `LibMPI` and `MPIUtil` tiers interact on the sender (left) and receiver (right) sides, while Figure 6b pictures the inter-process interaction from the perspective of message exchanges. `LibMPI` is responsible for checking the input parameters and creating the communication requests that will be used by `MPIUtil` (steps (1.1) and (2.1)). Requests include the information to be matched between `MPI_Send` and `MPI_Recv`, such as communicator, context, tag, source/destination and, the memory address where the IPC call will use to place/retrieve data to be received/sent. Consequently, this implementation avoids any temporary buffers.

Proceeding with the `MPI_Send` operation, the sender inquires an address lookup to the Nanvix *Name Server* (step (1.2)) if the corresponding translation is not present in the name cache of the *Local Name Daemon* as explained in Section 3.4. With the target MPI process address, the sender submits a *request-to-send* message to the receiver (step (1.3)) through the *mailbox* abstraction (Figure 6b) and blocks waiting for a confirmation message to arrive from the receiver when a matching `MPI_Recv` is posted. This additional step in the handshake is required to confirm from which port of the remote Compute Cluster the acknowledgment message will come in the completion stage, since we may have more than one MPI process per Compute Cluster.

When an `MPI_Recv` call is issued, the receiver first constructs the communication request. Then, it searches in an internal FIFO queue (step (2.2)) for a send request that matches the received request built in step (2.1). If the queue is empty or no match is found, the receiver waits for a matching request to arrive from the interconnection. Any other requests that arrive in the meantime are placed at the end of the queue to be fulfilled later.

In Nanvix, threads allocated in the same Compute Cluster share the same physical communication resources, which are distinguished only by their logical addresses[24]. Since we do not know in advance which thread will check the underlying buffers when receiving a requisition, all threads in the same Compute Cluster need to agree on a common address from which they can all consume and store messages to unlock the communication mechanisms. Thus, all requests arrive at a common address that is prefixed and known by all threads in the system, and only from step (2.3) onward those communications use the specific addresses of the communicating MPI processes ranks.



(a) Interactions between LWMPI and Nanvix.

(b) Communication protocol.

Figure 6 Implementation details of `MPI_Send` and `MPI_Recv` in LWMPI.

When a matching request is found, the receiver consumes and handles it promptly as follows. First, the receiver identifies the sender logical address that comes in the *request-to-send* message, and then sends its own address in the confirmation message (step (2.3)). Along with this confirmation, the receiver grants permission for the data transfer using the *portal* abstraction, allowing the sender to proceed with the communication in steps (3.1) and (3.2). When the receiver starts to receive data through its input *portal*, it sends an *acknowledge* message to the sender via *mailbox* (step (4.1)), indicating to the sender that it can successfully return. Finally, the sender returns from `MPI_Send` when it has sent all of its data and has received the *ack* from the receiver (step (4.2)). The receiver returns from `MPI_Recv` when it has read all the data from the channel or when it has read the amount of data equivalent to the local buffer size.

3.6 | Local Communication Optimization via Shared Memory

It is important to note that the protocol presented in Figure 6 is generic enough to carry out both local and remote communications by taking advantage of the transparency given by the Nanvix IPC abstractions on handling specificities of each type of communication. The Nanvix IPC module itself leverages the shared memory in a Compute Cluster to provide faster local communications that do not use the NoC. However, the IPC module still interacts with the Nanvix asymmetric microkernel, resulting in undesired overheads when several MPI processes are running in parallel.

To avoid the aforementioned problem, and trying to handle local communications even faster, we propose a new communication protocol in LWMPI that is especially designed to handle local communications almost completely in user space. The main advantages of this new communication protocol are the following:(i) it considerably reduces the number of system calls invocations, thus minimizing the pressure over the Nanvix asymmetric microkernel; and (ii) it reduces the number of intermediate copies of internal buffers, thereby enabling much faster communications for all MPI processes within the Compute Cluster.

Figure 7 presents the new protocol to handle intra-cluster communications. Similar to the non-optimized version, sender and receiver peers first build requests that will be matched to establish the communication (steps (1.1) and (2.1)). The difference for this new version is that when the sender dispatches an address lookup request (step (1.2)), it will receive a local address as a response and will proceed with the new part of the protocol. First, it reserves a buffer slot (step (1.3)) in a new data structure that associates a pointer in the local memory of the Compute Cluster (i.e., the pointer to the user-level buffer), with an identifier that represents the buffer slot inside this structure. The sender then adds the reserved buffer identifier in its request and sends it to the receiver using the *mailbox* abstraction (step (1.4)). After that, it blocks waiting for a signal from the receiver to indicate that the buffer slot is free (step (1.5)).

At the receiver side, after having its request built, it searches for an already received request in the requisitions queue (step (2.2)), or waits for a new request to arrive like in the original protocol. When a matching request is found and the communication is local, it retrieves the buffer slot identifier (step (2.3)) associated with the received request and copies the data directly from the memory address linked in the respective buffer slot (step (2.4)). When all data were copied from the sender's buffer to the receiver's buffer, the receiver sends a signal to the sender, allowing the sender to safely reuse that buffer (step (2.5)). At this point, both receiver and sender are ready to return (steps (2.6) and (3.1), respectively).

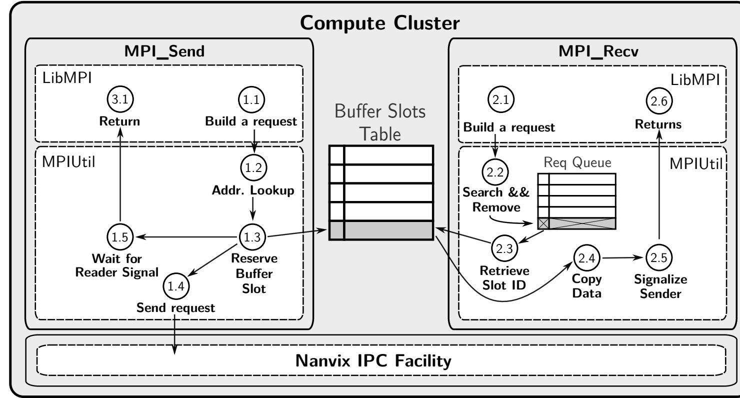


Figure 7 Interactions between LWMPI and Nanvix in local communications.

Overall, the simplified protocol for local communications educes both the number of messages between peers (from five to one) and number of system calls invocations. In Section 5, we evaluate the benefits of this new optimization when compared to the standard non-optimized solution.

3.7 | Process Mapping Policies

Finally, another important feature of LWMPI is the *process mapping policies*, which define how MPI processes with consecutive MPI ranks are assigned to Compute Clusters of a lightweight manycore. Currently, LWMPI supports the following policies:

Compact Each MPI process is assigned to a free PE within the same Compute Cluster c. When there is no more free PEs in c, the remaining MPI processes are assigned to a neighbor Compute Cluster according to the NoC topology ($c + 1$). This procedure is repeated until all MPI processes are assigned to PEs. Overall, this policy concentrates MPI processes in less Compute Clusters, improving resource sharing.

Scatter Each MPI process is assigned to a different Compute Cluster in a round-robin fashion. Overall, this policy distributes MPI processes across Compute Clusters, reducing local resource contention. Moreover, this policy allows MPI processes to allocate more memory in Compute Clusters, since the number of MPI processes per Compute Cluster is reduced.

Figure 8 illustrates how 14 MPI processes (ranks 0 to 13) are assigned to PEs in a conceptual lightweight manycore with six Compute Clusters (0 to 5) and four PEs per Compute Cluster. As it can be noticed, the *compact* policy assigns all MPI processes to Compute Clusters 0–3, whereas the *scatter* policy spreads MPI processes across all Compute Clusters in a balanced way.

4 | EVALUATION METHODOLOGY

In this section, we first give a brief description of the applications that we used to evaluate LWMPI. Then, we describe the experimental design employed in this paper.

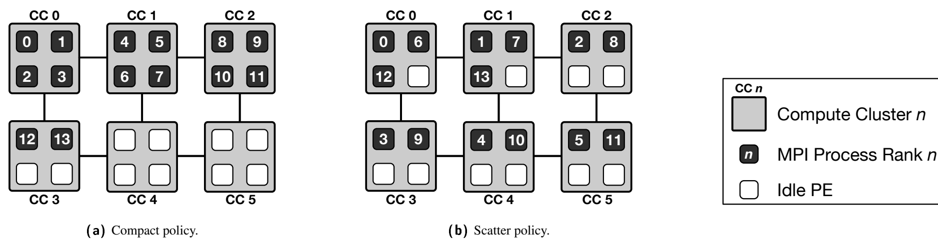


Figure 8 Example of LWMPI process mapping policies.

4.1 | Applications

To deliver a comprehensive assessment of LWMPI, we relied on two distinct types of applications: (i) a synthetic application that stresses the all-to-all communication pattern; and (ii) three applications extracted from CAP Bench[8], a benchmark suite designed to assess the performance of lightweight manycores. All applications were implemented in C language with MPI[#]. An overview of each application is given below.

All-to-All (A2A) is a synthetic application that executes a sequence of *supersteps* $s = 0, 1, \dots, n$ in a Bulk Synchronous Parallel (BSP) scheme but with no actual computation. In a superstep s, each MPI process sends a fixed number of messages carrying a payload of size p bytes to all other MPI processes (`N:N` communication pattern) and blocks in a global barrier. Then, p is exponentially increased before the next superstep ($p = 2^{s+7}$ bytes). The application stops when the last superstep is finished. This application is communication-bound and is employed to stress intra- and inter-cluster communications.

Friendly Numbers (FN) is an application that finds all subsets of numbers in a range $[n, m]$ that share the same *abundance*. The abundance of n is the ratio between the sum of divisors of n by n itself. FN implements the *MapReduce* parallel pattern and has tasks with regular loads. The problem is predominantly CPU-bound.

Gaussian Filter (GF) is a filter that reduces the noise of an image by applying a matrix convolution operation with a special two-dimensional Gaussian mask to the image pixels. GF performs the *Stencil* parallel pattern to equal-sized parts of the image, thus being CPU-intensive and having a medium communication intensity.

K-Means (KM) is a clustering technique employed in data analysis. KM gets a set of n points in real d-dimensional space and randomly split them into k partitions. Then, it applies the *Map* parallel pattern to distribute points and replicate data centroids between the Compute Clusters. The irregular workload is both CPU- and memory-bound. Since each iteration must update data centroids, this kernel operates with high communication intensity.

The standard CAP Bench applications follow a master/slave model, where a *global master* distributes tasks to *slaves* to be computed. We kept the same approach when implementing the MPI versions of FN, GF and GM applications in our previous work[15], since we were restricted to a single MPI process per Compute Cluster (maximum of 15 *slaves* on Kalray MPPA-256). However, this simple model clearly prevents applications to scale to hundreds of *slaves*. Since now LWMPI can exploit all PEs in Compute Clusters, we modified the applications to include a *local master* on each Compute Cluster, which is responsible for making the bridge between the *global master* and *slaves*. This modification greatly improved the overall scalability of the applications because *slaves* running on the same Compute Cluster can communicate locally with their corresponding *local master*. Consequently, we reduce the number of messages transferred through the NoC. Because of that, we adopted this new version in all experiments discussed in this paper.

4.2 | Experimental Design

We carried out all experiments on the baremetal lightweight manycore presented in Section 2.1 (Kalray MPPA-256). In all experiments, we were restricted to 12 MPI processes per Compute Cluster on Kalray MPPA-256, because: (i) Nanvix can only spawn a single thread per PE; (ii) one PE is reserved for the Nanvix asymmetric microkernel; (iii) two PEs are reserved for Nanvix services; and (iv) one PE is reserved for the *Local Name Daemon* proposed in this paper (Section 3.4).

We conducted two sets of experiments to asses LWMPI. First, we employed the synthetic application (A2A) to evaluate the impacts of the local communication optimization presented in Section 3.6. In this experiment, we considered scenarios with 12 MPI processes running with different process mapping policies (*compact* and *scatter*) presented in Section 3.7. We employed the optimized version of LWMPI (`LWMPI-opt`) in these experiments.

Second, we carried out *weak scaling* experiments with applications from CAP Bench. For that, we varied the number of MPI processes from 1 to the maximum of 192 (which corresponds to 16 Compute Clusters running 12 MPI processes each) and increased their input problem sizes (N) with respect to the number of Compute Clusters ($nclusters$). Applications were executed with the *compact* mapping policy and with the two variants of LWMPI (`LWMPI-opt` and `LWMPI-unopt`) as well as with their implementations using only the Nanvix IPC abstractions (`IPC`). Our goals was to evaluate the overhead introduced by LWMPI when compared to the low-level Nanvix API.

[#]Publicly available at: https://github.com/nanvix/benchmarks.

Type	Name	Abstractions	Parameters	Trials
Synthetic	A2A	`LWMPI-opt`, `IPC`	Payload sizes ranging from 128 bytes to 32,768 bytes	30
CAP Bench	FN	`LWMPI-unopt`, `LWMPI-opt`, `IPC`	Numbers in [1000001; 1000001 + N], $N = 1536 * nclusters$	10
	GF	`LWMPI-unopt`, `LWMPI-opt`, `IPC`	N images of 256×256 pixels, 7×7 mask, $N = 1200 * nclusters$	10
	KM	`LWMPI-unopt`, `LWMPI-opt`, `IPC`	N 2D points, 128 centroids, $N = 13440 * nclusters$	10

Table 1 Parameters of synthetic and CAP Bench applications.

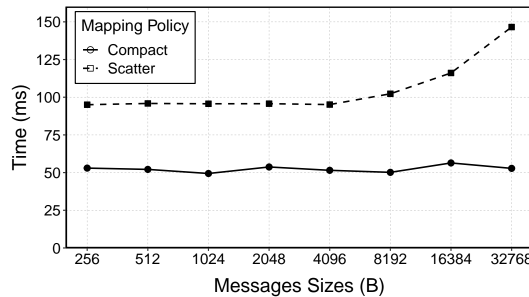


Figure 9 Execution times of A2A with 12 MPI processes and different mapping policies when running with `LWMPI-opt`.

We collected the following metrics from applications to evaluate LWMPI: execution time, power dissipation and energy consumption. All time measurements were extracted from hardware performance counters to enable monitoring with minimum interference. We relied on an external device attached to the Kalray MPPA-256 board to retrieve the power dissipation and energy consumption (these measurements comprise PEs, NoCs, and other on-chip resources). Table 1 summarizes the parameters used in each application. All results are based on a confidence interval threshold of 95% (significance of 5%). The maximum coefficients of the variance observed with A2A and CAP Bench applications 7% and 3%, respectively.

5 | EXPERIMENTAL RESULTS

In this section, we present and discuss our experimental results. First, we evaluate the impacts of MPI process mapping policies when running a synthetic communication-bound application (A2A) with the optimized version of LWMPI (`LWMPI-opt`). Then, we examine the performance and energy consumption of MPI-based implementations of CAP Bench applications when running with the optimized (`LWMPI-opt`) and unoptimized (`LWMPI-unopt`) versions of LWMPI. A comparison with results obtained from IPC-based implementations of these applications (`IPC`) is also presented.

5.1 | Impacts of MPI Process Mapping Policies

Figure 9 presents the execution times obtained with the A2A application when executed with 12 MPI processes and with different MPI process mapping policies (*scatter* and *compact*). As expected, *compact* delivered the best execution times, since in this scenario all MPI processes carry out local communications. The execution time was nearly constant, no matter the message size. The rationale behind this result is that the time spent in synchronizations among communicating peers dominates the time spent in local data copies from source to destination buffers, making the size of messages involved in local communications not significant for determining the overall transfer time.

However, *scatter* achieved a nearly constant execution time with up to 4096-byte messages. After that, the execution times increased significantly along with the size of messages. The nearly constant execution time with up to 4096-byte messages is due to the granularity of data transfers used by the Nanvix IPC *portal* abstraction, which is 4096 bytes long (one memory page). This means that any message carrying a payload smaller than 4096 bytes will be transferred in a packet of size 4096 bytes, resulting in

a constant transfer time. Messages carrying payloads greater than 4096 bytes will require more packets to be transferred through the NoC, resulting in higher transfer times.

This experiment allowed us to conclude that the performance gains obtained with intra-cluster communications (*compact* policy) surpass the costs involved in synchronizations to access shared data structures in local memory. Thus, *compact* should be used for communication-bound applications whereas *scatter* is preferable to CPU-bound applications (or to those that make a moderate amount of communications), thus allowing MPI processes to allocate more memory in Compute Clusters.

5.2 | Weak Scaling Analysis with CAP Bench Applications

Figure 10 presents the weak scaling results for FN, GF, and KM applications based on two metrics: execution times and weak scaling efficiency. In the following paragraphs, we highlight our main findings.

FN is a CPU-bound application and has a low communication demand. Because of that, results obtained with `IPC`, `LWMPI-unopt`, and `LWMPI-opt` solutions are fairly similar. This result is expected since most of the differences between these solutions come from the way they manage communications. Moreover, such low communication demand in FN is not sufficient to highlight the benefits of `LWMPI-opt` over `LWMPI-unopt`. The efficiency of 77% achieved with 16 Compute Clusters (192 MPI processes in total) shows that this application can scale to hundreds of MPI processes.

The time spent in communications is not negligible in GF. We observed that `LWMPI-opt` achieved the best execution times, followed by `LWMPI-unopt` and `IPC`. `LWMPI-opt` clearly presents a significant improvement when compared to `LWMPI-unopt`, achieving performance gains of up to 3.2× (1.8× on average). The results show that the performance gains achieved by `LWMPI-opt` in comparison to `LWMPI-unopt` tend to decrease as we increase the number of MPI processes. We believe that this performance degradation observed with `LWMPI-opt` is related to the communications between the *global master* and the *local masters*. This completely synchronous communication tends to hide the benefits of the optimized local communications in Compute Clusters, resulting in *local masters* waiting for their turn to communicate with the *global master*.

Similarly, `LWMPI-opt` achieved the best execution times for KM, followed by `LWMPI-unopt` and `IPC`. We observed a fairly consistent growth in execution times of all solutions as we increased the number of MPI processes. Since KM is a communication-bound application, it is ideal for evaluating the performance gains that can be achieved with the local communication optimization implemented in `LWMPI-opt`. Overall, the lowest performance improvement achieved by `LWMPI-opt` was 1.4× (scenario with 192 MPI processes), being up to 2× faster than `LWMPI-unopt` in a scenario with 12 MPI processes.

Figure 11 shows the power consumption when running KM with 12, 48, and 192 MPI processes. As it can be noticed, the power consumption of Kalray MPPA-256 when running KM with `LWMPI-opt` is slightly higher than with `LWMPI-unopt` and `IPC`. This increase in power consumption is due to the optimizations in local memory communications discussed in Section 3.6, which allow on-chip resources to be better exploited. A similar behavior was also observed with GF. Since execution times of

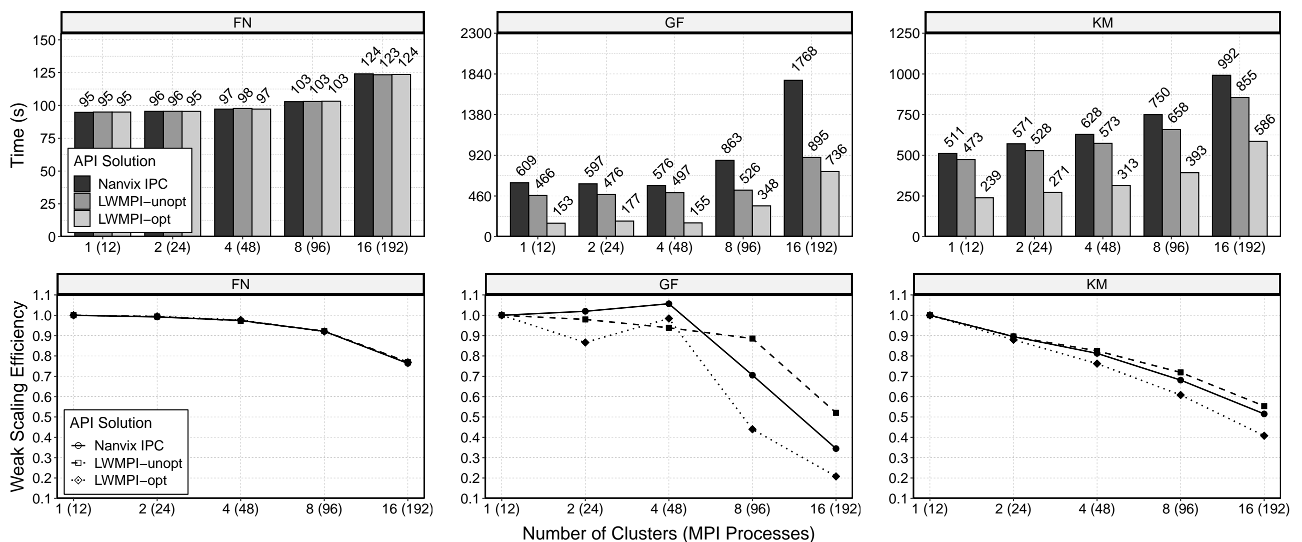


Figure 10 Weak scaling results: execution times (top) and efficiencies (bottom) for FN, GF and KM applications.

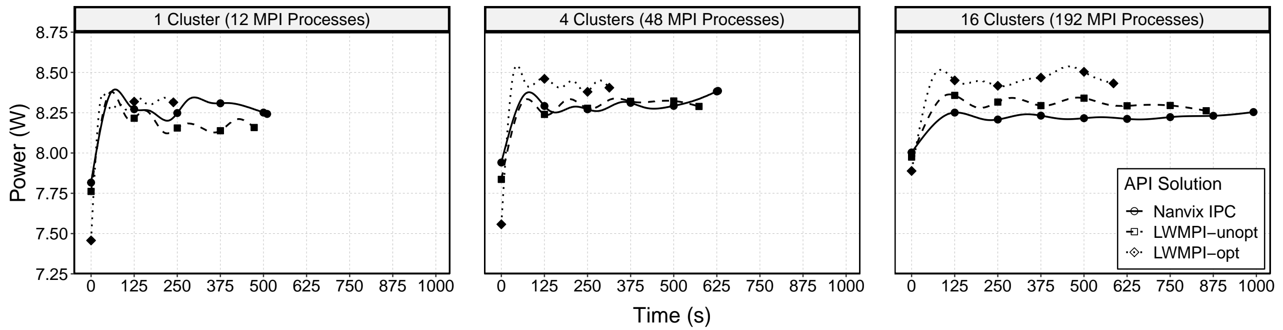


Figure 11 Power consumption (in watts) for KM when varying the number of clusters/problem sizes.

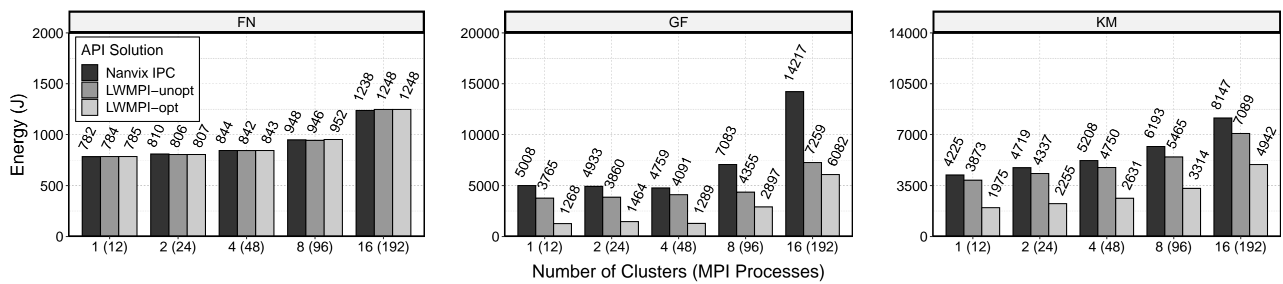


Figure 12 Energy consumption (in joules) for FN, GF and KM when varying the number of clusters/problem sizes.

GF and KM are drastically decreased with `LWMPI-opt`, their overall energy consumption is also reduced as shown in Figure 12. As expected, the energy consumption of FN was the same for all solutions because it has very few communications. Overall, execution times and energy consumption follow the same trend on all applications considered in this study.

6 | RELATED WORK

Software development for lightweight manycores is challenging because current solutions do not provide a good balance between performance and programmability. Currently, the solutions available are either based on vendor-specific communication libraries, which expose a performance-oriented interface narrowed to the underlying architecture, or industry-standard communication libraries, which provide a richer communication interface, in exchange for some performance penalty.

Vendor-specific solutions mostly rely on specific architectural features and may fit programming models to achieve high performance. For instance, Intel Single-Cloud Computer processor supports synchronous [28] and asynchronous [29] communication interfaces on top of the Message Passing Buffer (MPB), an architecture-dependent feature. Alternatively, Kalray MPPA-256 features a POSIX-based communication library [13] and a particular one-sided communication interface [9] on top of it. Conversely, a distributed shared memory using a distinct development flow [14] is provided on top of the low-level communication API [6] of the Adapteva Epiphany processor.

In contrast, industry-standard communication interfaces benefit from the abstraction of hardware aspects to improve programmability and portability. In particular, some well-known distributed programming models may be suitable for lightweight manycores. For instance, Unified Parallel C (UPC) [30] and OpenSHMEM [31] are implementations of the PGAS programming model for the Intel Single-Cloud Computer and Adapteva Epiphany processors, respectively. Alternatively, MPI ports have been proposed for Kalray MPPA-256 [22] and Adapteva Epiphany [32]. However, these solutions differ from our work mainly because: (i) they leverage vendor-specific communication libraries narrowed to a particular lightweight manycore; (ii) they do not conform with the MPI standard; and/or (iii) they require hybrid programming (MPI + OpenMP or *pthreads*) to use all PEs.

In general, both approaches have their advantages and disadvantages. On the one hand, vendor-specific libraries can provide optimal application performance but they require significant design efforts and knowledge of the target architecture. On the other

hand, industry-standard APIs heavily improve programmability issues but current implementations lack of portability and/or require hybrid programming. In this context, our work stands out in two main aspects. First, we provide a flexible, extendable and lightweight MPI implementation designed from scratch on top of an open-source distributed OS for lightweight manycores (Nanvix) to improve both programmability and portability for lightweight manycores. Second, our solution provides a transparent management of MPI processes using user-level threads, which allows for optimizations on intra-cluster communications on lightweight manycores. Contrary to AMPI[26] and MPC[27], our solution is lightweight so as to cope with restrictive local memory resources of lightweight manycores.

7 | CONCLUSION

Lightweight manycores bring concepts of parallel and distributed systems into a single die to deliver high performance and energy efficiency. Nevertheless, the architectural intricacies of lightweight manycores and the absence of APIs that embrace both programmability and portability aspects make software development an arduous task. Currently solutions to improve programmability are narrowed to a specific lightweight manycore and/or are based on non-standard and vendor-specific APIs. Because of that, they are not portable across different lightweight manycore processors.

In order to unite programmability and portability in the context of lightweight manycores, we proposed LWMPI: a lightweight and portable MPI implementation on top of a POSIX-compliant distributed OS that targets this class of processors. LWMPI was designed from scratch and follows a two-tier approach to separate (and self-contain) the MPI interface from the OS-dependent layer. LWMPI manages system processes and user-level threads transparently, exporting to developers a single MPI process abstraction. Moreover, it features optimizations to significantly improve intra-cluster communications via shared memory. Finally, it provides two MPI process mapping policies that define how MPI processes are assigned to Compute Clusters of lightweight manycores. These policies can either improve resource sharing for communication-bound applications by concentrating MPI process in less Compute Clusters or allow applications to allocate more memory in Compute Clusters by spreading MPI processes through different Compute Clusters.

The results obtained with a synthetic benchmark and a subset of the CAP Bench applications running on the Kalray MPPA-256 lightweight manycore processor unveil that LWMPI not only delivers a lightweight and richer programming interface but also presents good performance and scalability results for parallel applications. We also showed that the MPI-based implementations of CAP Bench applications achieved superior performance and energy efficiency when executed with the optimized version of LWMPI (`LWMPI-opt`) in comparison to their low-level implementations using Nanvix IPC abstractions.

As future work, we intend to: (i) implement a mechanism to dynamically choose the IPC abstraction that best fits the data granularity in communications (i.e., *mailbox* for fine-grain messages to improve latency and *portal* for coarse-grain transfers to improve bandwidth); (ii) extend the Nanvix IKC with message forwarding capabilities so that the *Name Server* could forward an address lookup directly to the *Local Name Daemon* residing in the Compute Cluster of the destination MPI process, thus reducing the overhead of address resolutions; and (iii) design and implement collective communications in LWMPI.

ACKNOWLEDGEMENTS

This work was partially supported by *Conselho Nacional de Desenvolvimento Científico e Tecnológico* – Brasil (CNPq) and by *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* – Brasil (CAPES) under the Capes-PrInt Program (grant number 88881.310783/2018-01).

References

1. Francesquini E, Castro M, Penna PH, et al. On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. *Journal of Parallel and Distributed Computing (JPDC)* 2015; 76(C): 32–48. doi: 10.1016/j.jpdc.2014.11.002

2. Rossi D, Pullini A, Loi I, et al. Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster. *IEEE Micro* 2017; 37(5): 20–31. doi: 10.1109/MM.2017.3711645

3. Melpignano D, Benini L, Flamand E, et al. Platform 2012, a Many-Core Computing Accelerator for Embedded SoCs. In: DAC '12. ACM Press; 2012; New York, USA: 1137–1142

4. Davidson S, Xie S, Torng C, et al. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro* 2018; 38(2): 30–41. doi: 10.1109/MM.2018.022071133

5. Dinechin dBD, Ayrignac R, Beaucamps PE, et al. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In: HPEC '13. IEEE; 2013; Waltham, USA: 1–6

6. Varghese A, Edwards B, Mitra G, Rendell AP. Programming the Adapteva Epiphany 64-Core Network-on-Chip Coprocessor. In: IPDPSW '14. IEEE; 2014; Phoenix, USA: 984–992

7. Fu H, Liao J, Yang J, et al. The Sunway TaihuLight Supercomputer: System and Applications. *Science China Information Sciences* 2016; 59(7): 072001–0720016. doi: 10.1007/s11432-016-5588-7

8. Souza M, Penna PH, Queiroz M, et al. CAP Bench: A Benchmark Suite for Performance and Energy Evaluation of Low-Power Many-Core Processors. *Concurrency and Computation: Practice and Experience (CCPE)* 2017; 29(4): 1–18. doi: 10.1002/cpe.3892

9. Hascoët J, Dinechin dBD, Massas dPG, Ho MQ. Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor. In: ESTIMedia '17. ACM Press; 2017; Seoul: 51–60

10. Kluge F, Gerdes M, Ungerer T. An Operating System for Safety-Critical Applications on Manycore Processors. In: ISORC '14. IEEE; 2014; Reno, Nevada: 238–245

11. Asmussen N, Völp M, Nöthen B, Härtig H, Fettweis G. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In: ASPLOS '16. ACM; 2016; Atlanta, Georgia: 189–203

12. Penna PH, Souto J, Lima DF, et al. On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In: SBESC '19. SBC; 2019; Natal, Brazil: 1-8

13. Dinechin dBD, Massas dPG, Lager G, et al. A Distributed Run-Time Environment for the Kalray MPPA-256 Integrated Manycore Processor. *Procedia Computer Science* 2013; 18(International Conference on Computational Science): 1654–1663. doi: 10.1016/j.procs.2013.05.333

14. Richie D, Ross J, Infantolino J. A Distributed Shared Memory Model and C++ Templated Meta-Programming Interface for the Epiphany RISC Array Processor. *Procedia Computer Science* 2017; 108: 1093–1102. doi: 10.1016/J.PROCS.2017.05.221

15. Uller JF, Souto JV, Penna PH, Castro M, Freitas H, Méhaut JF. Enhancing Programmability in NoC-Based Lightweight Manycore Processors with a Portable MPI Library. In: SBC; 2020; Porto Alegre, RS, Brasil: 155–166.

16. Haghbayan MH, Miele A, Rahmani AM, Liljeberg P, Tenhunen H. Performance/Reliability-Aware Resource Management for Many-Cores in Dark Silicon Era. *IEEE Transactions on Computers (TC)* 2017; 66(9): 1599–1612. doi: 10.1109/TC.2017.2691009

17. Boyd-Wickizer S, Clements A, Mao Y, et al. An Analysis of Linux Scalability to Many Cores. In: OSDI '10. ; 2010; Vancouver, Canada: 1–16.

18. Penna PH, Maciel LA, Souto JV, et al. Co-Designing Clusters of Lightweight Manycores and Asymmetric Operating System Kernels. *IEEE Embedded Systems Letters* 2020: 1–5. doi: 10.1109/LES.2020.3040819

19. Nightingale EB, Hodson O, Mcllroy R, Hawblitzel C, Hunt G. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In: SOSP '09. ACM Press; 2009; Big Sky, Montana: 221–234

20. SPI . Open MPI: Open Source High Performance Computing.; 2020.

21. MPICH . MPICH: High-Performance Portable MPI.; 2020.

22. Ho MQ, Tourancheau B, Obrecht C, Dinechin dBD, Reybert J. MPI communication on MPPA Many-Core NoC: Design, Modeling and performance Issues. In: . 27 of *ParCo '15*. IOS Press; 2015; Edinburgh, UK: 113–122

23. Wallentowitz S, Lankes A, Zaib A, Wild T, Herkersdorf A. A Framework for Open Tiled Manycore System-On-Chip. In: FPL '2012. IEEE; 2012; Oslo: 535–538

24. Penna PH, Souto JV, Uller JF, Castro M, Freitas H, Méhaut JF. Inter-Kernel Communication Facility of a Distributed Operating System for NoC-Based Lightweight Manycores. *Journal of Parallel and Distributed Computing (JPDC)* 2021.

25. MPI-Forum . MPI: A Message-Passing Interface Standard Version 4.0.; 2020.

26. Huang C, Lawlor O, Kalé LV. Adaptive MPI. In: Rauchwerger L., ed. *Languages and Compilers for Parallel Computing*Springer Berlin Heidelberg; 2004; Berlin, Heidelberg: 306–322.

27. Pérache M, Jourdren H, Namyst R. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In: Luque E, Margalef T, Benítez D., eds. *Euro-Par 2008 – Parallel Processing*Springer Berlin Heidelberg; 2008; Berlin, Heidelberg: 78–88.

28. Wijngaart v. dRF, Mattson TG, Haas W. Light-Weight Communications on Intel's Single-Chip Cloud Computer Processor. *SIGOPS Operating Systems Review (OSR)* 2011; 45(1): 73–83. doi: 10.1145/1945023.1945033

29. Clauss C, Lankes S, Reble P, Bemmerl T. Evaluation and improvements of programming models for the Intel SCC many-core processor. In: IEEE; 2011: 525–532

30. Gamell M, Rodero I, Parashar M, Muralidhar R. Exploring cross-layer power management for PGAS applications on the SCC platform. In: ACM Press; 2012; New York, USA: 235

31. Ross J, Richie D. Implementing OpenSHMEM for the Adapteva Epiphany RISC Array Processor. *Procedia Computer Science* 2016; 80(C): 2353–2356. doi: 10.1016/J.PROCS.2016.05.439

32. Richie D, Ross J, Park S, Shires D. Threaded MPI programming model for the Epiphany RISC array processor. *Journal of Computational Science* 2015; 9: 94-100. Computational Science at the Gates of Naturedoi: https://doi.org/10.1016/j.jocs.2015.04.023

APPENDIX B – LIST OF IMPLEMENTED FUNCTIONS

MPI_Comm_group(*MPI_Comm comm, MPI_Group *group*)

MPI_Comm_rank(*MPI_Comm comm, int *rank*)

MPI_Comm_size(*MPI_Comm comm, int *size*)

MPI_Comm_get_errhandler(*MPI_Comm comm, MPI_Errhandler *errhandler*)

MPI_Comm_set_errhandler(*MPI_Comm comm, MPI_Errhandler errhandler*)

MPI_Finalize(*void*)

MPI_Finalized(*int *flag*)

MPI_Get_count(*const MPI_Status *status, MPI_Datatype datatype, int *count*)

MPI_Group_rank(*MPI_Group group, int *rank*)

MPI_Group_size(*MPI_Group group, int *size*)

MPI_Group_free(*MPI_Group *group*)

MPI_Init(*int *argc, char ***argv*)

MPI_Initialized(*int *flag*)

MPI_Recv(*void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status*)

MPI_Send(*const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm*)

APPENDIX C – LWMPI SOURCE CODE EXAMPLES

The source code producted during the development of this undergraduation conclusion work, is part of a bigger project called Nanvix OS, which aims at providing a general-purpose distributed operating system for lightweight manycores. More specifically, LWMPI focus on providing a standard communication interface on top of the Nanvix IPC module, as a way of offering better programmability for this class of processors. Since all the project is open-source and collaborative, all of its source code is available on Github.[1] Despite the fact that most of LWMPI source code is concentred at the `Libmpi` Nanvix's repository,[2] the contributions made in the present work are spread over all levels of the Nanvix source tree. Since commenting all the developed code in this dissertation is impossible, in this appendix we present some small portions that serve as example for the ideas employed in LWMPI that were exposed in Chapter 5.

It is important to note that the snippets of code that will be presented here are simplified versions of the original source code available in https://github.com/nanvix/libmpi. The original source files will always be informed, possibiliting for a curious reader to compare the original code with the didatic simplifications that we discuss here. In this dissertation and in this appendix, we are considering the `libmpi` repository in the commit 3fe79bf0e9fe08cfeba8cad65906ebf01dc7868a (3fe79bf).

First, in Section C.1, we present the layers structure of LWMPI by showing the execution flow of a common MPI function (`MPI_Comm_rank`) from its entry point exposed in the `mpi.h` header, until its underlying implementation over the `MPIUtil` exposed abstractions. Next, we present snippets of code for some of the main functions exposed by LWMPI, namely, `MPI_Init`, `MPI_Send` and `MPI_Recv`.

C.1 MPI_COMM_RANK

Starting with `MPI_Comm_rank(`*`MPI_Comm comm, int *rank`*`)`, accordingly with the MPI specification, this function receives an input parameter *comm* and an output parameter rank, that specifies the target communicator in which we want to discover the rank of the calling process and store it, respectivelly.

Algorithm 1 represents the entry point for the `MPI_Comm_rank` function in LWMPI, which can be seen in `src/mpi/communicator/comm_rank.c`. In this snippet, the first concept previously presented is the `LibMPI` tier responsibility in making the first parameters checking, to ensure their validity before proceeding for the function implementation itself. This activity of parameters checking can be seen in lines #5, #8 and #12, where we evaluate if this call is between a `MPI_Init` and a `MPI_Finalize` calls, if *comm* is a valid MPI_Communicator and if the output rank holder is also valid, respectively.

[1] Nanvix is available at: https://github.com/nanvix
[2] LWMPI is available at: https://github.com/nanvix/libmpi

It is important to note that in each different error case, the runtime tries to return a significant error code that helps the user to identify what type of error occured. Proceeding with the routine execution, in line #16 we see the call for an underlying function that implements the next step in the execution flow, while in line #19 a static routine evaluates if an error was returned by the underlying function, and raises an error in the predefined error handler associated with *comm*, as specified in MPI. If no error occured, this function returns the `MPI_SUCCESS` code.

Algorithm 1 MPI_Comm_rank entry point.

Source: Developed by the author.

```
1: function MPI_Comm_rank(MPI_Comm comm, int *rank)
2:     int ret;
3:
4:     # Checks if the runtime was correctly initialized.
5:     MPI_CHECK_INIT_FINALIZE(void);
6:
7:     # Checks if the communicator is valid.
8:     if (mpi_comm_is_valid(comm)) then
9:         return (MPI_ERR_COMM);
10:
11:     # Checks if the rank holder is valid.
12:     if (rank = NULL) then
13:         return (MPI_ERR_ARG);
14:
15:     # Retrieves the current process rank in the given communicator.
16:     ret ← mpi_comm_rank(comm, rank);
17:
18:     # Checks if there was an error in the underlying levels.
19:     MPI_ERRHANDLER_CHECK(ret, comm, ret, MPI_Comm_rank);
20:
21:     return (MPI_SUCCESS);
```

In Algorithm 2, we see the implementation of the underlying `mpi_comm_rank` function seen in Algorithm 1. This file location is `include/mpi/communicator.h`. One interesting point that we see in this function is the fact that is simply a wrapper for another `mpi_group_rank` function. The reason for that arises from the fact that the basis of a communicator is a group of processes. This way, discovering the rank of a given process in a communicator is nothing more than discovering the rank of this process in the communicator's underlying group.

Algorithm 2 mpi_comm_rank underlying function.

Source: Developed by the author.

```
1: function mpi_comm_rank(mpi_communicator_t *comm, int *rank)
2:     return (mpi_group_rank(comm → group, rank));
```

Finally, in Algorithm 3 we present the implementation of `mpi_group_rank`, which is located in `src/mpi/group/group.c`. In this function, we traverse the list of processes of a group (line #8 and on) comparing the reference for the current running process (*curr_proc*) with all elements in this list of processes. Here, we clearly see how `LibMPI` relies in the abstractions exposed by `MPIUtil`, like the MPI Process abstraction. The `curr_mpi_proc` function is a routine exposed by `MPIUtil` in which the reference for the MPI Process associated with the running Nanvix thread is returned. By doing that, functions in the `LibMPI` tier are completelly isolated from the OS-dependent code, as exposed in Chapter 5.

Algorithm 3 mpi_group_rank implementation.
Source: Developed by the author.

```
1: function MPI_GROUP_RANK(mpi_group_t comm, int *rank)
2:     mpi_process_t *curr_proc;
3:
4:     # Retrieves current process reference.
5:     curr_proc ← CURR_MPI_PROC(void);
6:
7:     # Traverses the group's processes list looking for current proc.
8:     for (i = 0; i < group → size; i = i + 1) do
9:         if (group → procs[i] = curr_proc) then
10:            *rank = i;
11:
12:            return (MPI_SUCCESS);
13:
14:    return (MPI_ERR_UNKNOWN);
```

C.2 MPI_INIT

The `MPI_Init` function, which receives two optional input parameters *argc* and *argv*, is the main routine for the runtime initialization. According with the MPI specification, all MPI programs must contain exactly one call to an MPI initialization routine (MPI-FORUM, 2020). In the case of LWMPI, we provide `MPI_Init` for this initialization setup. Since in the last Section we already presented the layers structure of LWMPI, from now on, we refrain ourselves from presenting the whole structure again, to focus exclusively in the underlying implementations.

Algorithm 4 presents the `mpi_init` routine, which is the implementation of `MPI_Init`, found in `src/mpi/runtime/runtime.c`. As we can see in line #2, the first thing that we do in LWMPI is to fork the MPI initialization in two distinct execution flows: one for the MPI process that runs on the system process and acts as a master of the cluster, and another for the MPI processes that runs on the system threads that were spawned by the original process, as explained in Section 5.5. The reason for this is that,

Algorithm 4 Runtime initialization implementation.

Source: Developed by the author.

```
1:  function MPI_INIT(int argc, char **argv)
2:      if (!CURR_PROC_IS_MASTER(void)) then
3:          goto slave;
4:
5:      # MPI_Init already called?
6:      if (mpi_state != MPI_STATE_NOT_INITIALIZED) then
7:          return (MPI_ERR_OTHER);
8:
9:      mpi_state ← MPI_STATE_INIT_STARTED;
10:
11:     # Local fence to ensure that master already spawned all other processes.
12:     MPI_STD_FENCE(void);
13:
14:     # Initialize MPI_Process dependant structures.
15:     MPI_LOCAL_PROC_INIT(void);
16:
17:     # Initialize all MPI structures.
18:
19:     MPI_COMM_REQUEST_INIT(void);
20:
21:     MPI_COMM_INIT(void);
22:
23:     .
24:     .
25:     .
26:
27:     # Marks the runtime as Initialized.
28:     mpi_state ← MPI_STATE_INITIALIZED;
29:
30:     # Last barrier to ensure that all processes in the environment were initialized.
31:     MPI_STD_BARRIER(void);
32:
33:     return (MPI_SUCCESS);
34:
35:     slave:
36:
37:     MPI_STD_FENCE(void);
38:
39:     MPI_LOCAL_PROC_INIT(void);
40:
41:     MPI_STD_BARRIER(void);
42:
43:     return (MPI_SUCCESS);
```

despite the MPI processes that are in the same cluster behave as independent entities during the runtime, internally, they share some common structures to avoid data duplication and a bigger memory footprint. So, it is necessary for one to initialize these common structures, which in our case is this special master of the cluster. It is important to note that this notion of master of the cluster in only internally relevant for the runtime in its initialization and in the finalization. For all the purposes, or from the user's perspective, there is no distinction between this process and the others. The basic structure for these two execution flows, however, is the same:

(i) a **local fence** between all MPI processes in the cluster to ensure that the master already initialized the basic information of all processes that were spawned (lines #12 and #37);
(ii) the call of the `mpi_local_proc_init` routine (lines #15 and #39) to initialize the process-dependent structures, like their inboxes and their inportals that will be used to communicate with other processes;
(iii) a final **barrier** (lines #31 and #41) between all MPI processes in the system to ensure that all of them already finished their initialization and are, consequently, ready to communicate.

Additionally to the common basic structure, the master process controls the runtime actual state (lines #6, #9 and #28), and performs the steps involved in the common structures initialization, which we simplified showing only the `mpi_comm_request_init` and `mpi_comm_init` routines (lines #19 and #21), that initialize the communication requests queue and the predefined communicators, like MPI_COMM_WORLD, respectively. To check the complete initialization routine, refer to the indicated source file.

C.3 MPI_SEND AND MPI_RECV

The `MPI_Send` and `MPI_Recv` functions are the basic routines for sending receiving data, respectively, in point-to-point communication using MPI. Algorithm 5 presents the implementation of the synchronous send (`__ssend`) in LWMPI, while Algorithm 6 presents the receive operation implementation (___recv). These routines can be found in `src/mputil/communication.c`, and it is important to note that since these operations use the IPC abstractions exposed by Nanvix, and are, consequently, OS-dependent, their implementation is made in `MPIUtil` instead of `LibMPI`, making the layers separation previously presented even more clear. Since we already discussed the implementation details of these two functions and their protocols in Section 5.7, we leave their algorithms here as an extra visualization, rather than detailing them again.

Algorithm 5 Synchronous send implementation.

Source: Developed by the author.

```
1: function ___SSEND(int cid, const void *buf, size_t size, int src, int dest
   mpi_process_t *dest_proc, int datatype, int tag)
2:     int remote;
3:     int remote_port;
4:     const char *remote_pname;
5:     struct comm_message message;
6:     struct comm_message reply;
7:
8:     # Retrieves target name and looks for its logical address.
9:     remote_pname ← PROCESS_NAME(dest_proc);
10:    remote ← NANVIX_NAME_ADDRESS_LOOKUP(remote_pname, &remote_port);
11:
12:    # Builds the request's that will be sent header.
13:    REQUEST_HEADER_BUILD(&message, cid, src, dest, datatype, size, tag);
14:
15:    # Is it a local communication?
16:    if (remote = local_node) then
17:        int bufferid = BUFFER_SLOT_RESERVE(buf, size);
18:
19:        # Sends reserved bufferid along with the request.
20:        message.msg.send.bufferid ← bufferid;
21:
22:        # Sends request to the receiver using the kernel abstractions.
23:        KMAILBOX_WRITE(outbox, &message, sizeof(struct comm_message));
24:
25:        # Waits in the reserved buffer slot for the receiver's signal.
26:        BUFFER_SLOT_WAIT(bufferid);
27:
28:        BUFFER_SLOT_RELEASE(bufferid);
29:
30:        return MPI_SUCCESS;
31:
32:    # Sends the comm. request to the target process.
33:    KMAILBOX_WRITE(outbox, &message, sizeof(struct comm_message));
34:
35:    # Receives confirmation message from the target process.
36:    NANVIX_MAILBOX_READ(inbox, &confirm, sizeof(struct comm_message));
37:
38:    # Sends data to the receiver using the high bandwidth channel.
39:    KPORTAL_WRITE(outportal, buf, size);
40:
41:    # Waits for the receiver's ACK message.
42:    NANVIX_MAILBOX_READ(inbox, &message, sizeof(struct comm_message));
43:
44:    return (MPI_SUCCESS);
```

Algorithm 6 Receive function implementation.
Source: Developed by the author.

```
1: function ___RECV(int cid, void *buf, size_t size, mpi_process_t *src, int datatype,
   struct comm_request *req)
2:     int remote_node;
3:     struct comm_message message;
4:     struct comm_message reply;
5:
6:     # Builds request to be matched with a received candidate.
7:     COMM_REQUEST_BUILD(req.cid, req.src, req.target, req.tag, &message.req);
8:
9:     # Receives a matching request in address of message.
10:    COMM_REQUEST_RECEIVE(&message);
11:
12:    # Is this a local communication?
13:    remote_node ← message.msg.send.nodenum;
14:    if (remote_node = local_node) then
15:        bufferid ← message.msg.send.bufferid;
16:
17:        # Directly copies the data from the local sender's buffer.
18:        BUFFER_SLOT_READ(bufferid, buf, size);
19:
20:        return (MPI_SUCCESS);
21:
22:    MUTEX_LOCK(&recv_lock);
23:
24:    # Builds confirmation message.
25:
26:    # Writes confirmation message to the sender.
27:    KMAILBOX_WRITE(outbox, &reply, sizeof(struct comm_message));
28:
29:    # Authorizes the remote peer to send the data using its portal.
30:    NANVIX_PORTAL_ALLOW(inportal, remote_node, message.msg.send.portal_port);
31:
32:    # Receives the data from the high bandwidth channel.
33:    NANVIX_PORTAL_READ(inportal, buf, req → received_size);
34:
35:    # Builds ACK message.
36:
37:    # Writes ACK message to the sender.
38:    KMAILBOX_WRITE(outbox, &reply, sizeof(struct comm_message));
39:
40:    MUTEX_UNLOCK(&recv_lock);
41:
42:    return (MPI_SUCCESS);
```