Enhancing Programmability in NoC-Based Lightweight Manycore Processors with a Portable MPI Library

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

[1]Distributed Systems Research Laboratory (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – Brazil

[2]Computer Architecture and Parallel Processing Team (CArT)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Brazil

[3]Laboratoire d'Informatique de Grenoble (LIG)
Université Grenoble Alpes (UGA) – France

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

Abstract. *The performance and energy efficiency provided by lightweight manycores is undeniable. However, the lack of rich and portable support for these processors makes software development challenging. To address this problem, 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 and evaluated it on the Kalray MPPA-256 processor. Results obtained with three applications from a representative benchmark suite unveiled that LWMPI achieves similar performance scalability in comparison with the low-level vendor-specific API narrowed for MPPA-256, while exposing a richer programming interface.*

1. Introduction

Lightweight manycore processors emerged to address demands on high-performance and energy efficiency [Francesquini et al. 2015]. On the one hand, to deliver high-performance and scalability, these processors rely on a distributed memory architecture and a rich Network-on-Chip (NoC). On the other hand, to achieve energy efficiency, they are built with simple low-power Multiple Instruction Multiple Data (MIMD) cores and Scratchpad Memories (SPMs) with no hardware coherency support. Moreover, they exploit heterogeneity by combining cores with different capabilities. Some industry-successful examples of lightweight manycores are the Kalray MPPA-256 [de Dinechin et al. 2013a], the Adapteva Epiphany [Olofsson 2016] and the Sunway SW26010 [Fu et al. 2016].

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 accesses across multiple physical address spaces. Hence, software should explicitly fetch data from remote 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 (OSes) [Kluge et al. 2014, Asmussen et al. 2016, Penna et al. 2019] and baremetal runtime systems [de Dinechin et al. 2013b, Varghese et al. 2014, Richie et al. 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 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 both approaches: a lightweight implementation of the Message Passing Interface (MPI) standard (named LWMPI) on top of Nanvix, a Portable Operating System Interface (POSIX)-compliant distributed OS that targets lightweight manycores [Penna et al. 2019]. LWMPI is compatible with the MPI specification and can be extended to support new features and other OSes with little effort.

To assess LWMPI with representative computing workloads, we carried out experiments with three applications extracted from the CAP Bench suite [Souza et al. 2017]. All experiments were executed on the Kalray MPPA-256 processor, a baremetal lightweight manycore. Our results unveiled that the proposed implementation delivers similar performance scalability when compared with a vendor-specific low-level Application Programming Interface (API) for the Kalray MPPA-256, while exposing a richer programming interface.

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 Blueprints

Lightweight manycores have an endeavour 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 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 example of such type of processor: the Kalray MPPA-256 [de Dinechin et al. 2013a]. Notwithstanding, the following discussion extends to other lightweight manycores [Olofsson 2016, Fu et al. 2016]. Figure 1a 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 Processing Elements (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 Dynamic Random Access Memory (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 Non-Uniform Memory Access (NUMA) design are built with multiple CPU packages interconnected by a dedicated hardware outside of the processor chips (e.g., NUMAlink); and
- Graphics Processing Units (GPUs) do not cope efficiently with MIMD workloads.

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 [de Dinechin et al. 2013a] and application deployment [Souza et al. 2017].

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 paragraphs, we examine each of them, and we state where our work is positioned.

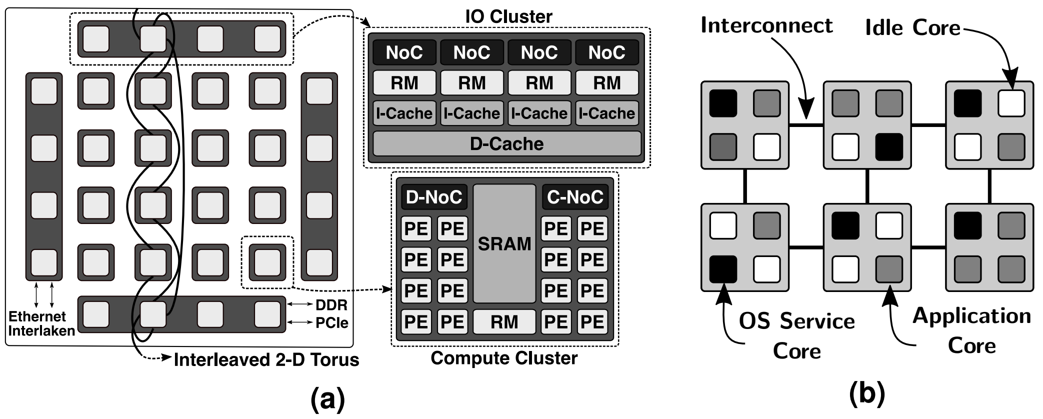


Figure 1. Kalray MPPA-256 (a) and distributed OS on a lightweight manycore (b).

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]. Figure 1b pictures a distributed OS running on a lightweight manycore. 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. Examples of such distributed OSes are Barrelfish [Baumann et al. 2009], FOS [Wentzlaff and Agarwal 2009], HeliOS [Nightingale et al. 2009], MOSSCA [Kluge et al. 2014], M^3 [Asmussen et al. 2016] and Nanvix [Penna et al. 2019].

In contrast to OSes, baremetal runtime systems aim at exposing a rich programming environment that is narrowed for the underlying architecture. They are provided on top of the hardware as libraries and are directly linked with applications. As an immediate consequence, baremetal runtime systems neither provide low-level resource sharing nor multiplexing. 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. Examples of such runtime systems are NodeOS [de Dinechin et al. 2013b], libasync [Hascoët et al. 2017], Epiphany SDK [Varghese et al. 2014] and CLETE [Richie et al. 2017].

Beyond the aforementioned approaches, we focus on a third alternative that we see as complementary to both: we rely on an OS to provide rich hardware management, sharing and multiplexing and we implement and deploy a high-level, industry-standard runtime system on top of this OS. Indeed, this approach is currently employed in multicore architectures, where parallel programming environments are provided on top of GNU/Linux, such as OpenMP and Cilk. However, in the context of our work, we highlight two high-level runtime systems that concern distributed programming and consequently are suitable for lightweight manycores. First, MPI is an industry-standard interface for message passing programming. It exposes two-sided communication functions for sending and receiving arbitrary-sized messages, either synchronously or asynchronously, while its communicator abstraction allows multiple logical communication flows within a distributed application. Besides, a one-sided communication version is available in more recent implementations of MPI. Second, Partitioned Global Address Space (PGAS) is a distributed programming environment that provides a global and shared address spaces over a distributed memory configuration. To this end, PGAS implementations rely on a logical partitioning of the address spaces of several processes and provide simple primitives for reading/writing/synchronizing data from/to these logical partitions.

3. LWMPI: A MPI Library for Lightweight Manycores

Aiming at better programmability in lightweight manycores, we propose the LWMPI: an MPI library for these processors. In contrast to alternative solutions, we made LWMPI portable across different architectures thanks to a design and implementation that relies on top of a POSIX-compliant distributed OS for lightweight manycores. In this section, we present and detail the internals of our solution. First, we discuss the goals that guided our design. Then, we uncover the architecture and implementation of LWMPI.

3.1. 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:

(i) *portability*: the library should be portable and applicable to various lightweight manycores;
(ii) *compatibility*: the implementation must comply with the MPI specification;
(iii) *extendability*: it should be possible to add new functions or submodules to the implementation with little effort; and
(iv) *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 that we believe to cope with the aforementioned challenges: (i) design our library on top of an OS to enable *portability* across different architectures; (ii) stick to the MPI standard to deliver *compatibility*; (iii) follow a tier-based 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 [SPI 2020] or MPICH [MPICH 2020] to keep our solution *light* and suitable for lightweight manycores [Ho et al. 2015]. We developed our library on top of Nanvix, a POSIX-compliant research OS that targets lightweight manycores [Penna et al. 2019]. To the best of our knowledge, Nanvix is the only open-source distributed OS that runs on commercially available baremetal lightweight manycores, like Kalray MPPA-256 [de Dinechin et al. 2013a] and OpTiMSoC [Wallentowitz et al. 2012].

3.2. LWMPI Architecture

Currently, LWMPI implements an initial subset of the MPI specification (version 3.1). We opted for this partial support since fully implementing the entire standard would result in a much bigger memory footprint, violating our fourth design goal, which is the *lightness* of the solution, so important to cope with the restrictions of lightweight manycores.

Figure 2 presents the two-tier approach adopted by LWMPI[1] on top of Nanvix. 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

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

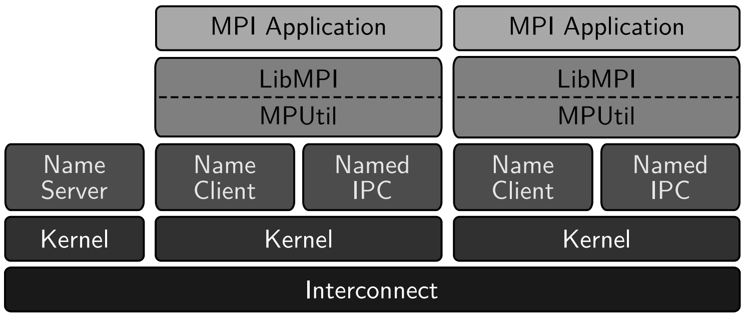


Figure 2. Architectural overview of LWMPI.

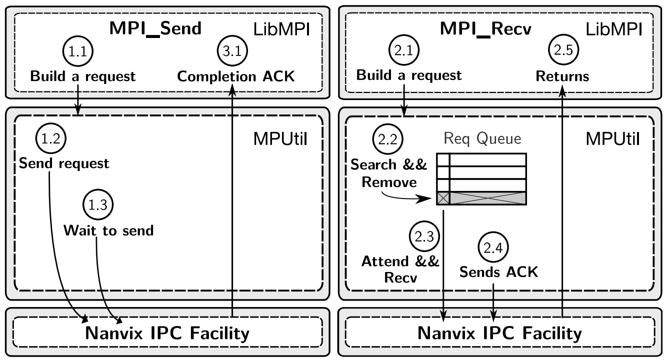
and implements the backend functions over the `MPUtil` tier. At this level, we focus on filtering the input parameters given by the user, performing the runtime management and correctly choosing the protocols employed by each MPI call in the underlying layer. In the current version, our library implements: functions for *runtime management*, such as `MPI_Init` and `MPI_Finalize`; support for *communicators* and information retrieving, such as `MPI_Comm_rank` and `MPI_Comm_size`; support for *groups* of communication with functions that are similar to those related with communicators; *error handlers*; and 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 `MPUtil` 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. `MPUtil` exposes elementary abstractions that support the top-level implementation of the MPI standard, aiming at 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, e.g., synchronous point-to-point sends and receives. To perform these protocols, `MPUtil` relies on the named Inter-Process Communication (IPC) abstractions exposed by the *Name Service* of the Nanvix runtime system, where processes names are translated into logical cluster identifiers. IPC abstractions of Nanvix include primitives for fine-grain fixed-size transfers (*mailbox*), coarse-grain fixed-size transfers (*portal*), and synchronization points (*sync*) [Souto et al. 2020].

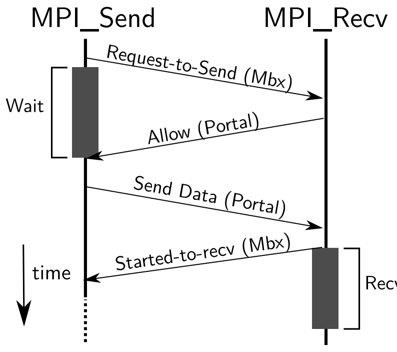
3.3. Point-to-Point Communication in LWMPI

Currently, LWMPI uses the *synchronous mode* to carry out communications in `MPI_Send` and `MPI_Recv` functions to avoid extra memory usage and keep the library thin (i.e., messages are not buffered). Figure 3a illustrates how the layers interact, showing `MPI_Send` (on the left) and `MPI_Recv` (on the right), while Figure 3b shows the inter-process interaction from the perspective of message exchanges.

As shown in Figure 3a, `LibMPI` is responsible for checking the input parameters and creating the communication requests (steps 1.1 and 2.1) that will be used by `MPUtil`. The requests include the information to be matched between `MPI_Send` and `MPI_Recv` (communicator, tag, and source/destination) and the user buffer employed to place/retrieve data by the IPC call, removing any intermediary buffering needs. After that, the sender submits its request to the receiver (step 1.2) via *mailbox*, sending a *request-to-send* message (Figure 3b), and blocks (step 1.3) until a matching `MPI_Recv` is posted and the receiver grants permission for data transfer. At the receiver side, `MPUtil` searches in



(a) Interactions between LWMPI and Nanvix.



(b) Communication protocol.

Figure 3. Implementation details of `MPI_Send` and `MPI_Recv` by LWMPI.

an internal queue (step 2.2) for a send request that matches its receive request built in step 2.1. If the queue is empty or no match occurred, the client waits for a matching request to arrive. Any other requests that arrive in the meantime are placed at the end of the queue.

When a matching request is found, the receiver consumes and attends it (step 2.3). At this point, the receiver issues permission to the portal of the sender, who wakes up and transfers the data to the user buffer. When the receiver starts to receive data, it sends an *ack* message to the sender via *mailbox* (step 2.4), 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 3.1). The receiver returns from `MPI_Recv` when it has read all the data from the channel, or have read the amount of data equivalent to the local buffer size (step 2.5).

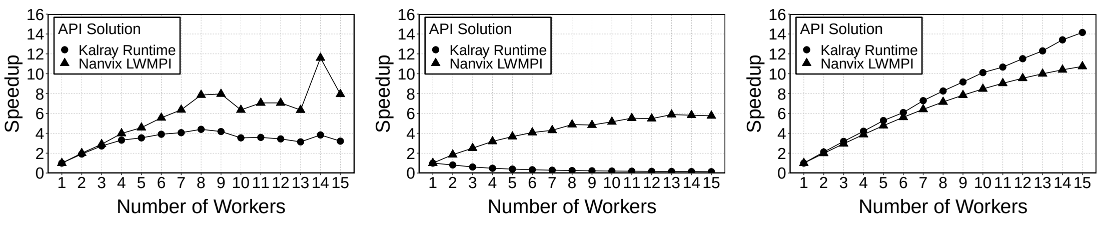
4. Evaluation Methodology

To deliver a comprehensive assessment of LWMPI, we relied on a subset of the CAP Bench suite [Souza et al. 2017], which is used to assess the performance of lightweight manycores. Applications in CAP Bench feature different parallel patterns, task types, communication intensity, and task loads, and they are developed in the C language, using one out of two environments: (i) OpenMP (for shared-memory manycores); and (ii) a vendor-specific baremetal API for the Kalray MPPA-256 processor. In this work, we employed the following applications in our analysis to exercise different characteristics.

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.



(a) FN Kernel **(b) GF Kernel** **(c) KM Kernel**

Figure 4. Experimental results.

We implemented these applications with MPI[2] and contrasted them with the original implementation of the benchmark for Kalray MPPA-256. Noteworthy, *a direct performance comparison between these two solutions is unfair*, since the original implementation relies on a vendor-specific baremetal runtime system that is narrowed for Kalray MPPA-256 but does not provide any means of software portability across architectures.

Applications in CAP Bench have a single *leader* process, which coordinates the execution, and may have several *workers*, which perform the computations. However, due to some current limitations in Nanvix, we could only deploy one process per Compute Cluster on Kalray MPPA-256, thus reaching a maximum of 16 processes in total. Overall, we carried out strong scaling experiments, where we varied the number of workers from 1 to 15 and we fixed the problem sizes of applications as follows: (i) numbers ranging from 1000001 to 1000129 for FN; (ii) 512×512 image and $7\times$ mask for GF; and (iii) 30720 points and 64 centroids for KM. We ran 30 trials of each configuration to ensure minimum variance in our results. The maximum coefficient of variance observed was below 1%.

5. Experimental Results

Figure 4a presents the speedup for the FN application, in which a leader process distributes equal-sized ranges of numbers to worker processes that compute the abundance values. Since FN is CPU-bound, communication has little interference and the results show a similar behavior in both solutions. We noticed an increase in speedup up to 8 workers. Thereafter, FN presents scalability issues, being the only exception with 14 workers. This general behavior is due to the problem design itself and the input workload. The leader process performs an integer division to compute the minimum amount of work to be sent to each worker. Then, the reminder is added to the last worker, which may result in load imbalance. This imbalance is very small up to 8 workers, but becomes substantial with more workers. With 14 workers, however, the workload is well balanced and the overall performance is improved. In general, these results show that our solution scaled well and was able to provide an easy adaptation of the kernel without introducing an overhead as the parallelism is increased.

Figure 4b pictures the speedup for the GF kernel, in which the leader process splits an image into equal-sized chunks and distributes them to worker processes that perform matrix computations using the Gaussian mask. As it can be noticed, LWMPI presented suboptimal scalability whereas the default runtime library did not scale at all. The small problem sizes may have resulted in insufficient workloads for the original benchmark implementation using the Kalray runtime, causing its performance to deteriorate. At the same time, for LWMPI this problem seems to be attenuated as the parallelism in-

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

creases, proving its scalability also in these situations. However, we believe that using asynchronous communications for both solutions would significantly reduce the bottleneck on the leader process and improve the overall performance.

Figure 4c shows the speedup for the KM kernel, in which the leader process iteratively orchestrates computing by gathering and broadcasting centroids to worker processes. As an immediate consequence, this application has a higher communication demand than the previous ones. This characteristic impacted the results, where LWMPI achieves lower speedups when compared to the Kalray runtime. This occurred because the baremetal runtime can fitly handle the irregular workload, while LWMPI is limited by the coarse-grained fixed-size messages of the *portal* abstraction in Nanvix, which is used to handle the data transfers in `MPUtil`. Thus, small problem sizes do not overcome the overhead imposed by this abstraction that is designed to fit dense data transfers. Nevertheless, this situation can be settled by a mechanism that dynamically chooses which communication abstraction fits better the data granularity to be sent. For instance, it would be possible to use the *mailbox* abstraction to send fine-grained messages and the *portal* abstraction for coarse-grained ones. As a result, we could transfer small messages with low latency and large messages with high bandwidth. We intend this to be an optimization in the future. Even so, both solutions had similar linear behaviors, showing that LWMPI was able to keep up with the speedup scalability presented by the Kalray runtime.

In general, LWMPI delivered a lightweight and richer programming interface, presenting good scalability for parallel and distributed problems. Consequently, we improve programmability and deliver implicit portability for lightweight manycores, which are our main contributions. However, the results lighted significant improvement cases, such as combining fine-grain and coarse-grain communications to deal with irregular communications and the use of asynchronous calls to overlap communication with computation.

6. Related Work

Software development for lightweight manycores is challenging because it strives in finding the balance between performance and programmability. In this context and specifically concerning communication, there are two approaches currently employed: (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 [van der Wijngaart et al. 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 [de 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 IMAPCAR2 [Kelly et al. 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 PGAS programming model, such as the Berkeley 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 and 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 et al. 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, even 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.

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.

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 believe that the proposed solution brings the best of the aforementioned approaches, since it offers a standard high performance solution that can be used in a broad range of lightweight manycores.

7. Conclusion

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 are 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. LWMPI is designed from scratch and follow a two-tier approach to separate and self-contain the MPI interface from the OS-dependent layer. Our experiments with applications from CAP Bench on the Kalray MPPA-256 processor unveil that LWMPI exposes a richer programming interface and achieves similar scalability in comparison with the low-level vendor-specific API narrowed for the Kalray MPPA-256 processor.

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. As future work, we intend to (i) improve the design and implementation of LWMPI; (ii) carry out experiments with other applications from CAP Bench; (iii) extend LWMPI to support a bigger subset of the MPI specification, e.g., collective communications and better protocols; and (iv) research and design a PGAS solution on top of Nanvix.

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

Asmussen, N. et al. (2016). M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 189–203, Atlanta, Georgia. ACM.

Baumann, A. et al. (2009). The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, Big Sky, Montana. ACM.

Boyd-Wickizer, S. et al. (2010). An analysis of linux scalability to many cores. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, pages 1–16, Vancouver, Canada.

Clauss, C. et al. (2011). Evaluation and improvements of programming models for the Intel SCC many-core processor. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 525–532. IEEE.

de Dinechin, B. D. et al. (2013a). A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In *IEEE High Performance Extreme Computing Conference*, HPEC '13, pages 1–6, Waltham, USA. IEEE.

de Dinechin, B. D. et al. (2013b). A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 18(International Conference on Computational Science):1654–1663.

Francesquini, E. et al. (2015). On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms. *Journal of Parallel and Distributed Computing (JPDC)*, 76(C):32–48.

Fu, H. et al. (2016). The Sunway TaihuLight Supercomputer: System and Applications. *Science China Information Sciences*, 59(7):072001–0720016.

Gamell, M. et al. (2012). Exploring cross-layer power management for PGAS applications on the SCC platform. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, page 235, New York, USA. ACM Press.

Haghbayan, M.-H. et al. (2017). Performance/reliability-aware resource management for many-cores in dark silicon era. *IEEE Transactions on Computers (TC)*, 66(9):1599–1612.

Hascoët, J. et al. (2017). Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor. In *Symposium on Embedded Systems for Real-Time Multimedia*, ESTIMedia '17, pages 51–60, Seoul. ACM Press.

Ho, M. Q. et al. (2015). MPI communication on MPPA many-core NoC: Design, modeling and performance issues. In *International Conference on Parallel Computing*, volume 27 of *ParCo '2015*, pages 113–122, Edinburgh, UK. IOS Press.

Kelly, B. et al. (2013). Autopilot: Message passing parallel programming for a cache incoherent embedded manycore processor. In *International Workshop on Many-Core Embedded Systems*, MES '13, page 62–65, New York, NY, USA. Association for Computing Machinery.

Kluge, F. et al. (2014). An Operating System for Safety-Critical Applications on Manycore Processors. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '14, pages 238–245, Reno, Nevada. IEEE.

MPICH (2020). Mpich: High-performance portable mpi.

Nightingale, E. B. et al. (2009). Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 221–234, Big Sky, Montana. ACM Press.

Olofsson, A. (2016). Epiphany-v: A 1024 processor 64-bit risc system-on-chip. *ArXiv*, 1610.01832:1–15.

Penna, P. H. et al. (2019). On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In *Brazilian Symposium on Computing Systems Engineering*, SBESC '19, pages 1–8, Natal, Brazil.

Richie, D. et al. (2017). A Distributed Shared Memory Model and C++ Templated Meta-Programming Interface for the Epiphany RISC Array Processor. *Procedia Computer Science*, 108:1093–1102.

Ross, J. and Richie, D. (2016). Implementing openshmem for the adapteva epiphany risc array processor. *Procedia Computer Science*, 80(C):2353–2356.

Serres, O. et al. (2011). Experiences with UPC on TILE-64 processor. In *Aerospace Conference*, pages 1–9. IEEE.

Souto, J. V. et al. (2020). Mecanismos de comunicação entre clusters para lightweight manycores no nanvix os. In *Escola Regional de Alto Desempenho da Região Sul*, ERAD/RS '20, pages 1–4, Porto Alegre, RS, Brasil. SBC.

Souza, M. et al. (2017). Cap bench: A benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computation: Practice and Experience (CCPE)*, 29(4):1–18.

SPI (2020). Open mpi: Open source high performance computing.

van der Wijngaart, R. F. et al. (2011). Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Operating Systems Review (OSR)*, 45(1):73–83.

Varghese, A. et al. (2014). Programming the adapteva epiphany 64-core network-on-chip coprocessor. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IPDPSW '14, pages 984–992, Phoenix, USA. IEEE.

Wallentowitz, S. et al. (2012). A Framework for Open Tiled Manycore System-On-Chip. In *International Conference on Field Programmable Logic and Applications*, FPL '2012, pages 535–538, Oslo. IEEE.

Wentzlaff, D. and Agarwal, A. (2009). Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85.