A performance evaluation using an MPI library over the communication infrastructure of Nanvix

João Fellipe Uller
Universidade Federal de Santa Catarina (UFSC)
Florianópolis, Brazil
joao.f.uller@grad.ufsc.br

***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. In this work, we propose a portable and lightweight MPI library (LWMPI) designed from scratch to cope with the 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 a low-level vendor-specific API narrowed for MPPA-256, while exposing a richer programming interface.**

***Index Terms*—lightweight manycores, runtime systems, MPI, high-performance computing**

I. Introduction

Lightweight manycore processors emerged to address demands on high-performance and energy efficiency [1]. 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 [2] and the Adapteva Epiphany [3].

However, while the aforementioned architectural features make lightweight manycores so interesting, they also introduce several challenges in software programmability. For instance, the *distributed memory architecture* requires a non-trivial software design where the data should be explicitly fetched from remote memories to local ones to be manipulated [1]. 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 [4]. Finally, the rich NoC exposes mechanisms for asynchronous programming to overlap communication with computation [5]; and hand-operated routing to guarantee uniform communication latencies.

Currently, two approaches are employed to address programmability challenges in lightweight manycores: Operating Systems (OSes) [6]–[8] and baremetal runtime systems [9]–[11]. 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, 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, resulting in non-portable applications.

In this work, we present a third proposition, trying to address the programmability and portability challenges of 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 [8]. To assess LWMPI with representative computing workloads, we carried out experiments with three applications extracted from the CAP Bench suite [4]. All experiments were executed on the Kalray MPPA-256, a baremetal lightweight manycore. Our results unveiled that LWMPI 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 II we discuss related works. In Section III, we present an architectural overview of LWMPI. In Section IV, we detail our evaluation methodology and present the Kalray MPPA-256. In Section V, we discuss the experimental results. Finally, in Section VI, we draw our conclusions.

II. 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 deliver performance. For instance, synchronous [12] and asynchronous [13] interfaces are provided on top of Message Passing Buffer (MPB) for the Intel Single-Cloud Computer. The Kalray MPPA-256 features both a communication library that shares some similarity with POSIX [9] and a specific interface for one-sided communications [5]. Finally, a specific communication API is provided for the Adapteva Epiphany processor [10].

In contrast, standard communication interfaces benefit from extensive improvements and optimizations, being 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, 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 [14] and the OpenSHMEM implementation [15] for the Adapteva Epiphany processor. Moreover, there have been some efforts on providing an MPI port for Kalray MPPA-256 [16] and Adapteva Epiphany [11]. The former is the closest work to the present one, also presenting an implementation from scratch to cope with the restrictions of lightweight manycores. The main difference, however, is the fact that it is implemented on top of a vendor-specific Inter-Process Communication (IPC) library, and so, being not portable to other processors/architectures. The latter, in addition, does not conform with the MPI standard.

Overall, both approaches lack in application portability. 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), offering a standard high performance solution applyable to a broad range of lightweight manycores.

III. LWMPI: Lightweight MPI for Manycores

Aiming at better programmability in lightweight manycores, we propose LWMPI[1]: an MPI library for these processors. In contrast to alternative solutions, we made LWMPI portable across different architectures by relying it on top of a POSIX-compliant distributed OS for lightweight manycores.

These processors bring several challenges to software development, thereby making easy-to-use interfaces an important requirement. 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; we enable this by designing it on top of a POSIX-compliant OS (Nanvix);
(ii) *compatibility*: the implementation must comply with the MPI specification; it sticks to the 3.1 version;
(iii) *extendability*: it should be possible to add new functions or submodules to the implementation with little effort; as so, we design it in a tier-based scheme; and
(iv) *lightness*: the implementation should be simple and lightweight to cope with restrictive resources of lightweight manycores; this way, we implemented it from scratch, rather than adapting an existing heavy-weight solution like OpenMPI[2] or MPICH[3]

A. 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 (*lightness*). Figure 1 presents the two-tier approach adopted by LWMPI on top of Nanvix.

[1]LWMPI is available at: https://github.com/nanvix/libmpi
[2]OpenMPI website: https://www.open-mpi.org
[3]MPICH website: https://www.mpich.org

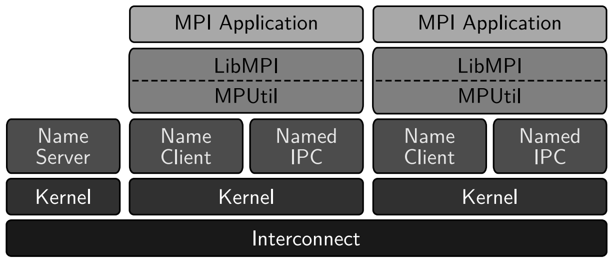

Figure 1: Architectural overview of LWMPI.

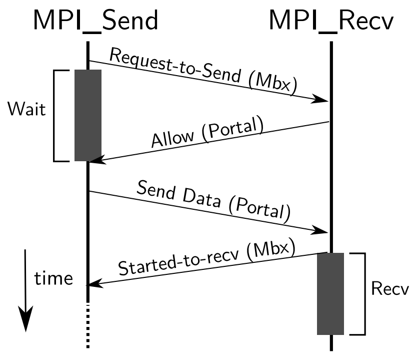

Figure 2: Communication protocol.

The `LibMPI` tier encapsulates the top-level library and is the entry point for user applications. This layer exposes the library interface and implements the backend functions over `MPUtil`. It focuses on filtering the input parameters given by the user, performing the runtime management and choosing the underlying protocols employed by the MPI calls. In its 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 similarly to communicators; *error handlers*; and point-to-point communication via `MPI_Send` and `MPI_Recv` in 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 tier, aiming at keeping the library implementation decoupled from the OS interface. It is also in this level where the communication protocols used in the MPI calls are implemented. To perform these protocols, `MPUtil` relies on the named IPC abstractions exposed by the *Name Service* of the Nanvix runtime system, which include primitives for fine-grain fixed-size transfers (*mailbox*), coarse-grain fixed-size transfers (*portal*), and synchronization points (*sync*) [17].

B. 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 2 shows the inter-process interaction from the perspective of message exchanges.

Initially, when a MPI_Send call is issued, the sender builds a request and transfers it to the target process through *mailbox*, waiting for the receiver to establish the communication. The transferred header includes all the information needed by the receiver to match a correspondent MPI_Recv with the registered request, i.e., communicator id, tag and source/destination. At the receiver side, when a MPI_Recv call is issued, it looks for incoming requests that arrived or blocks waiting for a matching one, if none has been found.

When a matching request is found, the receiver emits an allow signal to the output *portal* of the sender, unblocking it and giving permission to start the data transfer. The sender, then, starts to send the data using the high bandwidth channel. When the receiver starts to receive the data in its input *portal*, it issues a second started-to-receive signal to the sender, signaling that the sender can successfully return when it transmitted all the data through the channel, or if it has already done that. The receiver will return from MPI_Recv when it has read all the data from the channel, or have read the amount of data equivalent to the local user buffer size.

IV. Evaluation Methodology

To deliver a comprehensive assessment of LWMPI, we relied on a subset of the CAP Bench suite [4]. Applications in CAP Bench are developed in the C language, and feature different parallel patterns, task types, communication intensity, and task loads. The applications employed in our analysis are:

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.

We implemented these applications with MPI[4] 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 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 that coordinates the execution, and may have several *workers* that perform the computations. Overall, we carried out strong scaling experiments, where we varied the number of workers

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

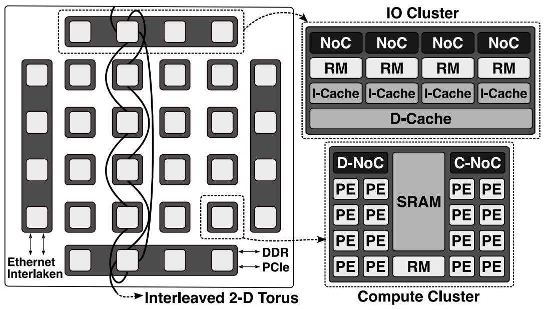

Figure 3: Kalray MPPA-256 architectural overview.

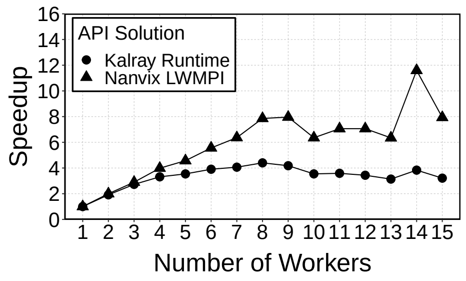

Figure 4: FN Kernel

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, and the maximum coefficient of variance observed was below 1%.

A. Experimental Platform

The Kalray MPPA-256 is an example of an industry-successful lightweight manycore, and Figure 3 presents its architectural blueprint. Overall, it integrates 288 cores disposed into 20 clusters. Each cluster is composed of heterogeneous 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. 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. Kalray MPPA-256 also features a built-in Direct Memory Access (DMA) engine in its NoC interfaces to enable asynchronous communications and higher bandwidth for dense data transfers.

V. Experimental Results

Figure 4 presents the speedup for the FN application. Since FN is CPU-bound, communication has little interference and the results show a similar behavior in both solutions: an increase in speedup up to 8 workers and scalability issues, thereafter, being the only exception with 14 workers. This 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, the results show that LWMPI scaled well and was able to provide an easy adaptation of the kernel without introducing an overhead as the parallelism is increased.

Figure 5 pictures the speedup for the GF kernel. 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, deteriorating the performance of the Kalray runtime. At the same time, for LWMPI this problem seems to be attenuated as the parallelism increases, proving its scalability also in these situations. We believe that using asynchronous communications for both solutions would significantly reduce the bottleneck on the leader process and improve the overall performance.

Figure 6 shows the speedup for the KM kernel. This application has higher communication demands than the previous ones, which 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. Thus, small problem sizes do not overcome the overhead imposed by this abstraction, designed to fit dense data transfers. Nevertheless, this situation can be settled by a mechanism that dynamically chooses which IPC abstraction fits better the data granularity to be sent. 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. 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.

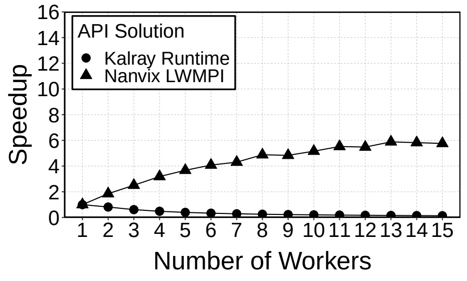

Figure 5: GF Kernel

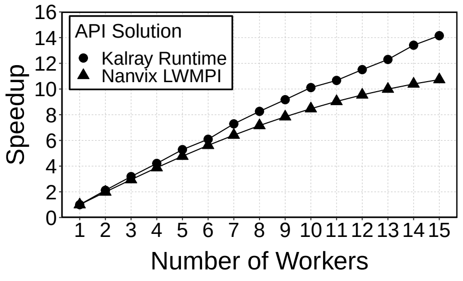

Figure 6: KM Kernel

VI. 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.

Acknowledgements

This paper is a simplified version of the article entitled ”Enhancing Programmability in NoC-Based Lightweight Manycore Processors with a Portable MPI Library”, submitted and accepted at the XXI Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2020), which included contributions of João Vicente Souto (UFSC), Pedro Henrique Penna (PUC-Minas / UGA - France), Márcio Castro (UFSC), Henrique Freitas (PUC-Minas) and Jean-François Méhaut (UGA - France). Since this version was specifically tailored to be presented in the discipline ”INE410129-41000025DO/ME (2020-1)”, their names were ommited at the beggining, but stay registered here the acknowledgments for their contributions.

References

[1] E. Francesquini, M. Castro, P. H. Penna, F. Dupros, H. Freitas, P. Navaux, and J.-F. Méhaut, “On the Energy Efficiency and Performance of Irregular Application Executions on Multicore, NUMA and Manycore Platforms,” *Journal of Parallel and Distributed Computing (JPDC)*, vol. 76, no. C, pp. 32–48, february 2015.

[2] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, “A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications,” in *IEEE High Performance Extreme Computing Conf.*, ser. HPEC ‘13. Waltham, USA: IEEE, sep 2013, pp. 1–6.

[3] A. Olofsson, “Epiphany-v: A 1024 processor 64-bit risc system-on-chip,” *ArXiv*, vol. 1610.01832, pp. 1–15, 2016.

[4] M. Souza, P. H. Penna, M. Queiroz, A. Pereira, L. F. Góes, H. Freitas, M. Castro, P. Navaux, and J.-F. Méhaut, “Cap bench: A benchmark suite for performance and energy evaluation of low-power many-core processors,” *Concurrency and Computation: Practice and Experience (CCPE)*, vol. 29, no. 4, pp. 1–18, february 2017.

[5] J. Hascoët, B. D. de Dinechin, P. G. de Massas, and M. Q. Ho, "Asynchronous One-Sided Communications and Synchronizations for a Clustered Manycore Processor," in *Symp. on Embedded Systems for Real-Time Multimedia*, ser. ESTIMedia '17. Seoul: ACM Press, oct 2017, pp. 51–60.

[6] F. Kluge, M. Gerdes, and T. Ungerer, "An Operating System for Safety-Critical Applications on Manycore Processors," in *Intl. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing*, ser. ISORC '14. Reno, Nevada: IEEE, sep 2014, pp. 238–245.

[7] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis, "M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. Atlanta, Georgia: ACM, mar 2016, pp. 189–203.

[8] P. H. Penna, J. Souto, D. F. Lima, M. Castro, F. Broquedis, H. Cota de Freitas, and J.-F. Méhaut, "On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores," in *Brazilian Symp. on Computing Systems Engineering*, ser. SBESC '19, Natal, Brazil, november 2019, pp. 1–8.

[9] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, no. Intl. Conf. on Computational Science, pp. 1654–1663, jan 2013.

[10] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the adapteva epiphany 64-core network-on-chip coprocessor," in *Intl. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, ser. IPDPSW '14. Phoenix, USA: IEEE, 2014, pp. 984–992.

[11] D. Richie, J. Ross, and J. Infantolino, "A Distributed Shared Memory Model and C++ Templated Meta-Programming Interface for the Epiphany RISC Array Processor," *Procedia Computer Science*, vol. 108, pp. 1093–1102, jan 2017.

[12] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on intel's single-chip cloud computer processor," *SIGOPS Operating Systems Review (OSR)*, vol. 45, no. 1, p. 73–83, feb 2011.

[13] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *Intl. Conf. on High Performance Computing & Simulation (HPCS)*. IEEE, jul 2011, pp. 525–532.

[14] M. Gamell, I. Rodero, M. Parashar, and R. Muralidhar, "Exploring cross-layer power management for PGAS applications on the SCC platform," in *Intl. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*. New York, USA: ACM Press, 2012, p. 235.

[15] J. Ross and D. Richie, "Implementing openshmem for the adapteva epiphany risc array processor," *Procedia Computer Science*, vol. 80, no. C, pp. 2353–2356, jan 2016.

[16] M. Q. Ho, B. Tourancheau, C. Obrecht, B. D. de Dinechin, and J. Reybert, "MPI communication on MPPA many-core NoC: Design, modeling and performance issues," in *Intl. Conf. on Parallel Computing*, ser. ParCo '2015, vol. 27. Edinburgh, UK: IOS Press, 2015, pp. 113–122.

[17] J. V. Souto, P. H. Penna, M. Castro, and H. Freitas, "Mecanismos de comunicação entre clusters para lightweight manycores no nanvix os," in *Escola Regional de Alto Desempenho da Região Sul*, ser. ERAD/RS '20. Porto Alegre, RS, Brasil: SBC, 2020, pp. 1–4.