To Share or Not to Share: a case for MPI in Shared-Memory

Julien Adam, Jean-Baptiste Besnard, Adrien Roussel, Julien Jaeger, Patrick Carribault, Marc Pérache

To Share or Not to Share: a case for MPI in Shared-Memory

Julien Adam[1], Jean-Baptiste Besnard[1], Adrien Roussel[2,3], Julien Jaeger[2,3], Patrick Carribault[2,3], and Marc Pérache[2,3]

[1] ParaTools SAS, France
[2] CEA, DAM, DIF, F91297 Arpajon, France
[3] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France

Abstract. The evolution of parallel computing architectures presents new challenges for developing efficient parallelized codes. The emergence of heterogeneous systems has given rise to multiple programming models, each requiring careful adaptation to maximize performance. In this context, we propose reevaluating memory layout designs for computational tasks within larger nodes by comparing various architectures. To gain insight into the performance discrepancies between shared memory and shared-address space settings, we systematically measure the bandwidth between cores and sockets using different methodologies. Our findings reveal significant differences in performance, suggesting that MPI running inside UNIX processes may not fully utilize its intranode bandwidth potential. In light of our work in the MPC thread-based MPI runtime, which can leverage shared memory to achieve higher performance due to its optimized layout, we advocate for enabling the use of shared memory within the MPI standard.

Keywords: MPI · NUMA · Memory · Thread · Programming Models.

1 Introduction

The parallel programming landscape has undergone significant changes in recent decades, posing challenges for developers working on long-standing applications. Initially relying solely on Message Passing Interface (MPI) for distributed memory parallelism, the evolution of hardware and increasing node complexity has necessitated the adoption of hybrid MPI+X models such as OpenMP for shared memory parallelism since the early 2000s. This trend is likely to persist through the development of more powerful nodes while memory per core decreases.

Moreover, the emergence of converged or topologically-sensitive hardware, characterized by Non-Uniform Memory Access (NUMA), has added another layer of complexity to parallel programming. Writing efficient codes in such environments is crucial for minimizing data movement across the memory hierarchy. This requirement often necessitates the use of multiple programming models,

including distributed and shared memory paradigms, as well as a thorough understanding of the hardware.

While these concepts are well-documented in literature, navigating this complex landscape with existing code while preserving performance remains an open research question. In this paper, we pose the question of the role of MPI in managing data locality. It is clear that, in its default form, MPI is NUMA-aware, as it creates separate processes with clear bindings. However, as we will further elaborate in the rest of this paper, communication between Unix processes is suboptimal in shared-memory configurations. After measuring this difference on recent hardware, we argue that having shared-memory communication primitives in MPI could be beneficial given the increasing size of nodes and their heterogeneity — requiring careful computing spatialization.

2 Background and Motivation

In this paper, we focus specifically on shared memory programming with increasingly large nodes that require hybridization. We first recall how programming interfaces are evolving to accommodate multiple models in a single application. Then, we discuss the contribution of our research; questioning the position of MPI in this model nesting and suggesting it could also apply to shared-memory systems. Furthermore, we touch upon existing methods for moving data inside and between processes on a single node.

2.1 On the Evolution of Programming Interfaces

When examining legacy applications with respect to parallel computing using shared memory, it has become increasingly specialized over time to reveal hierarchical parallelism. With GPUs emerging as the preferred choice for energy-efficient computing, an additional layer has been introduced to the hardware stack. As a result, vendor-specific interfaces are commonly employed due to their extensive support and efficiency compared to portable alternatives such as OpenMP Targets. This inclination towards higher-level, vendor-supported interfaces supports the development of abstraction layers like Kokkos in C++ [28], aiming to conceal the underlying differences to ease adoption and efficient parallelization.

MPI has long been the go-to choice for high-performance computing applications due to its widespread adoption. However, in the realm of machine learning, vendors have adopted smaller subsets of MPI into their vendor-specific libraries such as NCCL, RCCL, and oneCCL [32]. This shift can be attributed to several reasons. Firstly, the complexity of MPI and its lack of native support for GPUs–essential components in ML–could be a significant factor. The expectations of end-users for drop-in implementations may have further motivated vendors to develop customized solutions better suited to their hardware and topologies [5]. Abstracting these optimizations into standard interfaces is challenging in the face of industry competition.

Back to the context of HPC, MPI remains the de facto standard for distributing computations. Its stability and rigorous standardization make MPI reliable for long-running production applications historically used in high-performance computing landscapes. However, we argue that MPI requires significant evolutions to maintain its position as the preferred embedding model for modern HPC workloads. We refer to MPI as the *embedding model* because when a parallel program is initiated, it begins with UNIX processes operating under the batch manager, which are subsequently assigned ranks to establish a logical mapping of processes (a specific rank within `MPI_COMM_WORLD`). MPI plays a pivotal role in defining how jobs are structured and executed, while applications are constructed around its abstraction. Moreover, programs can further subdivide each logical unit using in-memory parallel programming models like OpenMP. In cases where necessary, they can even incorporate GPU code through the utilization of optimized BLAS libraries or by transferring part of the code to GPUs using frameworks such as CUDA, ROCm, or oneAPI.

2.2 Means of Moving and Sharing Memory in Linux

The long history of MPI has been instrumental in driving the development and exploration of not just data movement over high-performance networks, but also within nodes, i.e., between Unix processes. As such, at the node level, the MPI runtime provides a means to move small messages using shared memory segments (SHM) [4]. These segments are structured as message queues between MPI processes, with the same pages projected into the address space of multiple processes. When an MPI process sends or receives a message, it first copies the data into the shared-memory segment before copying from the segment and placing it in the final buffer. This necessarily leads to two separate copies of the same data: one in the segment, another from the segment. It should be noted that MPI's main messaging interface, designed for distributed memory, always involves copying message data from one buffer to another, leading to duplication of information.

Despite shared memory being the preferred approach for small messages due to its efficiency, larger messages require a different method. The projected memory region has a fixed size, thus fragmentation is necessary if SHM is used, leading to unnecessary copies and extra overhead. Alternative approaches include using DMA-capable network cards to perform local DMA for copy operations. If these methods are not feasible, the Linux kernel can be used as a trusted third-party to carry out memory copies across process address spaces. These interfaces, such as Cross-Memory Attach (CMA) [31], which is standard in Linux and performs cross-address space copies, take an IOVEC and a process ID to perform the copy. Similar interfaces, including KNEM [10], LiMIC [16], or more recently XPMEM [30], with improved performance or support for DMA engines such as I/O AT were also developed as kernel modules due to their need to securely bridge process address spaces.

2.3 Contribution

In the face of escalating demands for parallel programming models and divergent development approaches, it is imperative to recognize the necessity of advanced abstraction layers to ensure feasible and manageable program development. By abstracting away complexity, we can maintain productive development processes despite intricate underlying architectures. However, existing methods employing nested abstractions present significant challenges in both programming and performance optimization.

To substantiate this argument, we conduct a series of experiments using openly available codes[4] to evaluate the performance disparities between shared-memory and shared-address spaces on contemporary hardware platforms such as the Nvidia Grace Superchip and Intel Sapphire Rapids. Following earlier research by Hori et al. [13], we also conclude that for communication-intensive and/or memory-bound applications, MPI executed within UNIX processes is unable to fully tap into its potential on these devices – requiring a single process per node to take advantage of full bandwidth capabilities.

Our intention is not to propose a definitive solution to this predicament. Instead, we draw attention to various remedies that have been proposed in the literature (including our past work) and discuss potential enhancements to MPI, such as enhanced awareness of shared-memory execution, which could better exploit its capabilities.

3 Performance Measurement Approach

In this section, we outline our approach for evaluating memory transfer performance, focusing on empirical methods. Our tests aim to explore the bandwidth between different locations. Measuring latency typically involves exchanging small messages and calculating the round-trip time. Alternatively, bandwidth can be estimated by determining the duration required for a message of known size to transfer. However, in practice, gauging bandwidth proves simpler than assessing latency due to its role as a bandwidth penalty during transfer initiation. To bypass these costs, we opt for large data transfers (exceeding 1 MB) in our experiments. The bandwidth results are then averaged from N transfer trials to eliminate the impact of initialization. Based on the proposed practical data transfer model, our study aims to investigate the effectiveness of various data transfer methods across several representative architectures, such as NVIDIA Grace Superchips and Intel Sapphire Rapids. This allows us to evaluate the performance of inter-core connectivity and the quality of cross-chip interconnects. To achieve this goal, we ask the question: how can a program optimally utilize this interconnect and bandwidth using standard system interfaces, including MPI? To answer this question, we conducted measurements to determine both sequential and parallel core-to-core bandwidth within and between sockets.

[4] `https://github.com/besnardjb/memmapper`

The measurements we conducted enable us to assess the cost difference between a shared-memory approach (using memcpy) and a shared-address space method utilizing the Cross-Memory Attach interface on our selected architectures. Although interfaces such as XPMEM [30,13] may offer better performance, they were not available in our test environment. However, these interfaces still require a kernel mitigation to project the remote address space into the local one, which comes with its own cost and necessitates caching by MPI for optimal efficiency. Once implemented, this can be viewed as an on-demand shared memory segment, ensuring high performance. During this process, we also examine the data-movement performance of MPI, which is the de facto standard for moving data between processes in HPC. Hybrid approaches such as MPI+OpenMP still leverage the same fundamental data transfer technologies relying on methods like memcpies for exchanging information and the reason we chose to focus on pure MPI.

The idea behind these measures is to contextualize how well data can be moved across various architectures. Of course, there are other performance parameters, such as latency and operations per cycle. We do not purport to cover all aspects of the performance spectrum. In fact, some applications may not be sensitive to bandwidth and might instead be more computing-bound. However, in the case of MPI, which is focused on moving data around, our goal is to identify a measurable parameter (in this case, bandwidth), project it onto multiple architectures, outline overall performance improvements, and demonstrate the ability of interfaces to harvest maximum performance.

3.1 Sequential Shared-Memory Core-to-Core Bandwidth

When considering two cores exchanging data in point-to-point communication, what is the achievable bandwidth? To measure this, we set up a system where data segments are mapped to specific cores with proper affinity. We then copy that data to a buffer local to another processing unit while repeating these steps for the entire system. This measures the sequential bandwidth (as only one transfer occurs at a time) between cores, which is a common and basic measurement in many computer systems used as a baseline to evaluate the performance of the memory subsystems. We have developed a simple `memmapper`[5] tool, which can carry out this procedure with configurable sizes.

Looking at Figure 1, it is noticeable that NUMA effects have a significant impact on memory bandwidths between cores. As nodes become larger, the memory hierarchy also becomes more complex. For example, looking at the figure depicting 1MB transfers on the Grace Superchip (top left), one can see how the performance differs without considering cache effects when the data size increases to 100MB (bottom left). As expected, this analysis reveals that spatializing processing correctly can significantly enhance data transfer rates and reduce bottlenecks when performing concurrent transfers between cores. However, the question that arises from these measurements is whether this behavior

[5] `https://github.com/besnardjb/memmapper`

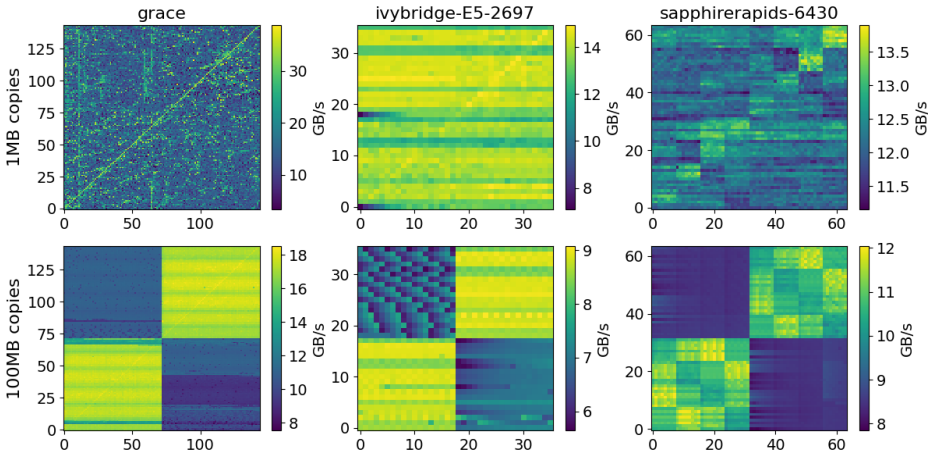

Fig. 1: Outline of NUMA effects on various architectures and transfer sizes. Coordinates `(x, y)` denotes data transfers from core `x` to core `y`. Color represents averaged bandwidth over 1000 transfers.

remains consistent when conducting several parallel transfers, or if there are any bottlenecks. To this purpose, in the next section, we will compare both CMA and memcpy when performing parallel data-transfers, which are more representative of HPC payloads.

3.2 Parallel Shared-Memory Core-to-Core Bandwidth

We repeated these measurements by creating data-transfer pairs between cores located in the same socket and across different sockets. To achieve this, we prepared buffers and synchronously initiated the transfers. The measured bandwidth is the average of the bandwidths over the entire duration of each transfer, not an average of individual bandwidths. This approach accounts for temporal scattering of data transfers, as it is virtually impossible to ensure that all transfers start simultaneously and complete at the same time.

In Figure 2, we present results for N concurrent data transfers within a given socket and between sockets of a particular architecture for both 1MB and 200 MB message sizes. These measurements were obtained using memory reuse, which involves averaging the bandwidth of 100 transfers of the same buffer after it has been bound to local affinity with a `memset` operation. Consequently, these measurements are susceptible to cache effects since the same data is copied multiple times. For instance, in Figure 2 bottom left, when transferring between sockets, the total bandwidth reaches up to 1300MB/sec on Grace Superchip, which surpasses the theoretical peak of NVLink interconnect (900GB/sec) due to caching effects. However, this effect diminishes at larger sizes because cache effects decrease.

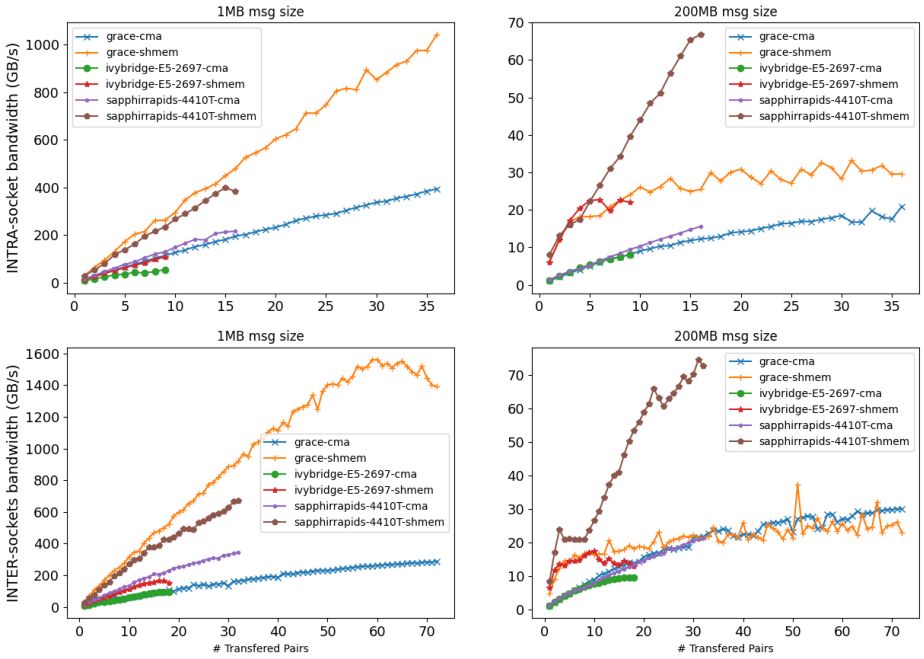

Fig. 2: Total bandwidth for concurrent transfers both inside and between sockets on various architectures using either Cross-Memory Attach or regular `memcpy` (shmem). All measurements are averaged 100 times.

To clarify, until now, we have discussed how shared memory bandwidth has evolved within a single node in the context of a single address space. However, it is essential to note that MPI is designed to run on distributed memory systems, which means multiple processes need to exchange data efficiently both within a node and between nodes. While high-speed interconnects can be used for communication between nodes, intranode communications require different methods. For smaller messages, most MPI implementations employ shared memory segments, where data is copied into the segment and subsequently transferred to the destination buffer. However, this approach may not be efficient for larger messages due to potential fragmentation. As a result, for larger messages MPI primarily uses Cross-Memory Attach or its more advanced implementation KNEM for interprocess communication across different address spaces.

Looking at Figure 2, a comparison between shared-memory and CMA bandwidth reveals that the performance difference increases with the number of involved cores. This leads to significant performance differences, as shown in Table 1. We observe that Intel Sapphire and Nvidia Grace Superchip exhibit different responses. Overall, shared-memory improves performance by a factor of 2.63 on average compared to CMA. However, with larger copies, the transfer performance of Grace is significantly decreased when compared to messages fitting in

	Nvidia GRACE		Intel SAPPHIRE	
Size	Intra	Inter	Intra	Inter
1MB	2.64	4.81	1.77	1.94
200MB	1.41	0.76	4.28	3.43

Table 1: Performance ratio between Shared-Memory and CMA for the largest number of concurrent copies, considering configurations of Figure 2.

caches, making CMA faster than `memcpy` in such cases. This is a singular result that we consider linked to saturation on the NVLINK, leading to performance improvements due to the extra overhead of CMAs alleviating contention. This hypothesis is supported by the noisier response of the system (despite the 100 averages), as evident from Figure 2 bottom right.

Note that this performance difference can be mitigated with more advanced shared-address space mechanisms, such as XPMEM. However, we did not have access to systems of the given architecture with this module installed. Nonetheless, despite the ability to bypass the kernel in the copies once the pages are projected, there is still a need to set up segments (and thus implement a rendezvous protocol and necessary caching). This necessarily creates overhead for teardown and setup. As such, even though the bandwidth penalty would be reduced on a projected segment, the projection cost would remain, particularly for applications that frequently change buffers over time.

4 Adapting MPI to Shared-Memory

Given the observed performance difference between CMA and a shared-memory copy, we must consider the implications for MPI. MPI is designed for distributed memory systems, which run on multiple nodes; however, its support for shared memory might be limited due to its design. In this section, we will discuss potential solutions to address this situation. We start by recalling current shared-memory support in MPI. Then, we discuss how the standard may progressively incorporate support for shared-memory. Lastly, we recall alternative methods for porting MPI in shared-memory including process virtualization.

4.1 Current and Past Support for MPI in Shared-Memory

Although MPI has made significant strides forward with the introduction of version 4.0, support for shared-memory has remained relatively stagnant since MPI 3.0. One notable exception to this trend is the addition of MPI 3.0 shared-memory windows, which have enabled a shared address space to be projected between MPI processes running on the same node. This window mechanism represents an important step towards harnessing the performance benefits of shared memory within the context of the MPI paradigm. By enabling in-memory communication and collaboration between processes, these shared-memory windows

can lead to improved efficiency and reduced latency in parallel computing applications that utilize both message passing and shared memory [12].

Moving forward, several mitigation strategies were introduced. The most notable advancement was in MPI 4.0's Partitioned Communications [7,9], which allowed multiple contributors to one communication. As matching is done at the initialization of the partitioned communication, the matching step can be removed and buffers can directly be moved to the destination.

FinePoints [11] were also considered, allowing for shared memory ranks to be added to a given rank in MPI, thus enabling MPI partitioned to work with shared-memory in a thread-aware manner. However, the Allocating Receive and Freeing Send operations [21] were initially considered but ultimately rejected by the MPI forum due to issues with implementing derived datatypes on such in-place transfers. Eventually, the proposal for an endpoint feature (the ability to add shared-memory ranks to a given rank), also known as MPI Endpoints [20], remained unfinished due to lack of consensus from the broader MPI community.

Overall, the most commonly used MPI interfaces are still copy-oriented and do not take advantage of a shared-memory setup, impeding memory duplication between the source and destination buffer. Furthermore, ways of numbering ranks within a shared-address space are not part of the standard, preventing the exploration of alternative copy-avoiding messaging mechanisms that might benefit from the performance gains we have exposed in previous sections. Consequently, MPI is bound to be avoided at the node level and limited to inter-node exchanges unless it can expose dedicated interfaces to efficiently address node-level topology. As an embedding programming model, being able to "name" threads bound to various localities within a given process may allow for finer-grain usage of nested programming models while facilitating software differentiation (workflows [19], MPMD [27], service oriented [25], invasive computing [17]).

4.2 Shared Memory Extensions to MPI

It is not because MPI did not address shared-memory issues in recent years that MPI cannot propose solutions for such scenarios. Indeed, MPI is primarily designed to share data between multiple MPI processes and even nodes, ensuring high performance and portability over many decades. However, as mentioned earlier, we have shown that it is no longer capable of delivering peak performance for node-local operations. In order to deal with this state of affairs, we identify two major axes of evolution for MPI: process addressing and buffer-sharing.

MPI is the programming model for spatializing computation. It acts as the *embedding* model, as most HPC simulations are initially structured around the linear ranking of MPI. However, we have seen that such addressing was not possible at the node level with full performance, encouraging nested programming models instead. In recent evolutions of MPI regarding topology support, it would be desirable to allow MPI to expose shared memory in its numbering scheme – manifesting as endpoints. Shared memory structuring becomes increasingly significant when observing converged architectures that feature differentiated computing units bound to the same memory. Similarly, this would facilitate the

use of a single process per node by enabling precise addressing of remote endpoints, which may exhibit differing behavior due to node-level NUMA effects.

Another way of improvement is the support for thread-aware communication primitives. This aligns with the ownership passing proposal, enabling data exchanges between ranks without doing any copy, effectively swapping pointers [2]. Additionally, this can serve as a foundation for coordination operations between collocated ranks, implementing shared memory queues [18,6,24] for inter-thread communication or Remote Procedure Calls (RPCs) [25,29]. In fact, there is a wide range of communication operations in a shared-memory environment that MPI has overlooked and which seems relevant to us. Therefore, enabling their use at the interface level, if possible as transparently as possible, appears crucial given the performance difference we have measured. Future work on MPI sessions exposing multiple MPI contexts in the same process could be a potential avenue for this support.

4.3 Thread-Based and Shared-Address Space Approaches

One potential strategy for transforming MPI into a more favorable execution environment is translating it into alternative models. In this context, several methods have emerged to bridge the gap between MPI processes. Two notable examples are Process in Process (PiP) [14,22], which employs the `dlmopen` call, and running MPI within threads [8,15,23]. However, the applicability of these methods raises concerns about compatibility with existing codes that might not be suitable for execution in threads due to their reliance on unique global variables. To overcome these challenges, it is essential to convert global variables into thread-local storage (TLS). For instance, MPC introduces a modified compiler that offers extended TLS levels for this purpose [1]. The concept of converting processes into a shared-address space represents an indirect approach to compensate the lack of shared-memory primitives in MPI. This method provides the advantage of enabling the porting of existing codes with minimal modifications, although it requires more effort at compile time to ensure correct compilation and privatization of the programs. Other approaches, such as SMARTMAP [3] and PVAS [26], consider exposing process images that can run in shared-memory by leveraging the operating system. One advantage of this indirect approach is its native support for legacy codes that do not require evolution to leverage shared-memory approaches. However, having such support in MPI would allow for a simpler implementation, without relying on either a modified compiler or operating system.

5 Evaluating Shared-Memory Gains for MPI

To better illustrate the performance improvements obtained with shared memory in MPI, let's examine a common benchmark: the OSU bibw, which measures bidirectional bandwidth between two MPI processes. We ran this benchmark on MPC with process-based (CMA enabled), thread-based, OpenMPI with CMA

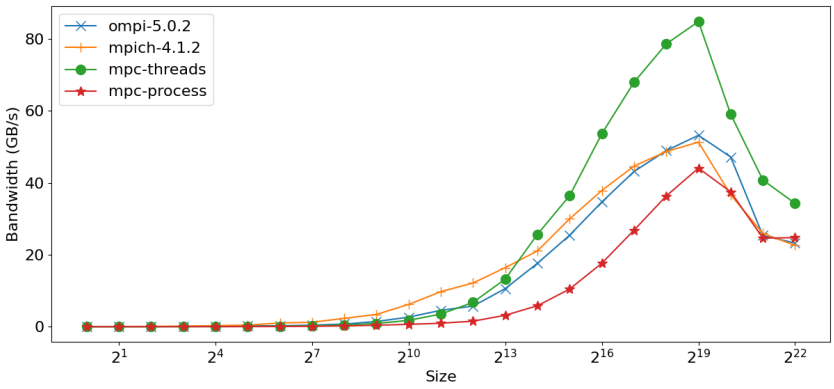

Fig. 3: Comparison of OSU Bi-Bandwidth benchmark performance between MPC other MPIs on the Intel Sapphire Rapids 4410T.

support[6], and MPICH compiled with UCX while forcing CMA [7]. Figure 3 shows a performance improvement of 59% between MPC and OpenMPI, aligning with the expected gain of 67% for small messages inside a socket indicated in Table 1. This improvement may be put into perspective because of the added overhead of locking and MPI semantics management. For smaller sizes, MPC incurs higher overheads due to its always-on thread-based nature, but it still manages to utilize shared memory bandwidth for larger messages unlike other MPI implementations. It is important to note that XMPEM would have alleviated this observation, allowing for improved bandwidth, but it is less portable and requires a modified kernel. In contrast, shared memory is a more common solution.

Overall, through a set of relatively simple measurements utilizing common and widely accessible interfaces, we have shown that extensions to MPI enabling shared memory present an appealing approach for maintaining the advantages of both thread-based and process-based MPI approaches. In general, standard MPI (both SHM and CMA) is incapable of fully exploiting the bandwidth in a shared-memory system. On the other hand, blending models introduces complexity to programs and may restrict them to either proprietary or non-portable shared-memory runtimes. Therefore, permitting MPI to address shared-memory locality would allow a more precise division of labor between shared-memory models while leveraging the widespread and portable nature of MPI.

6 Conclusion

In this study, we examined the limitations of data transfer performance between shared memory and accross processes, particularly within the context of MPI. Our investigation revealed that MPI's current model of mapping processes to

[6] `spack install openmpi fabrics=cma`
[7] `spack install mpich netmod=ucx ^ucx +cma` and `export UCX_TLS=sm`

UNIX ones creates a barrier that hampers efficient data transfers within nodes. This prompted us to explore alternative approaches for adapting MPI to shared-memory environments.

Three potential solutions were investigated: the existing model with one process per node, thread-based MPI with its associated complexities and possible locking overheads, and rethinking the nature of MPI processes to expose a node-level communication layer in shared memory contexts. We concluded that the latter approach offers the most promising extension to MPI, enhancing programmability in converged architectures by allowing locality and communication within processes to be expressed transitively with respect to existing codebases.

This conclusion is supported by our findings that demonstrate integrating shared memory support into MPI can significantly enhance performance, address locality issues, and enable a more accurate division of labor between shared-memory models. These benefits include the full utilization of bandwidth in shared-memory systems, ensuring compatibility with existing codes, and providing significant performance improvements while maintaining wide system and application compatibility.

7 Future Work

In this study, our objective is to delve into shared-memory extensions for MPI. Specifically, we believe that the Session model has the potential to allow us to expose endpoints in a transparent manner within MPI. The context nesting facilitated by sessions enables us to generate endpoints by dynamically assigning ranks to all local threads participating in the `MPI_Comm_create_from_group` collective call. With a dedicated *pset*, MPI could operate in shared-memory without requiring any modifications. However, it is likely that runtimes and associated MPI calls would need tuning to fully leverage this new configuration. Under such circumstances, all RDMAs would become memcpys, message transfers would also be handled as memcpys, and both the source and destination could initiate the copy. Ultimately, on these shared-memory communicators, pointers to data could be exchanged directly rather than systematically copying.

References

1. Besnard, J.B., Adam, J., Shende, S., Pérache, M., Carribault, P., Jaeger, J., Malony, A.D.: Introducing task-containers as an alternative to runtime-stacking. In: Proceedings of the 23rd European MPI Users' Group Meeting. pp. 51–63 (2016)
2. Besnard, J., Malony, A.D., Shende, S., Pérache, M., Carribault, P., Jaeger, J.: An MPI halo-cell implementation for zero-copy abstraction. In: Dongarra, J.J., Denis, A., Goglin, B., Jeannot, E., Mercier, G. (eds.) Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, Bordeaux, France, September 21-23, 2015. pp. 3:1–3:9. ACM (2015). https://doi.org/10.1145/2802658.2802669, `https://doi.org/10.1145/2802658.2802669`

3. Brightwell, R., Pedretti, K., Hudson, T.: Smartmap: Operating system support for efficient data sharing among processes on a multi-core processor. In: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. IEEE (2008)
4. Buntinas, D., Mercier, G., Gropp, W.: Implementation and evaluation of shared-memory communication and synchronization operations in MPIch2 using the nemesis communication subsystem. Parallel Computing **33**(9), 634–644 (2007)
5. Chen, C.C., Shafie Khorassani, K., Kousha, P., Zhou, Q., Yao, J., Subramoni, H., Panda, D.K.: MPI-xccl: A portable MPI library over collective communication libraries for various accelerators. In: Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. pp. 847–854 (2023)
6. Dilley, N., Lange, J.: An eMPIrical study of messaging passing concurrency in go projects. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 377–387. IEEE (2019)
7. Dosanjh, M.G., Worley, A., Schafer, D., Soundararajan, P., Ghafoor, S., Skjellum, A., Bangalore, P.V., Grant, R.E.: Implementation and evaluation of MPI 4.0 partitioned communication libraries. Parallel Computing **108**, 102827 (2021)
8. Friedley, A., Bronevetsky, G., Hoefler, T., Lumsdaine, A.: Hybrid MPI: efficient message passing for multi-core systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (2013)
9. Gillis, T., Raffenetti, K., Zhou, H., Guo, Y., Thakur, R.: Quantifying the performance benefits of partitioned communication in MPI. In: Proceedings of the 52nd International Conference on Parallel Processing. pp. 285–294 (2023)
10. Goglin, B., Moreaud, S.: Knem: A generic and scalable kernel-assisted intra-node MPI communication framework. Journal of Parallel and Distributed Computing **73**(2), 176–188 (2013)
11. Grant, R.E., Dosanjh, M.G., Levenhagen, M.J., Brightwell, R., Skjellum, A.: Finepoints: Partitioned multithreaded MPI communication. In: High Performance Computing: 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16–20, 2019, Proceedings 34. Springer (2019)
12. Hoefler, T., Dinan, J., Buntinas, D., Balaji, P., Barrett, B., Brightwell, R., Gropp, W., Kale, V., Thakur, R.: MPI+ MPI: a new hybrid approach to parallel programming with MPI plus shared memory. Computing **95**, 1121–1136 (2013)
13. Hori, A., Ouyang, K., Gerofi, B., Ishikawa, Y.: On the difference between shared memory and shared address space in hpc communication. In: Asian Conference on Supercomputing Frontiers. Springer International Publishing Cham (2022)
14. Hori, A., Si, M., Gerofi, B., Takagi, M., Dayal, J., Balaji, P., Ishikawa, Y.: Process-in-process: techniques for practical address-space sharing. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing. pp. 131–143 (2018)
15. Huang, C., Lawlor, O., Kale, L.V.: Adaptive MPI. In: Languages and CoMPIlers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers 16. pp. 306–322. Springer (2004)
16. Jin, H.W., Sur, S., Chai, L., Panda, D.K.: Limic: Support for high-performance MPI intra-node communication on linux cluster. In: 2005 International Conference on Parallel Processing (ICPP'05). pp. 184–191. IEEE (2005)
17. John, J., Narvaez, S., Gerndt, M.: Invasive computing for power corridor management. Parallel Computing: Technology Trends **36**, 386 (2020)
18. Malony, A.D., Reed, D.A., McGuire, P.J.: Mpf: A portable message passing facility for shared memory multiprocessors. Tech. rep. (1987)

19. Martinelli, A.R., Torquati, M., Aldinucci, M., Colonnelli, I., Cantalupo, B.: Capio: a middleware for transparent i/o streaming in data-intensive workflows. In: 2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC). pp. 153–163. IEEE (2023)
20. MPI Forum: MPI Endpoints Proposal. `https://github.com/MPI-forum/MPI-issues/issues/56` (2015), accessed: 2024
21. MPI Forum: Arecv/Fsend Proposal. `https://github.com/MPI-forum/MPI-issues/issues/32` (2016), accessed: 2024
22. Ouyang, K., Si, M., Hori, A., Chen, Z., Balaji, P.: Cab-MPI: exploring interprocess work-stealing towards balanced MPI communication. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE (2020)
23. Pérache, M., Jourdren, H., Namyst, R.: Mpc: A unified parallel runtime for clusters of numa machines. In: Euro-Par 2008–Parallel Processing: 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008. Proceedings 14. pp. 78–88. Springer (2008)
24. Pieper, R., Löff, J., Hoffmann, R.B., Griebler, D., Fernandes, L.G.: High-level and efficient structured stream parallelism for rust on multi-cores. Journal of Computer Languages **65**, 101054 (2021)
25. Ross, R.B., Amvrosiadis, G., Carns, P., Cranor, C.D., Dorier, M., Harms, K., Ganger, G., Gibson, G., Gutierrez, S.K., Latham, R., et al.: Mochi: Composing data services for high-performance computing environments. Journal of Computer Science and Technology **35**, 121–144 (2020)
26. Shimada, A., Gerofi, B., Hori, A., Ishikawa, Y.: Proposing a new task model towards many-core architecture. In: Proceedings of the First International Workshop on Many-core Embedded Systems. pp. 45–48 (2013)
27. Shimosaka, T., Murai, H., Sato, M.: A design of a communication library between multiple sets of mpi processes for mpmd. In: 2014 IEEE 17th International Conference on Computational Science and Engineering. pp. 1886–1893. IEEE (2014)
28. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., et al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2021)
29. Vef, M.A., Moti, N., Süß, T., Tocci, T., Nou, R., Miranda, A., Cortes, T., Brinkmann, A.: Gekkofs-a temporary distributed file system for hpc applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 319–324. IEEE (2018)
30. Venkata, M.G., Graham, R.L., Hjelm, N.T., Gutierrez, S.K.: Open MPI for cray xe/xk systems. Proceedings of the 2012 Cray User Group, Greengineering the Future. Stuttgart, Germany (2012)
31. Vienne, J.: Benefits of cross memory attach for MPI libraries on hpc clusters. In: Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment. pp. 1–6 (2014)
32. Weingram, A., Li, Y., Qi, H., Ng, D., Dai, L., Lu, X.: xccl: A survey of industry-led collective communication libraries for deep learning. Journal of Computer Science and Technology **38**(1), 166–195 (2023)