Can OpenMP Scale Beyond the Node? A Performance Evaluation of Remote Offloading via the MPI Proxy Plugin

Jhonatan Cléto[1], **Guilherme Valarini**[1], **Hervé Yviquel**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

j256444@dac.unicamp.br, hyviquel@unicamp.br

Abstract. *We evaluate the MPI Proxy Plugin (MPP), an extension to LLVM/OpenMP that enables remote offloading of target regions via MPI, allowing distributed GPU execution without modifying application code. Using benchmarks on NVIDIA H100 and AMD MI300A nodes, we show that MPP delivers competitive performance compared to traditional MPI+OpenMP, particularly for coarse-grained, compute-intensive workloads. While MPP simplifies development and enables communication–computation overlap, it introduces higher runtime overheads and task granularity requirements. Our results position MPP as a promising step toward unifying shared and distributed heterogeneous programming under the OpenMP model.*

1. Introduction

High-performance computing (HPC) applications increasingly rely on heterogeneous systems that combine multi-core CPUs with accelerators such as GPUs to achieve high computational throughput [Chatterjee et al. 2013]. OpenMP has emerged as a widely adopted programming model for exploiting shared-memory parallelism and offloading computations to local accelerators through the `target` directive. Its directive-based model enables portable and scalable parallelization with minimal code changes, lowering the barrier to entry for heterogeneous programming.

Despite these advantages, OpenMP offloading remains limited to devices within the local node, constraining its applicability in distributed-memory environments commonly found in modern HPC systems [Laguna et al. 2019]. To overcome this limitation, the LLVM/OpenMP runtime has been extended with the *MPI Proxy Plugin* (MPP) [Cléto et al. 2025], a runtime-level mechanism that enables OpenMP `target` regions to be offloaded to remote GPUs across a cluster through an MPI-based proxy layer. By integrating remote offloading into OpenMP's existing abstractions, MPP aims to unify heterogeneous and distributed programming under a single model, eliminating the need for hybrid MPI+OpenMP implementations at the application level.

However, this approach raises important questions regarding performance, overhead, and scalability. MPP introduces new sources of latency and coordination complexity, including host-side orchestration, inter-node communication, and data marshaling through the proxy layer. As OpenMP was originally designed around a host-centric, shared-memory model, it is unclear whether its extension to distributed environments via MPP maintains competitive performance when compared to conventional hybrid models.

This paper presents a comprehensive performance evaluation of MPP using a set of benchmarks. Our analysis spans multiple hardware architectures, and workload pat-

terns, enabling a detailed characterization of the runtime behavior, scalability, and overheads of remote OpenMP offloading in realistic HPC scenarios. The main contributions of this paper are:

- We provide a detailed characterization of the MPP across multiple hardware platforms and representative workloads.
- We analyze the computational and communication costs introduced by remote offloading, including data transfer overheads, kernel execution behavior, and inter-node latency.
- We compare the performance of MPP with traditional hybrid MPI+OpenMP implementations to identify strengths and limitations of each approach.
- We offer a cost-benefit analysis of using MPP in distributed heterogeneous systems, highlighting scenarios where MPP is advantageous and where its host-centric model imposes scalability constraints.

2. Background

OpenMP is a parallel programming model designed for shared-memory architectures, including multi-core CPUs and accelerators like GPUs [OpenMP 2024]. It enables developers to express parallelism in C, C++, and Fortran programs using compiler directives. In an OpenMP program, developers annotate code regions with directives and clauses that describe how the compiler should parallelize those regions. During compilation, the compiler interprets these annotations and generates the appropriate code to implement the specified parallel behavior. OpenMP follows the *fork-join* execution model. When a program encounters a `parallel` region, the master thread forks a team of threads that execute the enclosed code concurrently. After the region ends, all threads synchronize and join back into the master thread. Within a `parallel` region, additional constructs such as the `for` directive can be used to divide loop iterations among the threads. This allows for efficient work sharing in data-parallel loops, while the OpenMP runtime handles thread management and synchronization transparently to the programmer.

In addition to structured parallelism, OpenMP also supports task-based parallelism through the `task` directive. This allows developers to define units of work, or *tasks*, that can be executed asynchronously by any thread in the team. Unlike the `for` directive, which distributes iterations of a loop, the `task` directive enables more flexible parallelism, especially in irregular or recursive workloads. When a thread encounters a `task` construct, it may either execute the task immediately or defer its execution, allowing the runtime to schedule it dynamically. OpenMP maintains a task graph internally, capturing dependencies between tasks when specified using the `depend` clause. This abstraction allows the runtime to analyze the task graph and exploit parallelism while respecting data dependencies, offering a powerful model for fine-grained parallelism.

OpenMP also extends its task-based model to heterogeneous systems through the `target` directive, which enables offloading of computations to accelerators such as GPUs [Huber et al. 2022]. As show in Figure 1, when a program encounters a `target` region, the specified code is executed on the target device, while the host continues to orchestrate execution. This model integrates naturally with the task-based paradigm: each `target` region can be viewed as a task offloaded to an external device. OpenMP manages data movement between host and device automatically or under user control using

data-mapping clauses such as `map`. Combined with constructs like `target`, `teams`, and `distribute`, OpenMP allows expressing hierarchical parallelism on accelerators while preserving the high-level task graph abstraction. This unified approach enables portable and scalable parallel programs across CPUs and accelerators.

```
#pragma omp parallel for num_threads(num_devices)
for (int K = 0; K < num_devices; K++)
  ...
  #pragma omp target teams distribute parallel for \
          map(to: data[:size]) \
          device(K)
  gpu_task(data, ...);
```

Figure 1. Sample of OpenMP Target application.

In the LLVM implementation of OpenMP, target offloading is supported through a modular runtime infrastructure composed of two main components: libomp and liboffload. The libomp library is the standard OpenMP runtime that handles host-side execution and thread management. To support accelerator offloading, LLVM introduces liboffload (formerly libomptarget) [Antao et al. 2016], a runtime library responsible for managing device code execution and data transfers. liboffload uses a plugin-based architecture to support multiple types of target devices. Each plugin implements a low-level interface that handles communication with a specific backend, such as NVIDIA's CUDA or AMD's ROCm. At runtime, liboffload maps target regions to device-specific kernels, manages memory allocation and data mapping between host and device, and invokes the appropriate plugin to launch computation. This design enables portable offloading while keeping compiler and runtime decoupled and extensible.

To extend OpenMP offloading beyond the local node, we introduced the *MPI Proxy Plugin* (MPP) [Cléto et al. 2025], an extension to the liboffload runtime. MPP enables the offloading of OpenMP target regions to remote devices across a cluster by forwarding execution through an MPI-based proxy layer. From the application's perspective, the semantics of the `target` directive remain unchanged, however, the plugin intercepts offloading requests and communicates with a remote MPI rank that hosts the target device. This remote process acts as a proxy, receiving data, launching computation, and returning results. The MPP design preserves OpenMP's task-based model while introducing a distributed task graph that spans multiple nodes, enabling scalable parallelism across clusters. By leveraging standard MPI [MPI Forum 2023] communication and the existing liboffload plugin interface, MPP integrates seamlessly with the LLVM OpenMP runtime, requiring no changes to user code. Hybrid models typically combine MPI across nodes with OpenMP within nodes, offering performance but increasing complexity. In contrast, MPP hides inter-node offloading behind OpenMP's `target` directive, removing the need for explicit MPI code. This shifts communication to the runtime, simplifying development while preserving OpenMP's portability and extending it to distributed heterogeneous systems.

This paper investigates key research questions to assess the practicality of MPP in HPC settings: How does MPP perform and scale across different platforms and workloads? Is its performance comparable to traditional MPI+OpenMP? What are the computational and communication costs of remote offloading? While MPP offers a unified abstraction for distributed offloading, it inherits OpenMP's centralized, host-driven

model, where the host orchestrates all offloading, potentially limiting scalability. The additional indirection introduced by the proxy layer can also lead to overheads that offset performance gains. Through systematic evaluation, we aim to identify performance gaps, quantify overheads, and determine the cost-benefit trade-offs of using MPP versus hybrid MPI+OpenMP approaches.

3. Performance Study

To evaluate the performance of the MPI Proxy Plugin (MPP), a series of experiments was conducted. An overview of the experimental setup is provided in this section, followed by a discussion of the key findings and performance analysis across these scenarios.

3.1. Environment Setup

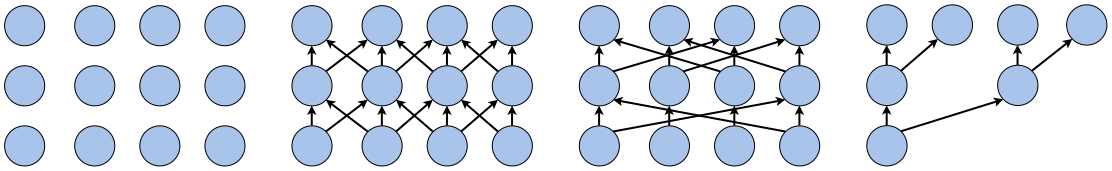
The experiments presented in the next section were conducted at the Santos Dumont 2 supercomputer located at the National Laboratory for Scientific Computing (LNCC) in Brazil. This supercomputer features a hybrid configuration of computing nodes, with Intel, AMD, ARM processors and NVIDIA GPUs, and AMD APUs (i.e., CPU + GPU on one die, sharing the same physical memory). Our experiments were conducted on two distinct node configurations, referred to throughout the paper as `sd2-h100` and `sd2-mi300a`. The `sd2-h100` nodes are equipped with two Intel Xeon SHR M9468 CPUs, four NVIDIA H100 GPUs, 1 TB of RAM, and 3.8 TB of SSD storage. The `sd2-mi300a` nodes feature two AMD Instinct MI300A APUs, 356 GB of RAM, and 3.84 TB of SSD storage. Due to cluster usage constraints, experiments were limited to a maximum of 11 nodes, with one GPU/APU per node, for each configuration.

The software stack includes Clang 21.0, CUDA 12.8, ROCm 6.4.1, MPICH 4.3, and UCX 1.18 with GPU-aware support enabled. Each data point reflects 5 trials, with 95% confidence intervals estimated via bootstrapping[1] and shown as translucent error bands. We employed Spinner [Ceccato et al. 2024], an open-source HPC benchmarking tool, to automate parameter sweeping and ensure reproducibility. The Spinner benchmark definitions, along other artifacts are available in a public repository[2].

3.2. Task Bench

Task Bench [Slaughter et al. 2020] is a configurable benchmark designed to evaluate the performance of parallel and distributed programming models using task graphs that model common communication and dependency patterns. To cover a wide range of application scenarios, Task Bench exposes a rich set of configuration parameters that define the structure and behavior of the task graph. These parameters control key aspects such as the graph dimensions (`height`, `width`), the communication pattern (`dependence`), the type (`kernel`) and duration (`iter`) of the kernel executed in each task, and communication volume (`output`). Together, they allow precise modeling of diverse workloads with varying degrees of parallelism, computational intensity, and communication overhead. Figure 2 shows examples of task graph configurations that can be modeled with Task Bench, including common patterns such as stencil, FFT, and tree.

[1] Bootstrapping (statistics)
[2] Experimental setup and results



(a) Trivial. (b) Stencil. (c) FFT. (d) Tree.

Figure 2. Sample task graphs from Task Bench.

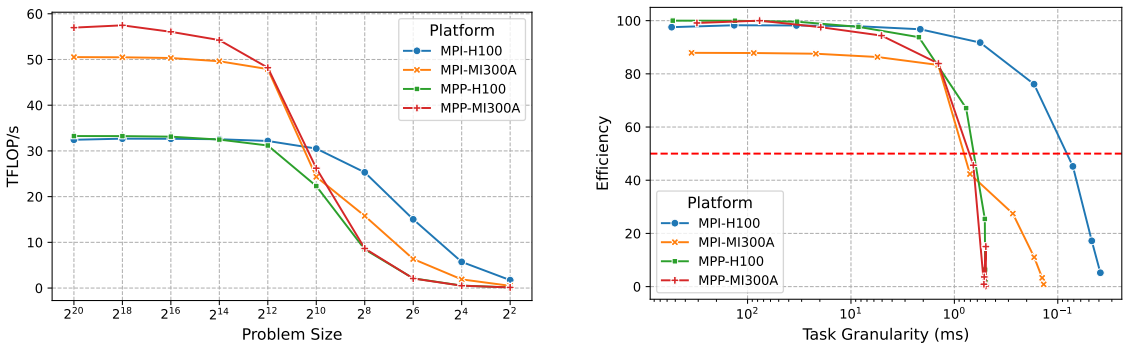
Instead of relying solely on strong or weak scaling, which may obscure system overheads, Task Bench introduces METG (Minimum Effective Task Granularity) as a metric. METG reflects the smallest task size at which a system maintains at least 50% of its peak FLOP/s, revealing how efficiently the system handles fine-grained parallelism. In this research, we adopt the same 50% efficiency threshold to assess and compare the scaling behavior of MPP on different configurations.

3.3. Experiments

To evaluate MPP performance, we extended Task Bench with MPP support and introduced a new GPU-based compute-bound kernel. We also used the original MPI+OpenMP implementation provided by Task Bench, modified slightly to support our custom GPU kernel. This hybrid version serves as the baseline for comparing MPP performance. Experiments used only MPICH, as other MPI libraries like OpenMPI showed inferior performance across platforms and benchmarks. To ensure that each system could reach its peak performance, we selected task graph configurations with sufficient granularity to offset runtime overheads.

3.3.1. Compute Kernel Performance

We first consider the peak performance achieved by each system in our tested platforms. For that experiment, we execute a 10×10 task graph with the trivial dependency type, that is without dependencies between the tasks. The experiment was conducted using only one device (GPU or APU) in each system with the aim of finding the maximum performance that each device is capable of providing.



(a) FLOPS vs problem size. (b) Efficiency vs task granularity.

Figure 3. Compute Kernel Performance.

Figure 3 shows the throughput (3a) and efficiency (i.e., as a percentage of the peak FLOP/s achieved) (3b) delivered by MPP and MPI as the problem size (i.e., number of iterations per kernel) and task granularity (i.e., wall time $\times$ num. devices/num. tasks) decreases. On both platforms, MPP delivers higher throughput than MPI at larger problem sizes. Indicating MPP's more dynamic task distribution compared to MPI's interleaved communication and computation. As task granularity decreases, the efficiency of both systems drops, approaching a vertical asymptote as overhead begins to dominate useful work. On `sd2-h100` nodes, the peak FLOP/s achieved was 33.4 TFLOP/s for MPP and 32.7 TFLOP/s for MPI, these values compares favorably with the officially reported number of 33.5 TFLOP/s. On `sd2-mi300a` nodes, the peak FLOP/s achieved was 56.8 TFLOP/s for MPP and 50.4 TFLOP/s for MPI, these values are respectively 7% and 17% less than the officially reported number of 61.3 TFLOP/s, which may indicate a space for optimization of the liboffload plugin for AMD GPUs.

In Figure 3b the horizontal dashed red line indicates 50% of efficiency, the interception between that line and the efficiency curves indicates the minimum task granularity on each system where the efficiency is at least 50%, in other words, it is the point that represents the METG(50%) of each system. As the Table 1 shows, MPI+OpenMP has a significantly smaller METG(50%) value than MPP in `sd2-h100` nodes. On `sd2-mi300a` nodes MPP has a smaller METG(50%), although the values for both systems are close. Additionally, `sd2-h100` nodes have also a smaller METG(50%) than `sd2-mi300a`. This indicates that METG is also influenced by the computing capacity of the platform.

Table 1. Minimum Effective Task Granularity - METG(50%)

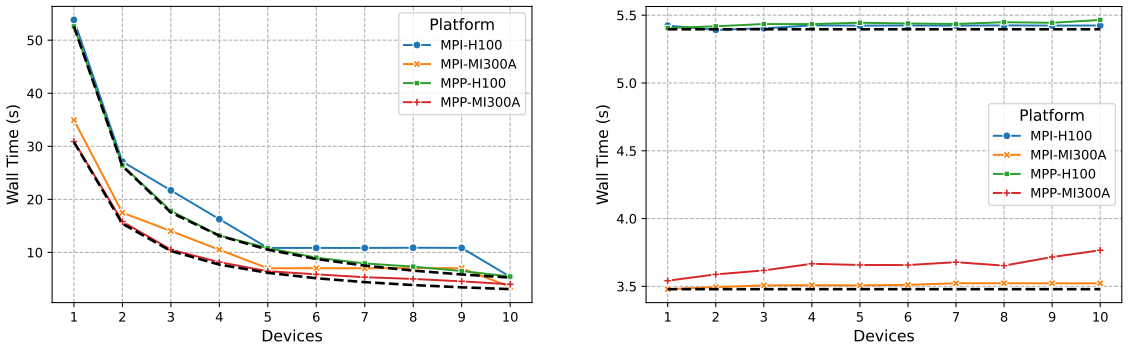
System	Device	METG(50%) (ms)
MPI+OpenMP	H100	0.09
MPP	H100	0.65
MPI+OpenMP	MI300A	0.79
MPP	MI300A	0.71

3.3.2. Scalability

Figure 4 shows the strong and weak scalability of both hybrid MPI+OpenMP and MPP. For strong scaling, a fixed 10×10 graph is used. For weak scaling, the graph has a fixed height of 10 and a width that increases with the number of devices (from 1 to 10), maintaining a constant workload of 10 tasks per device. The task graph uses stencil dependency type with 2^{20} iterations per kernel. The black line indicates ideal linear scaling from each platform's single-device peak performance.

In the strong-scaling plot (4a), wall time on `sd2-h100` nodes falls from 53 s to 7 s and on `sd2-mi300a` nodes from 35 s to 6 s as devices increase from 1 to 10, closely approaching the black line, indicating that the communication and runtime overheads has little impact on that task-graph. We also notice that MPP has better scalability than MPI+OpenMP in that configuration. In the weak-scaling plot (4b), wall time remains essentially flat, 5.4 s for `sd2-h100` and 3.48 s for `sd2-mi300a`, even as more devices are added, and both MPI+OpenMP and MPP maintain good scalability with a small gap

to the black line, confirming that proxy-mediated offloading imposes negligible penalty under balanced load. Note that MPI+OpenMP may scale better than MPP on more devices due to limitations of the centralized OpenMP model.



(a) Strong Scaling. **(b) Weak Scaling.**

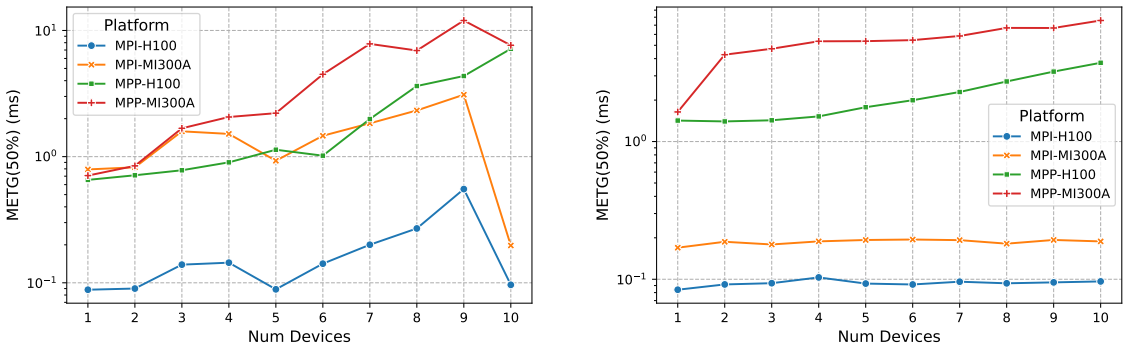
Figure 4. Wall Time per Number of Devices.

3.3.3. Runtime Overhead

Figure 5 reports the runtime overhead of each approach using the METG(50%) metric from Task Bench, which quantifies the smallest task granularity required to sustain half of peak efficiency. The METG(50%) curves plot the points where each runtime's efficiency curve, for each device count, intersects the 50% efficiency line.

In the strong-scaling plot (5a), the MPI+OpenMP baseline on `sd2-h100` maintains a very low METG(50%), around 0.1 ms on a single device, rising to 0.6 ms before dropping back to 0.1 ms at ten devices. The `sd2-mi300a` baseline starts near 0.8 ms and peaks at 3 ms before dropping back to 0.2 ms at ten devices. The performance improvement from nine to ten nodes in the MPI+OpenMP on METG(50%) graphs may reflect a shift in MPICH's communication strategy, such as switching protocols or enabling a more efficient progress engine. By contrast, MPP incurs significantly higher METG(50%): on `sd2-h100` it climbs from 0.7 ms on one device to over 7 ms at ten devices, and on `sd2-mi300a` from 0.7 ms to 12 ms. This highlights the increasing overhead of MPP as the number of devices grows, a consequence of OpenMP's host-centric model, which centralizes task management and limits scalability. In contrast, the distributed nature of the MPI+OpenMP approach handles larger device counts more effectively.

In the weak-scaling plot (5b), the MPI baselines exhibit nearly flat METG curves, 0.1 ms for `sd2-h100` and 0.2 ms for `sd2-mi300a`, demonstrating minimal overhead per device. MPP likewise shows stable METG(50%) line across devices but at elevated levels, growing from 1.3 ms to 4.7 ms for `sd2-h100` and from 1.64 ms to 7.6 ms for `sd2-mi300a`, indicating a runtime overhead on MPP that grows with scale. Overall, the METG analysis reveals that MPP's runtime overhead has different behavior in weak and strong scaling, having smoother growth when the workload per device is kept constant during scale-up. Although, its minimum efficiency task granularity requirements are substantially higher than the hybrid MPI+OpenMP implementation, particularly at high device counts.

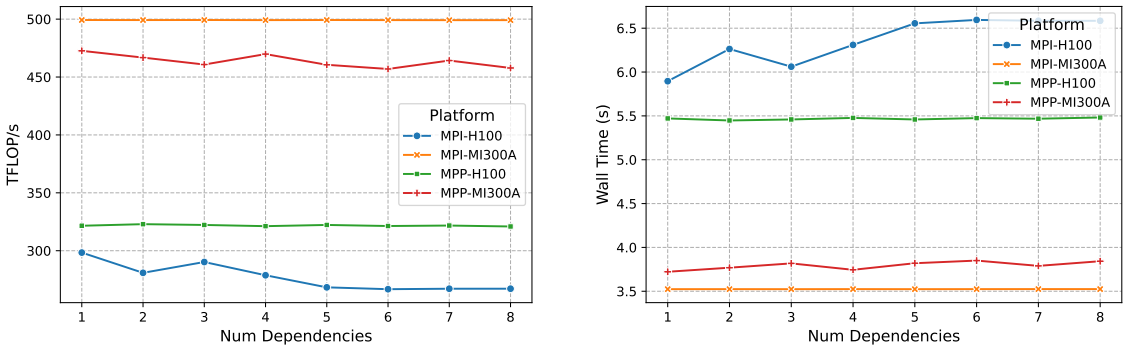


(a) Strong Scaling. **(b) Weak Scaling.**

Figure 5. METG(50%) per Number of Devices.

3.3.4. Communication Overhead

Figure 6 shows how increasing the number of dependencies in a task graph affects communication overhead on both the `sd2-h100` and `sd2-mi300a` platforms. The experiment used a fixed 10×10 task graph over ten devices, with 2^{20} iterations and 2^{16} output bytes per task, varying dependencies from 1 to 8. On `sd2-h100`, the hybrid MPI+OpenMP implementation shows a clear degradation as dependencies grow from 1 to 8: FLOP/s falls from 298 TFLOP/s to 267 TFLOP/s, while wall time increases from 5.9 s to 6.6 s (Figures 6a and 6b). By contrast, MPP on `sd2-h100` maintains a nearly flat profile of 320 TFLOP/s and 5.45 s, indicating that its proxy layer effectively hides the incremental synchronization cost. On `sd2-mi300a`, the MPI+OpenMP baseline remains essentially constant at 500 TFLOP/s and 3.5 s wall time across all dependency counts, reflecting robust communication handling. MPP on `sd2-mi300a` exhibits a modest decline in throughput, from 475 TFLOP/s at one dependency to 455 TFLOP/s at eight, and a slight increase in wall time from 3.7 s to 3.85 s.



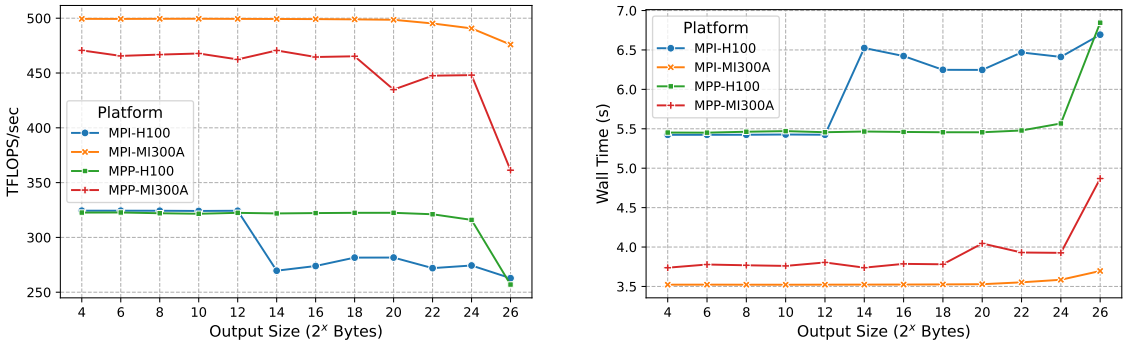
(a) FLOP/s. **(b) Wall Time.**

Figure 6. Communication Overhead.

3.3.5. Data Transfer Overhead

Figure 7 shows the impact of increasing the output size per task, isolating data transfer overhead. The experiment used a fixed 10×10 task graph over ten devices, with

2^{20} iterations and three dependencies per task, varying the output size per task from 2^4 to 2^{26} bytes. Both MPP and MPI+OpenMP use GPU-aware MPI to avoid unnecessary host staging. On `sd2-mi300a`, MPP sustains over 450 TFLOP/s and low wall time up to 2^{22} bytes, but performance drops sharply at 2^{26} due to accumulated memory pressure and limited overlap. MPI+OpenMP maintains higher stability, with minimal wall time increase and nearly constant throughput, thanks to fine-grained control over data movement. On `sd2-h100`, MPP achieves higher throughput for small to moderate sizes, while MPI+OpenMP delivers comparable throughput up to 2^{12}, but its performance declines beyond that point. These results highlight MPP's effective communication-computation overlap using asynchronous coroutines and non-blocking MPI, though at the cost of increased runtime overheads under high data volume. In summary, MPP is preferable for moderate transfer sizes, benefiting from its simpler model and automatic overlap, while MPI+OpenMP offers superior performance when communication dominates, particularly in the `sd2-mi300a` scenario.



(a) FLOP/s. **(b) Wall Time.**

Figure 7. Data Transfer Overhead.

3.4. Outcomes

Our evaluation reveals a nuanced cost–benefit trade-off between the MPI Proxy Plugin (MPP) and a traditional MPI+OpenMP hybrid approach. On one hand, MPP delivers best or near strong- and weak-scaling behavior to MPI+OpenMP (Figure 4) in workloads with high task granularity, and even masks the rising communication overhead of complex dependency patterns (Figure 6) by encapsulating data-movement and kernel launches behind its proxy layer. This makes MPP particularly attractive for workloads with irregular or heavy dependency graphs: developers benefit from automatic overlap of communication and computation without hand-tuned MPI calls, and the unified OpenMP-only codebase simplifies development, debugging, and portability across CPU-GPU clusters.

On the other hand, MPP's host-centric design imposes a higher minimum effective task granularity (METG), as shown in Figure 5: remote offloading overheads remain roughly an order of magnitude above the hybrid MPI+OpenMP baseline, and grow with scaling. Consequently, fine-grained tasks or memory-bound kernels suffer more pronounced performance penalties under MPP. Moreover, because all offload coordination still funnels through the host thread, extreme scale-out scenarios or very small tasks can expose bottlenecks in proxy orchestration and MPI communication. Another observation is that MPP achieves better scalability on H100 compared to MI300A. This suggests that

the CUDA-based implementation of liboffload handles asynchronous task graph execution more efficiently than its AMD counterpart, particularly on MI300A APUs.

In practice, MPP is most advantageous when task granularity is sufficiently coarse, so that METG remains below the task duration, and when the programming productivity gains of an OpenMP only model outweigh the modest runtime overheads. For compute, intensive kernels with moderate to high dependency complexity, MPP can match or even surpass the ease of use and maintainability of MPI+OpenMP, while delivering competitive performance. Conversely, for highly fine-grained workloads or scenarios demanding the absolute lowest communication latency, a hand-tuned MPI+OpenMP implementation remains the better choice.

4. Related Work

A number of prior studies have evaluated the productivity, portability, and performance of OpenMP and task-based models on modern heterogeneous systems. [Martineau and McIntosh-Smith 2017] investigate OpenMP 4.5 for scientific applications on Intel Xeon Phi, POWER8, and NVIDIA GPUs, demonstrating that while memory-bandwidth–bound codes map well across architectures, latency-bound kernels exhibit inconsistent performance. Their work highlights the importance of runtime optimizations and portable code idioms for maximizing OpenMP's effectiveness on diverse hardware.

Task-based runtime systems such as PaRSEC [Heroux et al. 2025] and Template Task Graph (TTG) [Bosilca et al. 2020] have pushed the frontier of fine-grained dataflow execution. [Schuchart et al. 2022] present optimizations in TTG to eliminate contentious atomics and reduce task-management overhead to a few hundred cycles, achieving scalable distributed execution that can rival OpenMP in shared memory. Their findings underscore the trade-off between low scheduling overhead and the complexity of managing dependencies in a distributed graph.

Performance portability across GPU programming models has also received considerable attention. [Davis et al. 2024] compare CUDA, HIP, Kokkos, RAJA, OpenMP, OpenACC, and SYCL on AMD and NVIDIA GPUs using proxy applications from multiple domains. They show that no single model delivers uniformly high performance on both platforms, and emphasize the need for portable abstractions that hide hardware-specific details without sacrificing efficiency.

In the context of distributed heterogeneous offloading, several frameworks enable distributed task graph execution. For example, Charm++ [Kale and Krishnan 1993], and Legion [Bauer et al. 2012] support remote task execution through sophisticated runtime schedulers, but both require application-level API changes. Our MPI Proxy Plugin differs by preserving the standard OpenMP interface and transparently forwarding target regions to remote GPUs via MPI.

5. Conclusion

This paper presented a detailed performance evaluation of the MPI Proxy Plugin (MPP), an extension to the LLVM/OpenMP runtime that enables remote offloading of target regions across a cluster via MPI. By analyzing a variety of benchmarks and mini-applications across multiple architectures, we characterized the behavior, overheads, and scalability of MPP in realistic HPC settings.

Our results show that while MPP provides a unified abstraction for distributed heterogeneous execution, its performance is highly dependent on workload characteristics and communication costs. The centralized host-driven model of OpenMP, combined with the overheads of proxy-based offloading, can limit scalability in fine-grained or communication-heavy scenarios. However, in compute-intensive kernels with coarse-grained tasks, MPP can offer performance competitive with traditional hybrid MPI+OpenMP approaches while significantly reducing programming complexity.

Overall, MPP represents a promising step toward unifying shared- and distributed-memory parallelism under the OpenMP model. Future work includes improving data management strategies, exploring decentralized task coordination, and integrating MPP with emerging runtime and scheduling strategies to further enhance its scalability and adaptability.

Acknowledgments

This work is supported by the São Paulo Research Foundation (FAPESP) under grants 2019/26702-8 and 2024/04232-8, by FAEPEX/UNICAMP through the Programa de Incentivo a Novos Docentes (PIND) under grant 2553/23. We also thank LNCC for granting the computational resources at the Santos Dumont supercomputer.

References

Antao, S. F., Bataev, A., Jacob, A. C., Bercea, G.-T., Eichenberger, A. E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C., and O'Brien, K. (2016). Offloading Support for OpenMP in Clang and LLVM. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–11.

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11.

Bosilca, G., Harrison, R., Herault, T., Javanmard, M., Nookala, P., and Valeev, E. (2020). The template task graph (ttg) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale. In *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 1–7.

Ceccato, R., Cléto, J., Leite, G., Rigo, S., Diaz, J. M. M., and Yviquel, H. (2024). Spinner: Enhancing HPC Experimentation with a Streamlined Parameter Sweep Tool. In *2024 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 1–11.

Chatterjee, S., Taşirlar, S., Budimlić, Z., Cavé, V., Chabbi, M., Grossman, M., Sarkar, V., and Yan, Y. (2013). Integrating asynchronous task parallelism with MPI. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 712–725.

Cléto, J., Valarini, G., Pereira, M., Araujo, G., and Yviquel, H. (2025). Scalable OpenMP Remote Offloading via Asynchronous MPI and Coroutine-Driven Communication. In *Proceedings of the 31st International Conference on Parallel Processing (Euro-Par 2025)*, Lecture Notes in Computer Science, Dresden, Germany. Springer.

Davis, J. H., Sivaraman, P., Kitson, J., Parasyris, K., Menon, H., Minn, I., Georgakoudis, G., and Bhatele, A. (2024). Taking GPU Programming Models to Task for Performance Portability.

Heroux, M., Bouteiller, A., Herault, T., Cao, Q., Schuchart, J., and Bosilca, G. (2025). PaRSEC: Scalability, flexibility, and hybrid architecture support for task-based applications in ECP. *Int. J. High Perform. Comput. Appl.*, 39(1):147–166.

Huber, J., Cornelius, M., Georgakoudis, G., Tian, S., Diaz, J. M. M., Dinel, K., Chapman, B., and Doerfert, J. (2022). Efficient Execution of OpenMP on GPUs. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 41–52.

Kale, L. V. and Krishnan, S. (1993). Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 91–108, New York, NY, USA. Association for Computing Machinery.

Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., and Sultana, N. (2019). A large-scale study of MPI usage in open-source HPC applications. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*.

Martineau, M. and McIntosh-Smith, S. (2017). The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs. In de Supinski, B. R., Olivier, S. L., Terboven, C., Chapman, B. M., and Müller, M. S., editors, *Scaling OpenMP for Exascale Performance and Portability*, pages 185–200, Cham. Springer International Publishing.

MPI Forum (2023). *MPI: A Message-Passing Interface Standard Version 4.1*.

OpenMP (2024). Openmp application programming interface version 6.0. Technical report, OpenMP Architecture Review Board.

Schuchart, J., Nookala, P., Herault, T., Valeev, E. F., and Bosilca, G. (2022). Pushing the Boundaries of Small Tasks: Scalable Low-Overhead Data-Flow Programming in TTG. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 117–128.

Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K. S., Cao, Q., Bosilca, G., Mirchandaney, S., Leek, W., Treichlerk, S., McCormick, P., and Aiken, A. (2020). Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.