

Comparing MPI, MPP, and StarPU Runtimes for Distributed GPU Execution: A Case Study Using N-Body Simulations

Jhonatan Cléto¹, Nicolas Vanz², Hervé Yviquel¹, Márcio Castro²

¹ Universidade Estadual de Campinas (UNICAMP) – Campinas – Brasil

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

² Universidade Federal de Santa Catarina (UFSC) – Florianópolis – Brasil

nicolas.vans@ufsc.br, marcio.castro@ufsc.br

Abstract. *Graphics Processing Unit (GPU) accelerators are a central component of modern High Performance Computing (HPC) systems, but scaling applications across multiple GPUs and nodes requires efficient coordination of computation, communication, data movement, and task dependencies. Different programming models and runtime systems expose these concerns at different abstraction levels, trading programmer control for runtime automation. In this work, we compare three approaches to distributed GPU execution: the Message Passing Interface (MPI), the MPI Proxy Plugin (MPP), and StarPU. Our evaluation uses a direct-method N-Body simulation and Task Bench on up to 32 AMD Instinct MI210 GPUs across eight nodes. The results show that distributed GPU performance depends jointly on runtime organization, kernel implementation, task granularity, and dependency structure, highlighting trade-offs between explicit control, programming portability, and runtime-managed execution.*

1. Introduction

High Performance Computing (HPC) workloads have been relying on GPUs as the primary source of computational throughput, while problem sizes continue to grow beyond the capacity and performance of a single accelerator. As a result, practical HPC applications must scale GPU execution beyond one device and one node, orchestrating a pool of accelerators spread across a cluster as one coherent execution flow rather than as isolated resources [Netto et al. 2018].

Distributed GPU execution provides two complementary benefits: it increases aggregate compute and memory capacity beyond what a single accelerator offers, and it reduces time-to-solution by exposing more parallelism across devices. Neither benefit is automatic: inter-node communication adds latency and bandwidth pressure, data placement affects locality, and synchronization can dominate execution time if dependencies are not structured carefully, so scalability is a consequence of runtime-level decisions about computation and communication orchestration [Augonnet et al. 2010]. The programming model and runtime determine how much of this coordination is exposed to the programmer and how much is managed automatically.

The Message Passing Interface (MPI) is the standard for distributed-memory parallelism: it gives the programmer full, explicit control over process placement and point-to-point or collective communication, but it also makes every rank responsible for know-

ing what data it owns, whom to talk to, and when. For GPU execution, Compute Unified Device Architecture (CUDA) and Heterogeneous-Compute Interface for Portability (HIP) [AMD 2026] are commonly used to program accelerators, and combining them with MPI is the conventional approach to multi-node, multi-GPU execution. OpenMP, since the introduction of the `target` directive, provides an alternative directive-based model for offloading computation to a locally attached accelerator, with data movement expressed through `map` clauses rather than explicit copies. MPI+OpenMP therefore provides an alternative to MPI+CUDA/HIP while retaining the OpenMP programming model for GPU execution, but it pushes communication, synchronization, and load balancing across distributed GPUs onto the application programmer, who must manually interleave MPI calls with OpenMP `target` regions and reason about correctness across both models at once.

Two lines of runtime research approach this integration problem from opposite directions. The first is *task-based runtimes*, exemplified by StarPU, which replace the fork-join/message-passing split with a graph of tasks over explicitly registered data: the runtime decides where and when tasks execute and how data is transferred and kept coherent across memory spaces [Augonnet et al. 2010], extending this model to clusters through fully-distributed and server-client MPI modes. The second line preserves OpenMP’s directive-based model while extending it transparently across the network: the OpenMP Cluster Model [Yviquel et al. 2023] pioneered this approach by offloading OpenMP tasks to remote nodes over MPI, though targeting remote CPUs only, and the MPI Proxy Plugin (MPP) [Cléto et al. 2025a] extends it to GPUs, using LLVM’s `liboffload` to intercept OpenMP `target` regions and forward them, via asynchronous MPI and C++20 coroutines, to remote accelerators, so that an unmodified OpenMP application can offload to a cluster-wide GPU pool as it would to a local device.

These approaches provide contrasting answers to the same problem: exposing remote GPUs as task execution targets while differing in abstraction and control. StarPU exposes a remote GPU as a *scheduler-visible worker*: the runtime manages data handles, infers dependencies, and selects task placement based on cost models and worker availability, at the cost of adopting StarPU’s codelet and data-handle model. MPP instead exposes a remote GPU as another OpenMP `target` device: the programmer retains standard OpenMP, while the proxy hides the network behind existing `map/device` semantics, at the cost of a host-centric control path in which offload coordination funnels through one host thread.

Prior work from our group has evaluated each side of this comparison in isolation: Vanz et al. assessed N-Body simulations with StarPU, OpenMP, and MPI on cloud infrastructure [Vanz et al. 2025a, Vanz et al. 2025b], and Cléto et al. evaluated MPP’s transparent remote offloading on GPU clusters [Cléto et al. 2025a, Cléto et al. 2025b]. GPU remote execution in StarPU and GPU remote offloading in MPP were thus never compared against each other on a common distributed multi-GPU workload, and none of these works separates kernel-language effects from runtime effects or characterizes how dependency patterns modulate overhead. This paper closes these gaps: it confronts the two runtimes directly, on the same cluster, application, and task graphs, with paired HIP and OpenMP variants.

This gap motivates a broader question: *how do different programming models and*

runtimes affect the performance of distributed GPU applications as the number of GPUs and the structure of computation change? Runtime overhead cannot be considered independently of task dependencies and GPU-kernel implementation, since both determine how much work is available to amortize communication and runtime costs.

To answer this question, this paper evaluates MPI, MPP, and StarPU on a direct-method N-Body simulation, which is compute-intensive and iterative, with an $\mathcal{O}(n^2)$ interaction phase that makes it useful for studying GPU scaling, data movement, synchronization, and task scheduling. The evaluation is complemented by Task Bench [Slaughter et al. 2020] to characterize the impact of task structure and dependency patterns. The goal is not to identify a universally superior runtime, but to determine how programming abstractions and runtime organization affect performance as the number of GPUs and task structure change. Overall, the contributions of this paper are as follows:

- A controlled comparison of MPI, MPP, and StarPU for distributed execution on a multi-node cluster with up to 32 AMD Instinct MI210 GPUs using the N-Body application as a case study;
- A quantification of how task dependency patterns affect MPI, MPP, and StarPU using Task Bench, analyzing trivial, stencil, and tree graphs across increasing GPU counts;
- A separation of runtime effects from kernel-language effects with native HIP variants, using a single-GPU roofline analysis to explain the performance gap between the HIP and OpenMP N-Body kernels, alongside application-level strong scaling across up to 32 GPUs.

The remainder of this paper is organized as follows. Section 2 introduces the programming models and runtime systems considered in this study, Section 3 presents the N-Body problem and its implementations using MPI, MPP, and StarPU, Section 4 details the experimental setup and methodology and discusses the results, Section 5 discusses related work, and Section 6 concludes the paper.

2. Background

This section provides the background on the programming models and runtimes evaluated in this work. We first introduce OpenMP and HIP as the two GPU programming models used in our implementations, and then describe MPP and StarPU, which provide contrasting approaches to distributed GPU execution: transparent remote OpenMP offloading and runtime-managed task-based execution, respectively.

2.1. OpenMP and HIP

OpenMP is the standard directive-based model for shared-memory parallelism: a sequential program is annotated with compiler directives (`pragmas`), and the compiler and runtime translate them into parallel execution, originally across the host’s own threads. Since version 4.0, OpenMP extends this model to accelerators through the `target` directive: a target region is offloaded to a device (typically a GPU), with `map` clauses controlling host-device data transfers and `nowait` enabling asynchronous execution, while `teams` and `distribute` express the hierarchical parallelism the GPU exposes internally. The asynchronous, dependency-driven execution of OpenMP’s `task` directive carries over to target regions, so a target offload can itself be a task in a larger dependency graph, and

multiple local GPUs can be driven from one host process via a host-side parallel loop with an explicit `device` clause per iteration. In the LLVM implementation, this is realized through `liboffload` (formerly `libomptarget`), which maps target regions to device-specific kernels through vendor plugins (CUDA, ROCm, and others), keeping the compiler and the accelerator backend decoupled.

HIP is a C++ runtime API and kernel programming model designed to map data-parallel algorithms to massively parallel GPU architectures [AMD 2026]. Closely aligned with the CUDA programming model, HIP facilitates porting CUDA applications to AMD GPUs: kernels are expressed explicitly using extensions such as `__global__` and launched with a grid and block configuration, while the runtime API manages device selection, memory allocation, data transfers, and synchronization. The same source can target AMD GPUs through ROCm and, with appropriate toolchain support, NVIDIA GPUs.

The conventional approach to distributed, multi-GPU execution combines MPI with CUDA or HIP, or uses MPI+OpenMP target offloading; both require the programmer to manually interleave explicit communication with accelerator execution, manage two concurrency models at once, and hand-tune the overlap between network transfers and kernel execution. StarPU and MPP address this problem: the former replaces both models with a single task-based abstraction, while the latter keeps the OpenMP programming surface and hides MPI behind it.

2.2. Distributed OpenMP Offloading: OMPC and MPP

The OpenMP Cluster Model (OMPC) [Yviquel et al. 2023] pioneered transparent distributed OpenMP by forwarding OpenMP tasks to remote nodes over MPI, but it primarily targets remote CPU execution. The MPI Proxy Plugin (MPP) [Cléto et al. 2025a] extends this philosophy to GPUs. Built on LLVM’s `liboffload` runtime, MPP intercepts offloading requests issued through the `target` directive and forwards them to a remote MPI process that serves as a proxy for the target device: the proxy receives the required data, launches the target region on the remote accelerator, and returns the results to the originating process. MPP thus preserves the semantics of the OpenMP `target` directive while extending its execution model across multiple nodes without requiring modifications to user applications.

2.3. StarPU

StarPU is a task-based runtime system for heterogeneous architectures that combines asynchronous task submission, dynamic scheduling, and runtime-managed data consistency across accelerators and cluster nodes [Augonnet et al. 2010]. Although StarPU workers can also be ordinary CPU cores, this article restricts its attention to GPU execution. Its central idea is to decouple *what* is computed from *where* and *how* it is executed: the application registers its data with the runtime and submits *tasks*, instances of a *codelet* that bundles one or more interchangeable device implementations. Dependencies between tasks are not declared explicitly: they are inferred from the read/write access each task declares on its data handles, which lets StarPU build a directed acyclic graph of tasks and execute it asynchronously while a scheduler assigns each ready task to a worker. Data is tracked through opaque *handles* rather than raw pointers; the runtime moves and

replicates data between memory spaces on demand and keeps replicas coherent, so the application never issues host-device or node-to-node copies by hand.

Beyond a single node, StarPU offers the *full MPI model*, in which every rank runs its own StarPU instance and actively participates in scheduling: the application declares which rank owns each data handle and inserts the same task graph on every rank, and StarPU derives from that ownership which rank executes each task and which point-to-point transfers this implies, issuing them automatically under an owner-computes policy. This model scales well but makes the application *rank-aware*: every rank must reason about data ownership and task insertion for the whole graph.

3. N-Body Simulations

This section introduces the N-Body simulation used as the main application case study in our evaluation and describes the implementations proposed in this work.

3.1. Overview of the Problem

The N-body problem models the dynamics of a system of particles that interact through a pairwise force, typically gravitational or electrostatic, and is a standard benchmark in computational physics and astrophysics [Heggie 2005]. Its defining computational property is that, in the direct formulation, every particle influences and is influenced by every other particle, so a naive solution evaluates $\vec{F}_{ij}$ for every ordered pair at each simulation step, giving $\mathcal{O}(n^2)$ force evaluations per step. Hierarchical approximations such as Barnes-Hut and the fast multipole method reduce this cost by grouping distant particles, trading exactness and implementation simplicity for reduced complexity [Barnes and Hut 1986]. Because the $\mathcal{O}(n^2)$ direct method is computationally heavy, embarrassingly data-parallel across particle pairs, and admits a straightforward decomposition into independent per-particle or per-block tasks, it is widely used as a stress-test workload for GPU-accelerated and distributed HPC systems [Nyland et al. 2007, Zwart et al. 2007, Vanz et al. 2025a, Vanz et al. 2025b].

A direct-method N-body step decomposes naturally into two phases that alternate at every time step. In the *force-computation* phase, the position of every particle is read (a read-only dependency shared by all tasks) to compute the total force, and hence the acceleration, acting on each particle; the velocity array is updated accordingly (a write dependency, disjoint across tasks if velocities are partitioned into blocks). In the *position-integration* phase, the roles invert: velocities are read to update each particle’s position. This layout requires broadcasting a replicated array to every worker, partitioning and redistributing a second array, and synchronizing between phases, exercising exactly the data-coherence, transfer, and completion mechanisms that this article sets out to compare under distributed GPU execution.

3.2. Implementations using MPI, MPP and StarPU

We implemented solutions for the direct-method N-body problem described in Section 3.1 using MPI, MPP and StarPU. In all solutions, positions are replicated across all workers at every step while velocities are partitioned, and the force/integration loop runs entirely on GPUs, so the implementations differ mainly in who orchestrates data movement, placement, and synchronization.

- MPI.** The MPI implementation follows a conventional Single Program Multiple Data (SPMD) model. Each MPI rank holds a slice of the bodies, broadcasts the position array (`MPI_Bcast`), distributes velocities (`MPI_Scatterv`), partitions its slice across its local GPUs, and re-synchronizes positions at each step with `MPI_Allgatherv`; all communication, placement, and synchronization are explicit and programmer-controlled. We evaluate two variants that share this communication structure but differ in the GPU programming model: **MPI+OpenMP**, which offloads computation through OpenMP `target` regions, and **MPI+HIP**, which uses native HIP kernels.
- MPP.** The MPP version is a single OpenMP process with no MPI code: each OpenMP `target` device acts as a worker that MPP transparently maps onto a remote GPU. Since MPI collectives are not available, the data movement of the baseline is reimplemented with OpenMP `target` transfers (`omp_target_memcpy`), which MPP turns into the actual inter-node communication: the application keeps a pure OpenMP programming surface while the network is hidden behind the `liboffload` proxy layer, but it must manually encode the collective patterns as device transfers.
- StarPU.** The StarPU implementation expresses the simulation as a task graph instead of explicit communication, using the full-MPI mode. The arrays are registered as data handles and partitioned into tiles, and two codelets encode the force and integration phases with their data access modes; StarPU infers task dependencies, schedules tiles onto the local GPUs, and issues the inter-node transfers automatically under an owner-computes policy, so the application is rank-aware only when declaring ownership and inserting the graph. As with MPI, we evaluate **StarPU+OpenMP** and **StarPU+HIP** variants that share this task graph and differ only in the GPU programming model.

4. Experimental Evaluation

This section presents the experimental evaluation of MPI, MPP, and StarPU for distributed GPU execution. We first describe the experimental platform and the runtime configuration. Then, we detail the methodology shared by the Task Bench and N-Body experiments. Finally, we present and analyze the results obtained with each workload.

4.1. Experimental Setup

All experiments were conducted on the AMD University Program (AUP) AI and HPC cluster. Each compute node is equipped with two AMD EPYC 7V13 processors, providing a total of 128 CPU cores, and four AMD Instinct MI210 GPUs interconnected via AMD Infinity Fabric. Nodes communicate through NVIDIA ConnectX-6 MT28908 InfiniBand adapters providing up to 200Gb/s of bandwidth, and evaluations were performed on configurations ranging from 1 to 8 nodes, corresponding to a maximum of 32 GPUs. The software stack consisted of Clang 21.0, ROCm 7.2.0, StarPU 1.4.12, MPICH 4.3.2, and UCX 1.20 with GPU-aware communication enabled. All experiments were executed under Slurm with exclusive node allocation. To ensure reproducibility, the benchmarks were run using Spinner [Ceccato et al. 2024], with all software dependencies and runtime environments encapsulated in Apptainer containers. Each N-Body experiment was repeated five times and each Task Bench experiment three times. Reported results corre-

spond to the mean execution time, with 95% confidence intervals estimated using bootstrap resampling and represented as semi-transparent error bands in the figures. Individual runs deviated from the mean by less than 2% for MPI+OpenMP and MPP and by at most 21% for StarPU+HIP at 32 GPUs, whose larger dispersion is visible in the confidence bands. Runtime configuration follows the same rules across implementations: one process per node drives the node’s four GPUs, with the rank–GPU mapping fixed through `ROCR_VISIBLE_DEVICES`; application processes are bound to NUMA node 1 with `numactl` (the MPP proxy runs on NUMA node 0), and OpenMP host threads use scatter affinity with four threads per rank. MPP uses four execution and four data event handlers, 16 MPI communicators, and 2 GB communication fragments over 16 CH4 VCIs, with MPI global progress disabled; StarPU uses its default `dmda` scheduling policy with one velocity tile per GPU worker. N-Body runs use 2^{20} bodies. The experimental data and scripts are publicly available at the [project repository](#).

4.2. Experimental Methodology

This subsection describes the configuration of our two workloads: Task Bench, which characterizes runtime overhead under controlled dependency structures, and the N-Body simulation, which evaluates application-level strong scaling; Sections 4.3 and 4.4 present the results.

We quantify the impact of task structure and dependency patterns using Task Bench [Slaughter et al. 2020], a parameterized benchmark that evaluates parallel runtime systems through task graphs representing common dependency and communication patterns, with parameters controlling graph dimensions, dependency pattern, kernel type and duration, and the amount of data produced by each task. We employed three graph patterns: *trivial*, with independent tasks; *tree* with a hierarchical dependency structure; and *stencil_1d*, with regular nearest-neighbor dependencies. Examples of these graphs are shown in Figure 1. All patterns execute the same custom compute-bound GPU kernel [Cléto et al. 2025b] with ten steps and radix five. Overhead is quantified with the Minimum Effective Task Granularity at 50% efficiency, a.k.a METG(50%). For wall-clock time T , N_{GPU} GPUs, and N_{tasks} tasks, granularity is $G = \frac{T \times N_{\text{GPU}} \times 10^3}{N_{\text{tasks}}}$ ms per task per GPU. Efficiency is normalized to the maximum throughput for the same runtime, GPU count, output size, and pattern, and METG is the interpolated task granularity at which efficiency crosses 50%; lower values indicate finer-grained effective tasks.

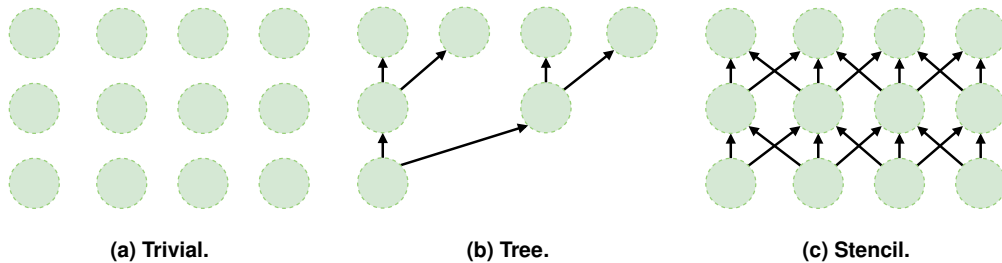


Figure 1. Task graph patterns used in the Task Bench experiments.

In this study, each runtime’s measured throughput is normalized to its maximum observed throughput for the same runtime, GPU count, output size, and pattern, so efficiency curves approach 100% for sufficiently coarse tasks. The METG(50%) is the task-granularity value at which the curve crosses the 50% efficiency threshold, estimated

by linear interpolation when the crossing falls between two measured points. The crossing gives the task size needed to amortize runtime overhead. Because the curves degrade at different granularities, each runtime requires its own trivial baseline for the relative comparison: we report relative $\text{METG}(p) = \frac{\text{METG}(p)}{\text{METG}(\text{trivial})}$ at 4, 16, and 32 GPUs under weak scaling, where values above one indicate a coarser task requirement for pattern p than for independent tasks.

MPI+OpenMP. The graph width is statically partitioned among MPI ranks. Each rank creates local output and input buffers for its assigned tasks, determines predecessor and successor tasks, and exchanges cross-rank dependencies with nonblocking `MPI_Isend/MPI_Irecv` operations followed by `MPI_Waitall`. Once its dependencies are available, the rank executes its local tasks with an OpenMP parallel loop and offloads the compute kernel to its GPUs with OpenMP target regions.

MPP. A single OpenMP process traverses the graph, assigns task outputs to devices, represents dependencies with OpenMP dependency objects, launches asynchronous target regions with `nowait`, and moves dependency data between devices with asynchronous target-memory copies; MPP converts these offload operations into communication with remote GPUs through its MPI proxy layer. Graph traversal, task generation, device assignment, and dependency encoding are thus performed explicitly by the benchmark implementation.

StarPU+OpenMP. The graph is represented using StarPU data handles: each task is inserted through StarPU’s MPI interface with read handles for its predecessors and a read-write handle for its output. StarPU constructs the distributed dependency graph, transfers data between ranks and GPU memories, and schedules the resulting codelets on HIP workers, using the same OpenMP target offloading code as the other implementations and delegating dependency resolution, data coherence, and task placement to the runtime.

We compare each runtime with its own trivial-graph baseline rather than ranking the implementations directly. The three runtimes are structurally different execution architectures, MPI distributes ranks that exchange messages explicitly, StarPU distributes scheduling decisions across all ranks, and MPP concentrates coordination in a single proxy process, but these differences are part of the phenomenon under study rather than a confounder: normalizing each runtime to its own baseline isolates dependency-pattern sensitivity from absolute overhead differences.

For the N-Body simulations, we performed a strong-scaling study with a fixed problem size of 2^{20} bodies on 4 to 32 GPUs (1 to 8 nodes), evaluating the implementations of Section 3.2: **MPI+OpenMP** and **MPI+HIP**, **StarPU+OpenMP** and **StarPU+HIP**, and **MPP** with its native OpenMP offloading model. MPP is not evaluated with a native HIP variant because its premise is to offload unmodified OpenMP target regions through `liboffload`: replacing the compiler-generated device kernel with handwritten HIP code would bypass the programming model MPP is designed to preserve. This design separates distributed-runtime effects from GPU kernel effects.

4.3. Task Bench Results

Figure 2 shows the relative METG comparison at 4, 16, and 32 GPUs. **MPI+OpenMP** exhibits the largest stencil penalty, with relative METG values of approximately 3.49, 3.11,

and 3.09 at 4, 16, and 32 GPUs, respectively, while its tree penalty increases from 1.35 to 1.90. This behavior is consistent with the explicit exchange and synchronization of cross-rank dependencies: the stencil graph’s regular-neighbor communication is much more expensive than independent tasks, while the tree graph produces a smaller but growing penalty. **MPP** is close to its trivial baseline at 4 GPUs, especially for the tree graph, whose relative METG is approximately 1.03. At higher GPU counts, the tree penalty grows to 2.16 and 2.72, while the stencil penalty remains between 1.26 and 1.61. MPP thus handles regular stencil dependencies more consistently than the hierarchical tree graph, whose centralized graph traversal, device coordination, and inter-node data movement become increasingly costly as the system grows. **StarPU+OpenMP** has the most stable stencil behavior, decreasing from approximately 1.25 at 4 GPUs to 1.14 at 32 GPUs. Its tree graph shows the opposite trend: near the trivial baseline at 4 GPUs but increasing to approximately 2.04 at 16 GPUs and 3.44 at 32 GPUs, suggesting that StarPU’s data-handle scheduling manages regular stencil dependencies effectively, whereas the growing irregularity of the tree graph creates a larger scheduling and data-management penalty. These results should be interpreted as dependency sensitivity within each runtime, not as direct evidence that one runtime is universally faster than another.

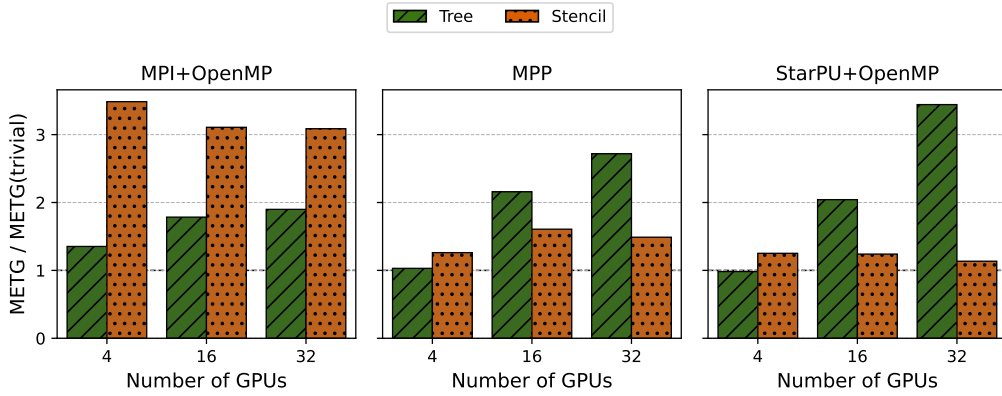


Figure 2. Relative Task Bench METG(50%), normalized to the trivial graph for each runtime and GPU count.

The absolute METG values complement this relative analysis, which by construction cannot establish which runtime incurs the lowest overhead. For the trivial graph, MPP sustains the finest effective tasks (14–17 ms), followed by MPI+OpenMP (15–20 ms) and StarPU (19–20 ms). For the stencil graph, MPI+OpenMP requires 53–62 ms, against 18–25 ms for MPP and 22–24 ms for StarPU. For the tree graph, METG grows with the GPU count for all runtimes, from 21 to 38 ms for MPI+OpenMP, from 15 to 46 ms for MPP, and from 19 to 67 ms for StarPU, confirming the hierarchical pattern as the most demanding dependency structure for runtime-managed and proxy-based coordination alike.

4.4. N-Body Results

Figure 3a reports the mean execution time of each implementation as the number of GPUs increases. The results show a consistent separation between the HIP- and OpenMP-based versions, and all speedups fall well short of the ideal $8\times$ from 4 to 32 GPUs. MPI+HIP delivers the best overall performance, dropping from approximately 6.5 s on four GPUs to 2.1 s on 32 GPUs ($3.2\times$, 40% scaling efficiency), while StarPU+HIP follows closely,

from 6.6 s to 2.4 s ($2.8\times$, 35%), indicating that the task-based runtime adds only a small overhead over explicit message passing when kernels are native. Among the OpenMP-based versions, MPI+OpenMP decreases from 15.1 s to 3.9 s ($3.9\times$, 49%), MPP from 15.1 s to 4.3 s ($3.5\times$, 44%), and StarPU+OpenMP from 15.2 s to 4.4 s ($3.5\times$, 44%), with MPI+OpenMP about 0.4 s faster than MPP at 32 GPUs.

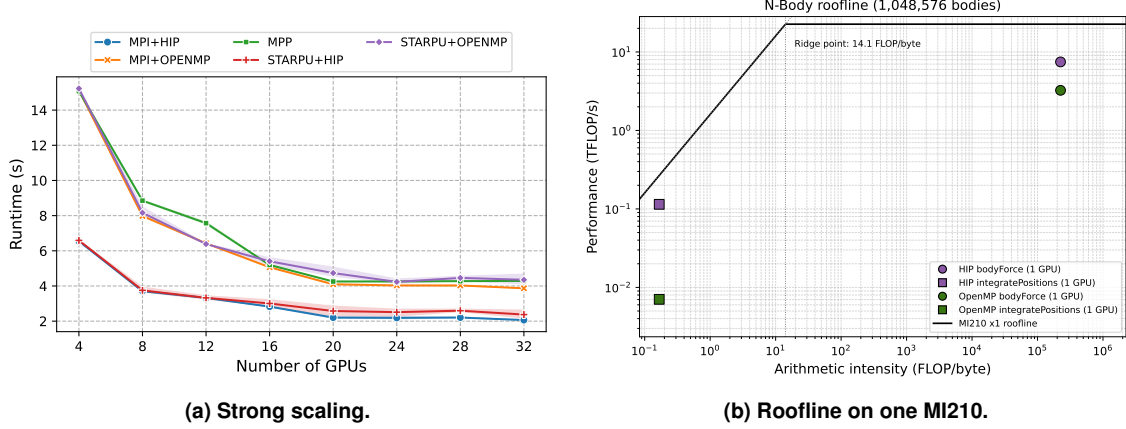


Figure 3. N-Body results: (a) execution time versus number of GPUs for each implementation and variant; (b) roofline analysis of the HIP and OpenMP N-Body GPU kernels on one AMD MI210.

The curves stabilize beyond 20 GPUs. From 20 to 32 GPUs, MPI+OpenMP improves by only 6% (4.10 s to 3.87 s), the StarPU variants fluctuate within their confidence bands, and MPP’s runtime slightly increases (4.26 s to 4.29 s). This saturation reflects the strong-scaling regime: per-GPU work falls as $\mathcal{O}(n^2/G)$ while every step still requires broadcasting the full position array and gathering velocities, so communication and synchronization increasingly dominate. For MPP, the effect is amplified by its centralized control path, in which all offload coordination funnels through a single host process, consistent with both the slight regression beyond 20 GPUs and the tree-graph degradation in Task Bench.

The strong-scaling results combine runtime, communication, and kernel effects. To isolate the latter, we analyzed the two GPU kernels used by the N-Body implementation, `bodyForce` and `integratePositions`, in their HIP and OpenMP target versions. The profiles use $2^{20} = 1,048,576$ bodies and ten iterations, with each dispatch computing 2^{18} targets (one GPU’s partition of the bodies) from the full source array. Kernel performance is computed from the total floating-point operations divided by the sum of the selected kernel dispatch durations, rather than the interval between first and last dispatch, which would include inter-iteration gaps and the other kernel’s time. For one interaction, the `bodyForce` model counts 18 FLOPs plus six velocity-update FLOPs per target body, giving $F_{\text{bodyForce}} = 18N_{\text{target}}N_{\text{bodies}} + 6N_{\text{target}}$; `integratePositions` performs $F_{\text{integrate}} = 6N_{\text{target}}$. Each structure is 12 bytes. Counting reused source positions once per dispatch gives $B_{\text{bodyForce}} = 12N_{\text{bodies}} + 36N_{\text{target}}$ and $B_{\text{integrate}} = 36N_{\text{target}}$. Arithmetic intensity is $I = F/B$; this is an effective traffic model, not a direct HBM-counter measurement.

Figure 3b shows the measured points against the roofline of one AMD MI210, using a peak FP32 throughput of 22.6 TFLOP/s and HBM2e bandwidth of 1.6

TB/s [AMD 2022], for a ridge point of approximately 14.1 FLOP/byte. With $N_{\text{target}} = 2^{18}$ and $N_{\text{bodies}} = 2^{20}$, the `bodyForce` points have an arithmetic intensity of approximately 2.2×10^5 FLOP/byte, far to the right of the ridge point, and are therefore compute-bound under this model. HIP reaches approximately 7.47 TFLOP/s, while OpenMP reaches 3.24 TFLOP/s, so the HIP kernel delivers about $2.3\times$ the measured throughput of the OpenMP target kernel, explaining a substantial part of the separation between the HIP- and OpenMP-based N-Body implementations in Figure 3a.

In contrast, `integratePositions` has an arithmetic intensity of approximately 0.167 FLOP/byte and lies in the bandwidth-bound region of the plot. Its measured throughput is much lower than that of `bodyForce`, approximately 0.114 TFLOP/s for HIP and 0.007 TFLOP/s for OpenMP, but it performs only a small amount of work per body and does not dominate the quadratic N-Body computation.

5. Related Work

Comparisons between explicit message passing and shared-memory programming models have long been used to characterize the trade-off between programmer control and runtime convenience: Sharma and Kanungo compare pure MPI with hybrid MPI+OpenMP on a multicore cluster, showing that the relative performance depends on the communication and computation characteristics of the workload [Sharma and Kanungo 2011]. More recent work considers task-based execution as an alternative to both approaches; StarPU is a foundational example, representing computations as tasks over data handles and scheduling work across heterogeneous processing units [Augonnet et al. 2010].

Several studies have directly compared task-based runtimes. Miletto and Schnorr evaluated GCC/libgomp, LLVM/libomp, and StarPU for task-based dense block QR factorization: the relative advantage depends on task granularity and graph size, with native OpenMP runtimes competitive for some configurations and StarPU providing lower task creation and submission overhead and better scalability for larger task graphs [Miletto and Schnorr 2019]. Agullo et al. bridged OpenMP with a task-based runtime for the fast multipole method, combining the OpenMP interface with data-flow scheduling and heterogeneous execution [Agullo et al. 2017], and Ferat et al. extended MPI+OpenMP task-based applications to heterogeneous architectures with GPU support [Ferat et al. 2022]. These works establish that programming-model comparisons must account not only for kernel performance, but also for task creation, dependency management, data movement, and communication overhead. Our results confirm and extend these observations.

Task Bench [Slaughter et al. 2020] itself evaluated more than a dozen runtime systems, including MPI, MPI+OpenMP, OmpSs-2, PaRSEC, Legion/Regent, and StarPU, but on multicore and distributed CPU platforms, where device-side kernel implementation plays no role. We apply the same METG-based methodology to distributed GPU execution, where the choice between OpenMP target kernels and native HIP kernels adds a dimension absent from CPU comparisons: the $2.3\times$ kernel gap dominates the runtime differences among implementations. The granularity dependence reported by Miletto and Schnorr and by Task Bench reappears in our relative-METG analysis: dependency patterns penalize each runtime differently as the system grows, with hierarchical trees degrading MPP and StarPU coordination, while explicit message passing is penalized

mostly by regular stencil communication.

This work also differs from our own previous evaluations in scope and in what is measured. Vanz et al. evaluated N-Body simulations with StarPU, OpenMP, and MPI on cloud infrastructure [Vanz et al. 2025a, Vanz et al. 2025b], but without MPP, a native-HIP baseline, or a dependency-pattern study; Cléto et al. evaluated MPP’s transparent remote offloading on GPU clusters [Cléto et al. 2025a, Cléto et al. 2025b], but without a task-based counterpart on the same hardware and workloads.

6. Conclusion

This paper compared MPI, the MPI Proxy Plugin (MPP), and StarPU for distributed GPU execution using a common N-Body workload and Task Bench. The N-Body strong-scaling results show that native kernel performance is the dominant factor: MPI+HIP was the fastest ($3.2\times$ from 4 to 32 GPUs), followed closely by StarPU+HIP ($2.8\times$), while the OpenMP-based variants scaled similarly but reached higher execution times. The roofline analysis explains this gap: the compute-bound `bodyForce` kernel reached 7.47 TFLOP/s with HIP versus 3.24 TFLOP/s with OpenMP, approximately a $2.3\times$ difference, whereas the low-intensity integration kernel did not dominate the application. All implementations stabilize beyond 20 GPUs, with MPP’s centralized proxy coordination showing a slight regression at the largest configurations.

The Task Bench results framed dependency sensitivity within each implementation: relative METG showed that MPI+OpenMP incurred the largest stencil penalty, MPP remained relatively stable for stencil but degraded for the tree graph at higher GPU counts, and StarPU handled stencil dependencies consistently while showing the largest tree penalty at 32 GPUs. These results highlight complementary trade-offs: MPI+OpenMP favors explicit control, MPP favors transparent remote offloading and OpenMP compatibility, and StarPU favors runtime-managed dependencies and data coherence, with no runtime uniformly superior across graph structures.

This study is limited to one direct N-Body workload, one GPU architecture, and one cluster configuration, and the relative-METG analysis does not constitute an absolute runtime ranking. Future work includes applications with more irregular communication, larger clusters, and additional GPU architectures, as well as computation–communication overlap in MPP’s centralized control path and compiler optimizations in the OpenMP target backend.

7. Acknowledgments

This work is supported by the São Paulo Research Foundation (FAPESP), Grants No. 2025/27771-4 and 2025/09515-0. This work was also supported in part by Advanced Micro Devices, Inc. under the AMD University Program’s AI & HPC Cluster, *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq) and *Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina* (FAPESC).

References

Agullo, E., Aumage, O., Bramas, B., Coulaud, O., and Pitoiset, S. (2017). Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2794–2807.

- AMD (2022). AMD Instinct MI210 Accelerators Specifications. Product specifications page.
- AMD (2026). Introduction to the HIP programming model. ROCm Documentation. Accessed: 2026-08-14.
- Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2010). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.
- Barnes, J. and Hut, P. (1986). A hierarchical $O(n \log n)$ force-calculation algorithm. *Nature*, 324(6096):446–449.
- 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.
- Cléto, J., Valarini, G., Pereira, M. M., Araujo, G., and Yviquel, H. (2025a). Scalable OpenMP Remote Offloading via Asynchronous MPI and Coroutine-Driven Communication. In Nagel, W. E., Goehringer, D., and Diniz, P. C., editors, *Euro-Par 2025: Parallel Processing*, pages 254–267, Cham. Springer Nature Switzerland.
- Cléto, J., Valarini, G., and Yviquel, H. (2025b). Can OpenMP Scale Beyond the Node? A Performance Evaluation of Remote Offloading via the MPI Proxy Plugin. In *Anais do XXVI Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 278–289, Porto Alegre, RS, Brasil. SBC.
- Ferat, M., Pereira, R., Roussel, A., Carribault, P., Steffemel, L.-A., and Gautier, T. (2022). Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support. In *OpenMP in a Modern World: From Multi-Device Support to Meta Programming*, Lecture Notes in Computer Science, pages 3–16. Springer.
- Heggie, D. (2005). The Classical Gravitational N-Body Problem. *Encyclopedia of Mathematical Physics*.
- Miletto, M. C. and Schnorr, L. M. (2019). OpenMP and StarPU Abreast: the Impact of Runtime in Task-Based Block QR Factorization Performance. In *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD 2019)*, pages 25–36.
- Netto, M. A. S., Calheiros, R. N., Rodrigues, E. R., Cunha, R. L. F., and Buyya, R. (2018). HPC Cloud for Scientific and Business Applications: Taxonomy, Vision, and Research Challenges. *ACM Computing Surveys*, 51(1).
- Nyland, L., Harris, M., and Prins, J. (2007). Fast N-Body Simulation with CUDA. In Nguyen, H., editor, *GPU Gems 3*, chapter 31. Addison-Wesley.
- Sharma, R. and Kanungo, P. (2011). Performance Evaluation of MPI and Hybrid MPI+OpenMP Programming Paradigms on Multi-Core Processors Cluster. In *2011 International Conference on Recent Trends in Information Systems*, pages 137–140.
- 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., Treichler, S., McCormick, P., and Aiken, A. (2020). Task Bench: A Parameterized Benchmark for Evaluating Par-

- allel Runtime Performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.
- Vanz, N., Munhoz, V., and Castro, M. (2025a). Performance Evaluation of N-Body Simulations on AWS with StarPU, OpenMP and MPI Runtime Systems. In *Anais do XXVI Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 61–72, Porto Alegre, RS, Brasil. SBC.
- Vanz, N., Munhoz, V., Castro, M., Lima Pilla, L., and Aumage, O. (2025b). Task-Based HPC in the Cloud: Price-Performance Analysis of N-Body Simulations with StarPU. In *IC2E 2025 - 13th IEEE International Conference on Cloud Engineering*, Rennes, France.
- Yviquel, H., Pereira, M., Franceschini, E., Valarini, G., Leite, G., Rosso, P., Ceccato, R., Cusihualpa, C., Dias, V., Rigo, S., Souza, A., and Araujo, G. (2023). The OpenMP Cluster Programming Model. In *Workshop Proceedings of the 51st International Conference on Parallel Processing*, New York, NY, USA.
- Zwart, S. F. P., Belleman, R. G., and Geldof, P. M. (2007). High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy*, 12(8):641–650.