ICICSET-2026

https://doi.org/10.65091/icicset.v3i1.88

Crossref

Heterogeneous Distributed Parallel Processing via gRPC with Zero-Configuration UDP Worker Discovery

Roshika Chitrakar*
Department of Computer Science and Engineering
Kathmandu University
Dhulikhel, Kavre
chiroshika@gmail.com

Suvam Karn
Department of Computer Science and Engineering
Kathmandu University
Dhulikhel, Kavre
suvamkarn49@gmail.com

Aryan Mahato
Department of Computer Science and Engineering
Kathmandu University
Dhulikhel, Kavre
mahatoaryan02@gmail.com

Sakar Maharjan
Department of Computer Science and Engineering
Kathmandu University
Dhulikhel, Kavre
sakarmaharjan123@gmail.com

Pukar Adhikari
Department of Computer Science and Engineering
Kathmandu University
Dhulikhel, Kavre
pukar.adhikari656@gmail.com

Bipesh Raj Subedi
Department of Computer Science and Engineering
Kathmandu University
Dhulikhel, Kavre
bipesh.subedi@ku.edu.np

*Corresponding author

Abstract*—Modern computational workloads, including large-scale Monte Carlo simulations and distributed machine learning training, demand parallelism that exceeds the capacity of a single machine. Existing distributed computing frameworks like Apache Spark and Berkeley Open Infrastructure for Network Computing (BOINC) require significant infrastructure overhead, static network configuration, and homogeneous runtime environments, limiting their accessibility in ad-hoc settings such as student laboratories or home networks. This paper proposes a zero-configuration heterogeneous distributed parallel processing framework based on a three-tier master-worker architecture, wherein worker nodes autonomously discover and register with a central orchestrator via a User Datagram Protocol (UDP) broadcast beaconing protocol, eliminating all manual network configuration. Task payloads comprising executable logic, data shards, and model state are dispatched to worker nodes via Google Remote Procedure Call (gRPC) over HTTP/2 using Protocol Buffers (Protobuf) binary serialization, achieving a significant payload size reduction compared to equivalent JSON representations. Worker liveness is maintained through an asynchronous heartbeat mechanism with automatic shard re-assignment upon node failure, providing fault tolerance with zero manual intervention. The framework is validated through a 200,000,000-trial Geometric Brownian Motion (GBM) Monte Carlo simulation distributed across a three-node heterogeneous cluster, specifically chosen to test extreme cross-platform heterogeneity. A superlinear speedup of 4.29x was observed over single-node execution with a measured parallel efficiency of 143%, attributable to the aggregate multi-core advantage of the heterogeneous worker pool. The proposed system demonstrates that enterprise-grade distributed computing capabilities can be realized without dedicated infrastructure, offering a practical and extensible platform for parallel scientific computation and distributed machine learning.

Keywords—distributed computing, parallel processing, gRPC, Protocol Buffers, UDP broadcast discovery, master-worker architecture, Monte Carlo simulation, fault tolerance, heterogeneous cluster

I. INTRODUCTION

The exponential growth of data-intensive computational workloads in scientific simulation and machine learning has created an increasing demand for distributed parallel processing systems capable of harnessing the aggregate computational capacity of multiple independent machines. [1] A single node, regardless of its specifications, imposes a fundamental ceiling on throughput, creating a bottleneck for intensive tasks such as large-scale Monte Carlo simulations where the volume of computation exceeds the limits of a single machine. [2]

Traditional High-Performance Computing (HPC) infrastructure, while capable of addressing this demand, requires significant capital investment in dedicated server hardware, specialized cluster management software, and administrative expertise that places it beyond the reach of academic research groups and small-scale development teams. [3], [4] Meanwhile, pools of underutilized commodity hardware such as student laptops and laboratory workstations remain an untapped resource for parallel processing.

Volunteer computing platforms such as BOINC have demonstrated that idle consumer hardware can be harnessed for scientific computation at scale, but these platforms are designed for internet-scale participation and impose rigid software requirements, centralized job management, and homogeneous execution environments that are unsuitable for local, ad-hoc deployment scenarios. [5] Contemporary data processing frameworks including Apache Spark and Hadoop MapReduce provide powerful abstractions for distributed computation but require dedicated cluster management infrastructure including services such as Apache ZooKeeper or Yet Another Resource Negotiator (YARN) and assume static network configurations where the address of every node is known in advance. [6], [7] These constraints are often incompatible with dynamic environments like student laboratories, where IP addresses frequently change due to Dynamic Host Configuration Protocol (DHCP) re-

assignments and nodes fluctuate between online and offline states across variable local network links. [8]

A major unresolved challenge is the heterogeneity of hardware and operating systems in these ad-hoc settings. Existing frameworks struggle with cross-platform compatibility and the network isolation introduced by virtualized environments like Windows Subsystem for Linux 2 (WSL2). Specifically, the Network Address Translation (NAT) used in virtualized stacks often blocks standard broadcast signals, making traditional discovery mechanisms non-functional. No existing lightweight distributed framework addresses this combination of zero-configuration discovery, cross-platform heterogeneity, and virtualized network compatibility simultaneously.

This paper proposes a zero-configuration heterogeneous distributed parallel processing framework that directly addresses the identified limitations through a three-tier master-worker architecture requiring no dedicated infrastructure beyond a shared local area network. The proposed framework transforms a pool of ordinary consumer laptops running heterogeneous operating systems into a functional distributed compute cluster without any manual configuration.

The framework resolves the network discovery problem through an autonomous UDP broadcast beaconing protocol wherein the central orchestrator node periodically announces its presence across the local subnet. For virtualized environments where standard broadcast propagation is obstructed by NAT, the framework implements a complementary parallel subnet scanning fallback that locates the Orchestrator through concurrent lightweight probing of all addresses on the local subnet.

Worker liveness is maintained through an asynchronous heartbeat mechanism modeled on decentralized fault detection schemes. [10] A dedicated background monitor detects unresponsive nodes and automatically reassigns their incomplete shards to healthy workers, providing fault tolerance that persists through hardware failures such as laptop sleep events and transient network drops.

Task payloads are serialized using Protobuf binary encoding and dispatched to worker nodes via gRPC over HTTP/2, reducing the parsing overhead and latency associated with traditional text-based interfaces. [9]

The use of gRPC as a high-performance Remote Procedure Call (RPC) framework for distributed systems has been extensively demonstrated in production environments at Google, Netflix, and other large-scale technology organizations. [6] Comparative networking benchmarks confirm that gRPC provides significant throughput and latency advantages over traditional RESTful HTTP (REST= Representational State Transfer) interfaces. [9] gRPC's foundation on HTTP/2 provides multiplexing, binary framing, and persistent connection management that are directly leveraged in the proposed framework's task dispatch layer.

II. Related Work

The design of the proposed framework draws from several established bodies of work in distributed systems, parallel computing, and network communication protocols.

The foundational data parallelism model was established by Dean and Ghemawat, whose MapReduce framework demonstrated that large-scale computations can be expressed as independent map operations followed by a centralized reduce aggregation, which enables distribution across commodity hardware clusters. [7] The proposed framework extends this principle to support arbitrary executable payloads rather than fixed map-reduce functions, allowing a broader class of computational workloads.

Anderson demonstrated through the BOINC platform that idle computational resources, including personal computers and laptops, represent a significant untapped source of parallel processing capacity. [5] The proposed framework shares this motivation but targets local area networks rather than internet-scale volunteer grids, trading global scale for low-latency communication and simplified deployment.

Zaharia et al. introduced Apache Spark as a unified engine for big data processing, supporting in-memory computation for iterative algorithms. [6] While Spark provides powerful distributed processing abstractions, it requires a dedicated cluster manager and does not support arbitrary executable payload dispatch, which limits its applicability to predefined computation patterns. Dynamic task graphs and modern cluster runtime frameworks like Ray have emerged to address these computational limits for artificial intelligence workloads. But they remain tailored to stable, enterprise-scale server installations rather than ad-hoc consumer setups. [11]

In the domain of heterogeneous resource management, systems like StarPU have explored runtime scheduling across mixed processing architectures. [4] However, such tools presuppose kernel-level instrumentation and direct operational control, which are rarely available across consumer operating systems.

Decentralized systems such as Petals and Horovod demonstrate scalable training and inference across heterogeneous nodes, but they are built around internet-scale peer-to-peer swarms or static multi-GPU servers, rather than dynamic, zero-configuration local networks. [12], [13]

III. Methodology

To ensure platform independence, resilience, and operational efficiency, the proposed framework is defined through formal system architecture, protocol schemas, deterministic state machines, execution procedures, algorithms, and mathematical formulations, without reference to any specific programming language or runtime environment.

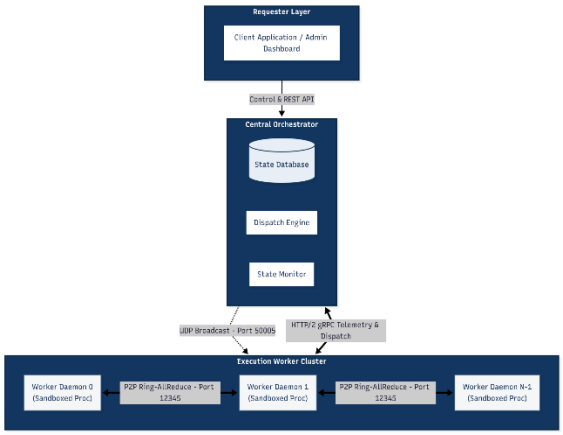


Fig. 1. Three-tier heterogeneous distributed processing architecture

A. *System Architecture and Core Design Principles*

The framework adopts a three-tier hybrid Orchestrator-Worker topology (illustrated in Fig.1) designed to decouple control orchestration from intensive computational execution. The architecture consists of three layers: Requester Layer for job submission, a Central Orchestrator for state management and task dispatch, and an Execution Worker Cluster comprising heterogeneous compute nodes.

Rather than relying on a monolithic communication stack, the architecture enforces strict design principles by separating network functions into four isolated logical planes:

- Autonomous Discovery Channel (UDP, Port 50005): Zero-configuration dynamic broadcast discovery without static IP configuration.
- Event-Driven Control Plane (HTTP REST): Manages asynchronous heartbeat telemetry and job lifecycle states, ensuring fault tolerance and dynamic recovery without manual intervention.
- High-Throughput Data Plane (HTTP/2 gRPC, Port 50051): Enforces a strictly-typed binary serialization contract (Protobuf) for task dispatch, replacing text-based formats to minimize payload overhead across heterogeneous environments.
- Dynamic Topology Switching (P2P, Port 12345): While task scheduling defaults to a master-worker topology, the architecture dynamically shifts to a decentralized Ring-AllReduce topology during distributed machine learning, which enables direct inter-worker parameter synchronization and eliminates Central Orchestrator bottlenecks.

B. *Automatic Node Discovery and Hardware Registration*

The primary challenge in establishing an ad-hoc distributed cluster is the initial discovery of the Orchestrator by the Worker nodes without the fragility of manual IP configuration. The proposed framework resolves this through a continuous, two-phase handshake that transitions from a stateless connectionless beacon to a stateful binary session (illustrated in Fig. 2).

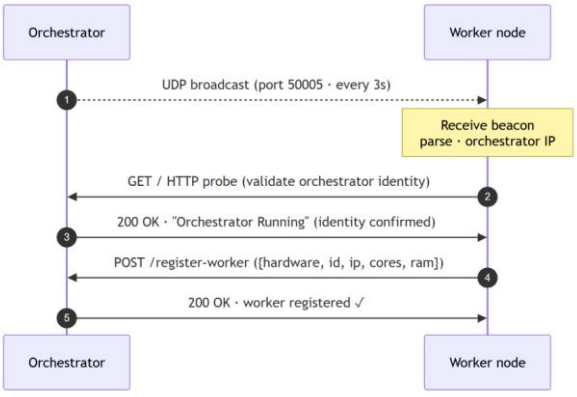


Fig. 2. Discovery and registration sequence

1) *Connectionless Beaconing (Discovery)*

To ensure a "Zero-Configuration" user experience, the Orchestrator acts as a network beacon, broadcasting its presence over UDP port 50005. UDP was selected over TCP as it permits one-to-many propagation across the local subnet without requiring pre-established handshakes. To maximize deterministic reachability and bypass security filters that block limited broadcasts (255.255.255.255), the Orchestrator programmatically calculates and targets the Directed Subnet Broadcast address (e.g., 192.168.1.255).

A significant hurdle in heterogeneous grids is the "Virtual Wall" introduced by NAT in virtualized environments like WSL2, which isolates the kernel and blocks standard Layer 2 broadcasts. The framework overcomes this by mandating Mirrored Networking Mode on the host, granting the Linux-based worker direct visibility into the physical network interface's broadcast frames.

2) *Capability-Aware Fingerprinting (Registration)*

Following the successful capture of the Orchestrator's identity via the discovery layer, the system initiates a formal TCP-based REST registration handshake to deliver its hardware manifest. This phase transforms an unmapped node into a managed resource within the cluster's dynamic state machine.

To prevent naming collisions and "Identity Theft" common in heterogeneous clusters, the protocol generates a Unique Hardware Fingerprint by hashing the Media Access Control (MAC) address of the primary active NIC into a Universally Unique Identifier (UUID). This guarantees a persistent 1:1 mapping in the Orchestrator's dynamic registry, allowing the system to track nodes even across DHCP lease renewals.

The registration payload functions as a comprehensive resource audit, detailing the operating system architecture (Windows, Linux, macOS), core counts, and available RAM. This hardware profile empowers the Orchestrator to construct a master capacity map, enabling intelligent, resource-aware task sharding during workload execution.

The registration signal triggers an update to an in-memory Worker Registry in the Orchestrator's central registry. Each entry in the registry is a state object containing the node's unique ID, current IP address, and capability manifest.

The discovery algorithm is formalized as follows:

Algorithm 1: Autonomous Orchestrator Discovery

Input: Network interface, broadcast port, beacon interval *T_b*, receive timeout *T_w*, fallback timeout *T_fallback*
Output: Valid Orchestrator IP address
Initialization:
    ip ← *get_local_network_address*()
    listen(port=50005)
    t_start ← *current_time*()
Orchestrator:
1: Loop every *T_b* seconds:
2: *broadcast*("ORCHESTRATOR:"*ip*, *port*=50005)
Worker:
3: Loop:
4: *msg* ← *receive_broadcast*(*timeout*=*T_w*)
5: **if** *msg matches* "ORCHESTRATOR:"*ip*:
6: **if** *validate*(*ip*, *port*=8000) == true:
7: **return** *ip*
8: (*current_time*() - *t_start* >*T_fallback*)
9: **AND** *virtual_env* == true:
10: **return** *parallel_subnet_scan*(*gateway*, *port*=8000)

C. Job Management and Task Execution

To efficiently distribute workloads and maintain a clear separation of concerns, the framework defines distinct control and data planes. The control plane handles job management, worker coordination, monitoring, and state transitions, whereas the data plane is responsible for transferring task payloads and execution results.

1) Job Lifecycle and Task Dispatch

Workload execution is governed by a finite state machine managed by the Orchestrator. Each submitted job progresses through predefined states according to worker availability (illustrated in Fig. 3), execution outcomes, and fault conditions.

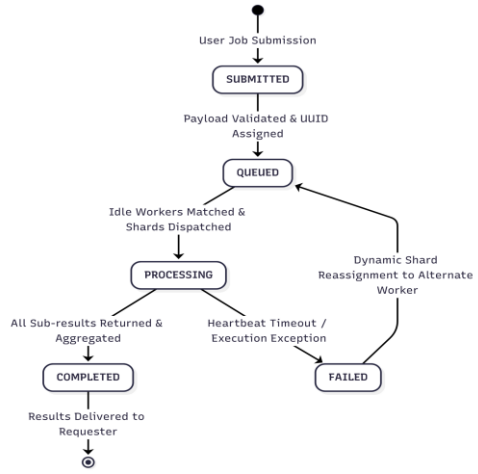


Fig. 3. State Transition Diagram

Once a job enters the ASSIGNED state, its payload is transmitted using a Unary gRPC communication pattern. This synchronous request-response model is suitable for batch-oriented parallel execution because transmission or execution failures can be detected explicitly, allowing the Orchestrator to re-queue the affected shard without leaving an untracked task.

The job-dispatch procedure is formalized as follows.

Algorithm 2: Job Dispatch

Input: *task_definition*, *data_payload*, *model_state*
Output: *job_id* and updated job status/result
Initialization:
    job_dir ← isolated job directory
    job_id ← *generate_uuid*()
    job.status ← *QUEUED*
Persist artifacts to isolated *job_dir*
W ← *find_idle_worker*()
if *W* == NULL:
    return *job_id* (*stays QUEUED*)
job.status ← *ASSIGNED*
W.current_job ← *job_id*
In background thread:
    payload ← *serialize*(task, *data*, *model_state*)
    result ← *dispatch_rpc*(*W.address*, *payload*)
    if *result*.success:
        job.status ← *COMPLETED*
        store(*result*.output)
    else:
        job.status ← *FAILED*
        log(*result*.error)
        requeue(*job_id*)

2) Communication Contract and Data Serialization

The communication contract is defined using a .proto file that serves as the single source of truth for communication among cluster components. The file specifies both the Service Schema, which defines the available remote procedures, and the Message Schema, which defines the structures exchanged between nodes.

The communication contract is compiled into native code for the Python-based Orchestrator and the Android dashboard, ensuring that all components use the same definitions for structures such as TaskPayload and TaskResult. This prevents serialization mismatches that can otherwise occur in heterogeneous distributed environments.

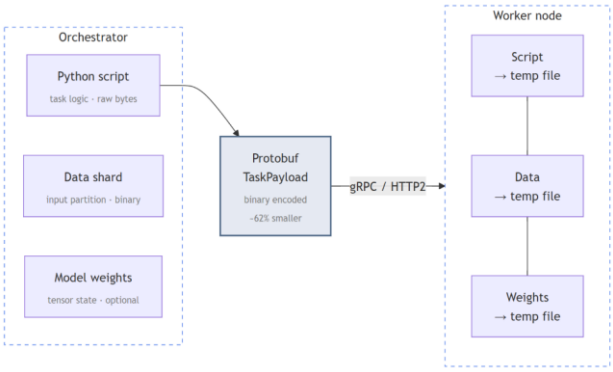


Fig. 4. Data Serialization Pipeline

As illustrated in Fig. 4, the Orchestrator employs an atomic packaging strategy in which the instructional logic, sharded data, and model state are encapsulated into a single TaskPayload object. The instructional logic is represented as Python bytes, while the data shards and model state contain the corresponding binary data and neural-network weights.

Google Protobuf is used to encode this communication contract. Protobuf employs field tagging and Varint encoding for compact metadata representation. Empirical implementation measurements demonstrate approximately a 62% reduction in payload size compared with equivalent JSON structures, while avoiding the approximately 33% overhead associated with Base64 encoding of binary weights in conventional REST-based data transfer.

While REST over HTTP/1.1 is sufficient for low-frequency operations such as heartbeat transmission and worker management, it is less suitable for the data plane, where large binary objects such as AI model weights and datasets must be transmitted. Furthermore, the framework employs gRPC as the primary data-plane communication mechanism, built on HTTP/2. gRPC supports persistent TCP connections and binary framing, reducing connection-establishment overhead during sequential task execution while providing efficient request-response communication.

A common limitation in standard RPC implementations is a default message size limit (typically 4MB), which is insufficient for machine learning, where modern neural network weights often exceed 100MB. To accommodate the large payloads required for distributed machine-learning

workloads, the framework configures the maximum RPC message length to 500 MB to support large machine-learning workloads.

The framework also utilizes HTTP/2 multiplexing to allow multiple logical streams to share a persistent connection. Consequently, control-related messages can be transmitted concurrently with large data transfers over the same connection rather than requiring separate connections for each operation.

3) Isolated Task Execution and Timeout Control

Executing remotely submitted Python scripts introduces potential stability and security risks, particularly when tasks consume excessive resources, terminate unexpectedly, or enter indefinite execution states. To address these conditions while retaining access to available hardware accelerators, each worker implements a process-level fault-isolation model.

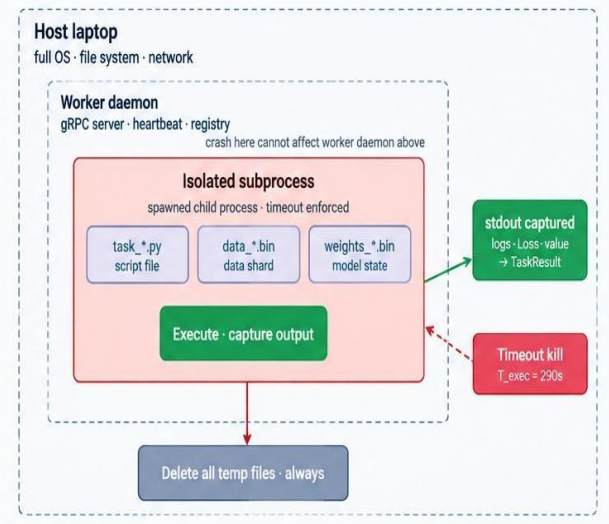


Fig. 5. Execution sandbox architecture

Tasks are staged using Job_ID and Shard_Index identifiers and executed in dedicated child processes to isolate failures and avoid file collisions. If the submitted script crashes, leaks memory, or enters an infinite loop, the failure remains isolated within the child process while the main worker daemon continues to handle orchestrator communication and telemetry.

While the application of WebAssembly as a portable execution environment for distributed computation has been explored by Haas et al., who demonstrated near-native performance for compiled binary modules across heterogeneous platforms, this framework uses native Python subprocesses. [7] This approach therefore provides flexibility for executing interpreted scripts and permits direct access to the scientific Python ecosystem (NumPy, TensorFlow, and PyTorch) and host hardware resources, including NVIDIA CUDA cores for tensor acceleration, while providing less comprehensive security isolation than a fully virtualized execution environment.

To prevent indefinitely running tasks from consuming CPU or GPU resources and reducing cluster availability, the framework implements a dual-layer timeout mechanism.

- Transport-Layer Timeout (Network Deadline): The gRPC client is configured with a strict network deadline. If the deadline is exceeded, the Orchestrator terminates the corresponding communication attempt and re-queues the task.
- Application-Layer Timeout (Subprocess Execution): The worker implements a slightly shorter execution timeout using the Python subprocess mechanism. If the submitted script exceeds the execution limit, the worker terminates the child process and returns a structured timeout failure to the Orchestrator before the network-level deadline is reached.

This hierarchical timeout strategy provides fault handling at both the network and local execution levels, preventing a single unresponsive task from blocking the broader heterogeneous cluster.

The isolated execution procedure is formalized as follows.

Algorithm 3: Isolated Task Execution

Input: TaskPayload *P*
Output: TaskResult
Initialization:
task_file ← *task_{job_id}_{shard_index}.py*
data_file ← *data_{job_id}_{shard_index}.bin*
weights_file ← *weights_{job_id}_{shard_index}.bin*

```
Write P.task_script file: task_{job_id}_{shard_index}.py
Write P.data_shard file: data_{job_id}_{shard_index}.bin
Write P.model_state file: weights_{job_id}_{shard_index}.bin
process ← spawn_child(
    cmd=[interpreter, script, data, weights],
    timeout=T_exec,
    capture_output=true
)
if process.exit_code == 0:
    loss ← extract_pattern("LOSS:", process.stdout)
    return TaskResult(success=true, output=stdout, loss=loss)
if timeout_exceeded:
    return TaskResult(success=false, error=Execution timeout)
else:
    return TaskResult(success=false, error=process.stderr)
finally:
    delete task_{job_id}_{shard_index}.py
    delete data_{job_id}_{shard_index}.bin
    delete weights_{job_id}_{shard_index}.bin
```

4) Fault Tolerance, Worker Liveness, and State Recovery

To maintain availability in a volatile, heterogeneous environment, the framework uses continuous worker-liveness monitoring and automated task recovery. Active workers periodically push a lightweight heartbeat containing CPU, RAM, and NVIDIA GPU vitals to the Orchestrator's REST interface at a fixed interval.

The Orchestrator logs each worker's latest timestamp and uses a background monitoring process to evaluate liveness. This push-based model detects failures independently of task execution without requiring continuous polling.

The fault-detection and recovery procedure is formalized as follows.

Algorithm 4: Fault Detection and Recovery

Input: Worker registry, job registry, heartbeat information, monitoring interval *T_m*, offline threshold *T_threshold*
Output: Updated worker status and reassigned jobs
Initialization:
Monitor loop ← executes every *T_m* seconds
Worker registry ← current registered workers
Job registry ← current active jobs

```
for each worker W in registry:
    elapsed ← current_time - W.last_seen
    if elapsed >T_threshold:
        W.status ← OFFLINE
        jobs ← find_running_jobs(W)
        for each job J in jobs:
            J.status ← QUEUED
            J.worker ← NULL
            W_new ← find_idle_worker()
            if W_new exists:
            assign(J, W_new)
            update_cluster_topology(W_new)
        end for
end for
Heartbeat sender (worker, every T_h seconds):
    vitals ← {cpu, ram, gpu}
    POST(orchestrator, heartbeat, vitals)
        W.last_seen ← current_time()
```

The worker-liveness parameters used by the framework are summarized in Table I.

TABLE I. SYSTEM PARAMETERS FOR WORKER LIVENESS DETECTION

Parameter	*Symbol*	*Value*	*Justification*
Heartbeat interval	T_h	4 seconds	Balance between telemetry overhead and detection responsiveness
Monitor interval	T_m	5 seconds	Slightly longer than heartbeat to allow for network jitter
Offline threshold	$T_{threshold}$	60 seconds	Tolerates 15 missed heartbeats before marking offline

To maintain responsiveness, the Orchestrator decouples heartbeat reception from failure detection. A background monitor evaluates the worker registry every five seconds, marking a worker OFFLINE if its last_seen timestamp exceeds a 60-second threshold (tolerating up to 15 missed heartbeats at a 4-second interval) to handle transient network delays.

This worker-level liveness monitoring operates as a part of a multi-tiered fault-tolerance strategy, complementing gRPC deadlines for network-level failures and subprocess timeouts for execution failures to ensure global computation never stalls.

Upon marking a worker offline, the Orchestrator unassigns its in-flight job shards, returns them to QUEUED state, and immediately re-dispatches them to available idle workers. The failed worker is removed from the active cluster topology to protect distributed peer-to-peer machine learning operations.

5) Security and Isolation Boundaries

The current execution model provides fault isolation through process-level separation, mandatory artifact cleanup, and hierarchical timeout enforcement. However, this approach trades security guarantees for execution flexibility and direct GPU access, resulting in two primary limitations:

- Unencrypted Transport: gRPC uses unencrypted binary serialization, leaving payloads vulnerable to interception on the local network.
- Shared Host Resources: Child processes share the host's filesystem and network interface, allowing scripts to potentially access resources outside their designated task directory.

D. Data Parallelism and Adaptive Sharding Model

The proposed framework employs a memory-bounded adaptive sharding strategy rather than naive equal partitioning. The total workload is divided into shards such that no single shard exceeds a configurable maximum size, preventing memory exhaustion on any worker node regardless of the total simulation scale.

The shard count is determined dynamically through the following procedure:

Algorithm 5: Adaptive Shard Count Determination

Input: *N_total, N_workers, MAX_SIMS_PER_SHARD*
Output: *shard_sizes*
Initialization:
num_shards ← N_workers
shard_sizes ← ∅

```
num_shards ← N_workers
While N_total / num_shards > MAX_SIMS_PER_SHARD:
    num_shards ← num_shards + N_workers
shard_sizes ← distribute(N_total, num_shards)
Return shard_sizes
```

Within each shard, the worker further subdivides execution into fixed-size memory batches, accumulating only scalar sufficient statistics and releasing intermediate arrays immediately after each batch:

1) Uniform Domain Partitioning

For a computational task requiring *M* independent iteration steps distributed across *N* verified execution nodes, the sub-range allocation m_i for worker *i* is defined as:

$$m_i = \left\lfloor \frac{M}{N} \right\rfloor \forall\, i \in \{0,1,\ldots,\ N-1\} \quad (1)$$

To maintain statistical independence in stochastic tasks across workers, each node receives a unique, deterministic pseudo-random seed parameter S_i:

$$S_i = S_{base} + i \quad (2)$$

2) Theoretical Scalability Model

The theoretical speedup limit of the distributed system is modeled using Amdahl's Law, extended to incorporate network communication latencies:

$$Speedup(N) = \frac{1}{(1-P) + \frac{P}{N} + \delta_{comm}(N)} \quad (3)$$

where P represents the parallelizable fraction of the computational workload, and $\delta_{comm}(N)$ accounts for serialization, network transmission, and scheduling latency as the node count scales.

E. Monte Carlo Simulation as Validation Workload

To evaluate dispatch performance, scheduling latencies, and payload transmission overhead without machine learning dependencies, a distributed GBM Monte Carlo simulation is used as a benchmark workload, which models asset price trajectories in computational finance. [14], [15] The embarrassingly parallel nature of Monte Carlo simulation, where individual trials are statistically independent, makes it an ideal workload for isolating distributed framework overhead from algorithmic communication costs.

Because the continuous-time Stochastic Differential Equation (SDE) admits a known closed form analytical solution under Itô's Lemma, it provides an exact theoretical baseline ($E[S(T)]$) against which the distributed numerical output can be rigorously verified. [15]

The stochastic behavior of an asset price $S(t)$ over time t is governed by the continuous SDE:-

$$dS(t) = \mu S(t)dt + \sigma S(t)\, dW(t) \quad (4)$$

where μ represents the constant expected risk-neutral drift rate, σ represents the volatility parameter, and $W(t)$ denotes a standard Wiener process (Brownian Motion).

Under Itô's Lemma, the analytical solution to the SDE yields the discrete log-normal asset price path update equation over time step Δt:

$$S(t+\Delta t) = S(t) \cdot exp\left(\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}Z\right) \quad (5)$$

where $Z \backsim N(0, 1)$ is a standard normal random variable.

To evaluate M independent Monte Carlo trajectories over horizon T with K discrete time steps ($\Delta t = T/K$), the Orchestrator partitions the workload uniformly across N verified worker nodes (Equation 1).

Each worker node i receives a unique pseudo-random seed S_i to generate independent Gaussian trajectories without random stream overlap. Upon completion, each worker computes the local expected terminal asset value $\overline{S_{i,T}}$:

$$\overline{S_{i,T}} = \frac{1}{m_i}\sum_{j=1}^{m_i} S_{j,T}^{(i)} \quad (6)$$

The Central Orchestrator aggregates returned terminal trajectory statistics to derive the global expected asset price estimate $\hat{E}[S(T)]$:

$$\hat{E}[S(T)] = \frac{1}{N}\sum_{i=0}^{N-1}\overline{S_{i,T}} = \frac{1}{M_{total}}\sum_{i=0}^{N-1}\sum_{j=1}^{m_i} S_{j,T}^{(i)} \quad (7)$$

Algorithm 6: Distributed Monte Carlo Brownian Motion Simulation

Input: Total Trajectories M, Time Steps K, Time Horizon T, Drift μ, Volatility σ, Initial Price S_0, Workers N, Seed S_base
Output: Global Expected Price $\hat{E}[S(T)]$
Initialization:
m_shard ← *floor(M / N)*
$\Delta t \leftarrow T / K$
S_i ← *S_base* + *i* for each worker *i*
Orchestrator Execution:
m_shard = *floor*(*M* / *N*)
t = *T* / *K*
for *i* = 0 *to N*-1 **do**
S_i = *S_base* + *i*
DispatchTaskAsync(*W*[*i*], *m_shard*, *K*, *t*, *S_0*, *S_i*)
end for
Await response payloads from all worker nodes W
[*S*(*T*)] = *sum*(*S_mean_i* * *m_shard* **for** all worker *i*) / (*m_shard* * *N*)
return [*S*(*T*)]
Worker Execution (Executed independently at Node *i*):
Initialize Pseudo-Random Normal Generator with Seed S_i
Sum_terminal = 0.0
for *path* = 1 *to m_shard* **do**
S_current = *S_0*
for *step* = 1 *to K* **do**
Z = *RandomStandardNormal*()
S_current = *S_current* * *exp*((*μ* – (0.5 * *σ*^{2})) * *Δt* + *σ* * *sqrt*(*Δt*) * *Z*)
end for
Sum_terminal = *Sum_terminal* + *S_current*
end for
S_mean_i = *Sum_terminal* / *m_shard*
Transmit *ShardResponse*(*S_mean_i*) to Orchestrator

F. Distributed Machine Learning and Gradient Aggregation

For deep learning workloads, the system utilizes data parallelism through synchronous input auto-sharding and decentralized Ring-AllReduce parameter aggregation.

To eliminate the centralized bandwidth bottlenecks inherent in traditional parameter-server architectures, the framework implements a decentralized Ring-AllReduce topology. [16], [17] During the backward pass, gradient updates are passed sequentially between peer nodes, allowing scalable parameter synchronization across heterogeneous hardware. [13], [16]

1) Deterministic Cluster Topology Generation

Workers retrieve addresses from the registry and sort them lexicographically by hardware ID to build a peer-to-peer cluster. Index_0 becomes the Chief node (handles logging/checkpointing); indices $k \geq 1$ act as Secondary workers.

2) Distributed Data Auto-Sharding

Input datasets are partitioned dynamically across participating workers. For a dataset $\mathcal{D}$ formatted as an ordered sequence of batches B_j , the batch selection rule for worker index $k \in \{0, 1, \ldots, N-1\}$ is defined as:

$$\mathcal{D}_k = \{B_j \in \mathcal{D} \mid j \bmod N = k\} \quad (8)$$

This guarantees mutually exclusive data processing per node without needing a distributed file system.

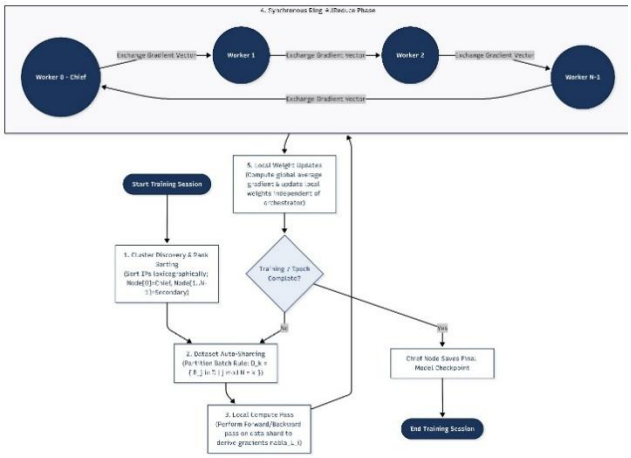


Fig. 6. Distributed machine learning training workflow with Ring-AllReduce

3) Synchronous Ring-AllReduce Gradient Aggregation

During the backward pass of training step *t*, worker *i* calculates its local gradients $\nabla L_i(\theta_t)$ on data shards D_i. Nodes then aggregate these updates directly across the peer-to-peer network channel using a synchronous Ring-AllReduce operation to derive the global mean gradient:

$$\overline{\nabla} L(\theta_t) = \frac{1}{N} \sum_{i=0}^{N-1} \nabla L_i(\theta_t) \nabla \quad (9)$$

Each worker then applies the global average gradient vector to its local parameter set θ using Stochastic Gradient Descent (SGD) or adaptive optimizers:

$$\theta_{t+1} = \theta_t - \eta \overline{\nabla} L(\theta_t) \quad (10)$$

This maintains strict parameter synchronization across all execution nodes throughout training without sending parameter tensors through the Central Orchestrator, thereby reducing communication overhead and avoiding a centralized bottleneck.

G. Monitoring, Observability, and Cross-Platform Deployment

The framework uses a push-based observability layer to provide continuous cluster visibility. Rather than relying on client polling, the Orchestrator uses a persistent bidirectional channel to stream real-time state updates directly to dashboard clients at the worker heartbeat frequency.

The Orchestrator ingests periodic worker vitals, aggregating them into a unified cluster state snapshot that is proactively pushed to dashboard clients via persistent, bidirectional gRPC streams at heartbeat cadence. The frontend dynamically updates individual worker state tiles in real-time, including handling state transitions (e.g., OFFLINE degradation, shard reallocation, and fault recovery events) through event-driven reactivity, eliminating client-side polling overhead and manual refreshes.

To ensure non-interactive runtime persistence and autonomous recovery across heterogeneous OS environments, the worker daemon leverages native process supervision frameworks. A platform-aware bootstrapping utility registers the worker with host initialization subsystems. This configures automated process lifecycle hooks, crash-restart policies, and environment setup as detailed in Table II.

TABLE II. WORKER INITIALIZATION AND CRASH RECOVERY CONFIGURATION

OS	***Initialization Subsystem***	***Privilege Required***	***Crash Recovery***
Windows	Task Scheduler (schtasks)	Administrator	Configurable restart policy
Linux	systemd service manager	sudo	Restart=always
macOS	launchd agent	User	KeepAlive=true

Upon system startup, the host process manager executes the worker daemon prior to user authentication, automatically initializing network discovery, completing orchestrator registration, and starting heartbeat transmission without manual intervention. Conversely, a platform-aware teardown script unregisters the daemon service and purges all local runtime artifacts when decommissioning a node.

IV. Experiments And Results

A. Experimental Configuration

The distributed Monte Carlo simulation was selected as the primary benchmark workload. The simulation parameters were configured as follows:

TABLE III. MONTE CARLO SIMULATION PARAMETERS

Parameter	***Symbol***	***Value***
Total simulations	N_{total}	200,000,000
Initial asset price	S_0	100.0
Annualized drift	μ	0.05
Annualized volatility	σ	0.20
Time horizon	T	1.0 year
Strike threshold	K	100.0
Max simulations per	—	5,000,000
In-shard batch size	—	2,000,000
Max shard retries	—	3

The heterogeneous worker cluster comprised three nodes:

TABLE IV. HETEROGENEOUS WORKER CLUSTER CONFIGURATION

Node	***Processor***	***GPU***
Node 1	Apple M-series (MacBook Air)	Integrated
Node 2	Intel Core i7-14th Gen HX	NVIDIA RTX 4060
Node 3	Intel Core i7-12th Gen P	NVIDIA RTX 2050

Each configuration was executed five times to account for system load variability.

B. Timing Results

TABLE V. SINGLE-NODE AND THREE-NODE DISTRIBUTED EXECUTION TIMES

Single-node		**Three-node**	
Run	***Time (s)***	***Run***	***Time (s)***
1	995.26	1	238.92
2	1096.53	2	278.42
3	628.15	3	186.88
4	1259.41	4	165.52
5	973.94	5	283.94
Mean	990.66	***Mean***	230.74
Std Dev	228.91	***Std Dev***	51.25

C. Simulation Output

The output produced by a representative distributed run is tabulated in Table VI.

TABLE VI. AGGREGATED STATISTICAL RESULTS FROM THE DISTRIBUTED MONTE CARLO SIMULATION

Shards combined:	38 of 42
Total simulations:	190,000,000
Estimated $E[S_T]$:	105.1254
Std error of estimate:	0.0015
95% CI:	[105.1224, 105.1284]
$P(S_T > 100.0)$:	55.9585%

The experimental configuration of *N_total* = 200,000,000 and *N_workers* = 3 with *MAX_SIMS_PER_SHARD* = 5,000,000 produces 42 shards of approximately 4,761,904 simulations each, dispatched in 14 sequential rounds of 3 shards.

Two shards comprising 10,000,000 simulations were permanently dropped after exhausting all retry attempts. Despite this, the standard error of 0.0015 on a mean of 105.1254 corresponds to a relative standard error of 0.0014%, confirming statistical robustness.

The theoretical expected terminal price under the configured parameters is:

$$E[S_T] = S_0 \times exp(\mu T) \quad (11)$$

$$E[S_T] = 100 \times exp(0.05) = 105.1271$$

The estimated value of 105.1254 differs from theory by 0.0017, well within the 95% confidence interval, confirming correctness of the distributed computation.

D. Scalability Analysis

The observed speedup of distributed over single-node execution is:

$$S_{observed} = 990.66 / 230.74 = 4.29x$$

This exhibits superlinear speedup against the linear expectation of 3.0x for three workers, arising from the heterogeneous cluster's greater aggregate CPU core count relative to the single-node baseline.

Applying Amdahl's Law with the measured serial fraction, we get:

$$S(N) = 1 / (s + (1 - s) /N) \quad (12)$$

$$s = \frac{T_{Serial}}{T_{Series}} \quad (13)$$

$$s = 6 / 990.66 \approx 0.006 \ (0.6\%)$$

TABLE VII. THEORETICAL AND OBSERVED SPEEDUP AND PARALLEL EFFICIENCY

Workers (N)	***Theoretical S(N)***	***Theoretical E(N)***	***Observed S(N)***	***Observed E(N)***
1	1.00x	100.0%	1.00x	100.0%
2	1.99x	99.5%	4.29x*	143.0%*
3	2.98x	99.3%		
4	3.98x	99.4%		
16	15.85x	99.1%		
∞	166.67x	—	—	—

The theoretical maximum speedup is:

$$S(\infty) = 1/s \quad (14)$$

$$S(\infty) = 1/0.006 \approx 166.7x$$

The near-perfect theoretical efficiency of 99.3% confirms that framework coordination overhead (shard dispatch, network transfer, result polling, and aggregation) constitutes less than 1% of total execution time at this workload scale.

E. Analysis and Discussion

While empirical validation was physically limited to a three-node cluster, this configuration was deliberately selected to stress-test extreme cross-platform heterogeneity rather than homogeneous horizontal scale. The cluster simultaneously integrated macOS (Advanced Reduced Instruction Set Computer Machine (ARM) architecture), Native Windows (x86), and Linux via WSL2 (virtualized network stack)—the most volatile combination of consumer environments.

Because the Monte Carlo workload is embarrassingly parallel, the observed serial fraction of $s = 0.006$ demonstrates that algorithmic synchronization overhead is mathematically negligible. Based on the extended Amdahl formulation (Equation 3), scaling bottlenecks up to $N = 10$ or $N = 50$ nodes will not be governed by algorithmic limitations, but rather bounded purely by physical wireless access point (AP) throughput during the 25-byte heartbeat intervals. Therefore, the 3-node heterogeneous test successfully validates architectural correctness, while the mathematical model projects robust scalability.

The experimental results demonstrate three principal findings. First, the zero-configuration UDP broadcast discovery mechanism successfully enables heterogeneous nodes running Windows, macOS, Linux, and WSL2 to form a functional distributed cluster without any manual IP configuration, validating the framework's deployment accessibility claim.

Second, the observed speedup of 4.29x with three workers, exceeding the linear bound of 3.0x, demonstrates that distributing computation across heterogeneous hardware provides performance benefits beyond simple parallelism. The aggregate multi-core capacity of three diverse machines significantly exceeds the parallelism available on a single baseline machine, producing superlinear efficiency. This finding suggests that heterogeneity, typically viewed as a

challenge in distributed systems, can be leveraged as a performance advantage when workloads are embarrassingly parallel.

Third, the statistical validity of the distributed result, with an estimated $E[S_T]$ of 105.1254 against a theoretical expectation of 105.1271 and a 95% confidence interval width of 0.006, confirms that the adaptive sharding and aggregation mechanism introduces no numerical error. Even with 5% of simulations excluded due to shard failures, the remaining 190,000,000 simulations produce an estimate of sufficient precision for practical applications.

The high standard deviation in single-node execution (228.91s) compared to distributed execution (51.25s) suggests that the distributed configuration is more resilient to background system load variability, as the impact of a single machine's load spike is diluted across the cluster.

The permanently dropped shards in the experimental run represent a known limitation. Shards assigned to workers that experience transient failures during the retry window are lost. For applications requiring exact simulation counts, a more aggressive retry strategy or a persistent shard queue with crash recovery would be necessary.

Furthermore, the current architecture relies on a centralized Orchestrator, representing a Single Point of Failure (SPOF). While the system successfully manages Worker-node volatility, the failure of the central manager results in a complete cluster halt and the loss of volatile state data.

V. Conclusion And Future Works

This paper presented a zero-configuration heterogeneous distributed parallel processing framework based on a three-tier master-worker architecture with UDP broadcast-based worker discovery, binary RPC task dispatch via gRPC and Protobuf, adaptive memory-bounded sharding, asynchronous fault detection, and cross-platform auto-deployment. The framework was validated through a 200,000,000-trial GBM Monte Carlo simulation achieving an observed speedup of 4.29x over single-node execution with superlinear parallel efficiency attributable to the heterogeneous cluster's aggregate multi-core capacity.

The low measured serial fraction of 0.6% projects a theoretical maximum speedup of approximately 166.7x, indicating substantial headroom for scaling to larger worker clusters. The framework's task-agnostic design was validated across two fundamentally different workload types Monte Carlo financial simulation and distributed machine learning training demonstrating its applicability as a general-purpose distributed computation platform.

Future work will focus on several directions. Replacing the current sequential round-based shard dispatch with fully asynchronous dispatch and worker-side queuing would eliminate the round latency overhead present in the current 14-round dispatch structure. Implementing Mutual Transport Layer Security (mTLS) for encrypted gRPC transport and OS-level resource quotas via cgroups or Docker containers would extend the framework from a research prototype to a production-ready secure distributed grid. Extending the worker discovery protocol to support Wide Area Networks through a lightweight rendezvous server would enable cross-network cluster formation. Finally, implementing dynamic load-aware shard sizing where shard sizes are adjusted based on registered worker capability profiles would further improve load balancing across heterogeneous hardware configurations. Future iterations could explore a High-Availability Orchestrator model utilizing a distributed consensus protocol like Raft or Paxos to allow a secondary Orchestrator to assume control in the event of primary failure.

References

[1] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, pp. 114-117, Apr. 1965.

[2] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proc. AFIPS Spring Joint Comput*, Conf, 1967, pp. 483–485.

[3] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel Iterative Algorithms: From Sequential to Grid Computing*. Boca Raton, FL, USA: CRC Press, 2007.

[4] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, “StarPU: A unified runtime system for heterogeneous multicore architectures,” *Concurr. Comput Pr. Exper*, vol. 23, no. 2, pp. 187-198, Feb. 2011.

[5] D. P. Anderson, “BOINC: A system for public-resource computing and storage,” in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput*, 2004, pp. 4–10.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput*, HotCloud, 2010, p. 10.

[7] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proc. 6th USENIX Symp. Operating Syst. Design Implement*, OSDI, 2004, pp. 137–150.

[8] P. Patel and A. Trivedi, “Evaluating transport layer performance over modern wireless local area networks,” *IEEE Trans Netw Serv Manage*, vol. 17, no. 3, pp. 1812-1824, Sep. 2020.

[9] R. Wang, S. Balasubramanian, S. Sivasubramanian, and N. Venkatasubramanian, “An in-depth comparative study of cloud networking: gRPC vs. REST,” in *Proc. IEEE 12th Int. Conf. Cloud Comput*, CLOUD, 2019, pp. 446–453.

[10] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J ACM*, vol. 43, no. 2, pp. 225-267, Mar. 1996.

[11] P. Moritz *et al.*, “Ray: A distributed framework for emerging AI applications,” in *Proc. 13th USENIX Symp. Operating Syst. Design Implement*, OSDI, 2018, pp. 561–577.

[12] A. Borzunov *et al.*, “Petals: Collaborative inference and fine-tuning of large models over the internet,” in *Proc. 61st Annu. Meeting Assoc. Comput*, Linguistics (ACL: System Demonstrations, 2023, pp. 558–568.

[13] A. Sergeev and M. D. Balso, “Horovod: Fast and easy distributed deep learning in TensorFlow.” 2018.

[14] C. Lemieux, *Monte Carlo and Quasi-Monte Carlo Sampling*. New York, NY, USA: Springer Science+Business Media, 2009.

[15] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. New York, NY, USA: Springer-Verlag, 2003.

[16] A. Gibiansky, “Bringing HPC techniques to deep learning,” *Baidu Res. Tech Rep*, 2017, [Online]. Available: https://github.com/baidu/allreduce/blob/master/docs/bfloat16.pdf

[17] M. Li, “Scaling distributed machine learning with the parameter server,” in *Proc. 11th USENIX Symp. Operating Syst. Design Implement*, OSDI, 2014, pp. 583–598.