

Seminar Report

Parallel Monte Carlo Methods for Numerical Estimation Using MPI

*Reproducible distributed integration using static MPI, dynamic load balancing,
SeedSequence/Philox, and Numba*

Carl Etornam Yevutsey

Evans Boateng

Ernest Osei Amofo

Supervisor: Patrick Höhn

Georg-August-Universität Göttingen

Practical Course on High-Performance Computing

September 30, 2026

Contents

Abstract

1 Introduction

- 1.1 Problem context
- 1.2 Research question and objectives
- 1.3 Contributions and scope
- 1.4 Reading guide

2 Mathematical model and estimator

- 2.1 Integral and exact value
- 2.2 Monte Carlo estimator
- 2.3 Analytical variance and convergence
- 2.4 Sampling error, implementation error, and timing

3 Computing architecture

- 3.1 Shared computation
- 3.2 True serial baseline
- 3.3 Static MPI partitioning
- 3.4 Dynamic master-worker scheduling
- 3.5 Resource accounting

4 Parallel random-number design

- 4.1 Why stream identity belongs in the method
- 4.2 Four policies in the code
- 4.3 Logical stream identities
- 4.4 What the RNG experiment tests

5 Implementation and data flow

- 5.1 Program organization
- 5.2 Local task execution
- 5.3 Static control flow
- 5.4 Dynamic control flow and termination
- 5.5 Output provenance and auditability

6 Correctness and verification strategy

- 6.1 Mathematical invariants
- 6.2 Random-stream and scheduling invariants
- 6.3 Statistical consistency
- 6.4 Limits of the checks

7 Experimental methodology

- 7.1 Fair comparison and timing boundary
- 7.2 Calibration and crossover
- 7.3 Strong scaling
- 7.4 Weak scaling
- 7.5 Chunk size and worker activity
- 7.6 Controlled scheduling stress

8 Statistical results

- 8.1 Data provenance and protocol
- 8.2 Summary of repeated estimates
- 8.3 Confidence intervals and interpretation
- 8.4 Numerical scale and practical meaning

9	Parallel random-stream results
9.1	Fixed-seed checks
9.2	Visual example and limits
10	Two-node cluster and evidence provenance
10.1	Measurement setting
10.2	Campaign design and separation of evidence
10.3	Timing and derived measures in the final data
11	Stage 03 workload calibration and crossover
11.1	Calibrated serial workload
11.2	Exploratory serial and static MPI comparison
11.3	What the crossover means
12	Stage 04 production scaling and scheduling
12.1	Strong scaling at fixed workload
12.2	Speedup and efficiency
12.3	Weak scaling
12.4	Static versus dynamic at equal computing-worker count
12.5	Chunk-size sensitivity and worker activity
13	Stage 05 real-MPI scheduling stress
13.1	Controlled task-cost profiles
13.2	Makespan and utilization results
13.3	Stopwatch boundary and causal caution
14	Discussion
14.1	What was demonstrated
14.2	Why the architecture comparison is conditional
14.3	Linking performance to numerical precision
14.4	Presentation of uncertainty and negative results
15	Limitations and future work
15.1	Hardware and scaling scope
15.2	Interpretation limits of the experiments
15.3	Next research steps
16	Conclusion
	References
	Project sources
	Appendix A Work sharing
	Appendix B Reproducibility protocol
	B.1 Software and machine record
	B.2 Minimal numerical smoke test
	B.3 Statistical campaign
	B.4 Production performance sequence
	Appendix C Data definitions and analysis checks
	C.1 Core variables
	C.2 Stage 06 columns
	C.3 Production performance checks
	Appendix D Evidence map for final submission

Figures

Figure 1. Empirical and analytical estimator standard deviation

Figure 2. Root mean squared error and empirical bias

Figure 3. Illustrative random-stream prefixes

Figure 4. Exploratory workload sweep

Figure 5. Production strong-scaling runtime

Figure 6. Paired speedup against true serial

Figure 7. Weak-scaling runtime

Figure 8. Dynamic chunk-size sensitivity

Figure 9. Real-MPI scheduler stress

Tables

Table 1. Repeated-estimate summary

Table 2. Confidence intervals for repeated-run means

Table 3. Finite random-stream validation

Table 4. Two-node worker placement

Table 5. Exploratory calibration comparison

Table 6. Production strong scaling

Table 7. Production weak scaling

Table 8. Dynamic chunk-size experiment

Table 9. Real-MPI synthetic scheduler stress

Abstract

This report studies a distributed Monte Carlo estimator for a known high-dimensional integral. The project evaluates the sum of squared coordinates over the hypercube $[-1,1]^d$, with $d = 10$ as the principal experiment. Its exact integral and estimator variance are available analytically, so numerical correctness can be assessed separately from execution time. The software provides a serial baseline, static MPI partitioning, and a dynamic master-worker scheduler. NumPy generators supply samples, Numba compiles the local reduction, and the principal parallel random-number construction combines `SeedSequence` with Philox.

The central research question is when MPI becomes useful for this problem, and how workload size, scheduling, and random-stream construction affect performance and statistical validity. A two-node heterogeneous Windows cluster campaign supplies seven-repeat calibration, production scaling, and real-MPI synthetic scheduler-stress evidence. At 60 million samples, the Stage 04 true-serial median was 23.748 seconds. Four-worker static MPI took 13.180 seconds; four-worker dynamic MPI took 9.470 seconds while using a fifth, coordinating rank. Repetition-paired speedups were 1.802 and 2.483. These are measured results on two unlike, non-isolated hosts over Wi-Fi, not a universal scheduler ranking.

The separately frozen Stage 06 campaign contains 30 estimates from spawned streams at each of 10,000, 100,000, and 1,000,000 samples. Their empirical standard deviations are 9.7989, 2.9903, and 0.7869, broadly consistent with analytical values 9.6544, 3.0530, and 0.9654. Finite stream-prefix checks reproduce intended streams and detect the deliberately duplicated `same_seed` baseline. The report distinguishes this statistical evidence from the two-node performance campaign, documents the extra dynamic master rank and stopwatch boundaries, and limits its conclusions to the measured system and workload.

Keywords: Monte Carlo integration; MPI; static partitioning; master-worker scheduling; Philox; `SeedSequence`; Numba; reproducibility; strong scaling; statistical validation.

1 Introduction

1.1 Problem context

Numerical integration becomes difficult as the number of dimensions grows. A tensor-product quadrature rule with m points in each of d directions needs m^d evaluations. The count grows rapidly even when each individual evaluation is cheap. Monte Carlo integration replaces a structured grid by sampled points. Its usual standard-deviation rate, proportional to $N^{-1/2}$ for N independent samples with finite variance, does not carry an exponential dependence on d . The constant in that rate still depends on the integrand and the dimension. Monte Carlo is therefore attractive for high-dimensional problems when deterministic grids are impractical and an uncertainty estimate is acceptable [1].

The estimator in this project is especially convenient for evaluating a computing method. Every sample can be generated and evaluated independently, so the work can be distributed across processes. The exact answer is also known. A wrong numerical result is visible without relying on

a second approximate algorithm. These advantages make the problem a useful controlled benchmark for MPI, but they also restrict generality: the smooth quadratic integrand is nearly uniform in cost per sample and cannot represent every scientific Monte Carlo workload.

Parallel execution introduces questions beyond the mathematical estimator. More processes do not necessarily reduce elapsed time, because process startup, stream construction, communication, synchronization, and task scheduling have costs. In a dynamic scheme, one process may coordinate rather than calculate. Random streams also matter: repeated pseudo-random samples can make nominal sample counts misleading. Finally, timing a JIT-compiled Python program requires a clear boundary around compilation and application work. The project addresses these concerns with separate serial, static, and dynamic execution paths and with an evidence protocol that records what has actually been measured.

1.2 Research question and objectives

The project asks: **When does MPI become worthwhile for high-dimensional Monte Carlo integration, and how do workload size, scheduling strategy, and random-stream construction affect performance and statistical validity?** This question is divided into linked objectives. First, the group builds a mathematically checkable estimator and a genuinely one-process, one-thread baseline. Second, it implements two MPI distributions that compute the same total sample count. Third, it defines fair strong-scaling, weak-scaling, and scheduler comparisons. Fourth, it checks repeated-estimate variability and finite random-stream behavior. Fifth, it makes performance claims only when the corresponding production data and environment records exist.

The work is a study of an implemented system, not a proof that one scheduling scheme always wins. For a uniform-cost integrand, static distribution has a simpler communication pattern; dynamic scheduling can respond to uneven task costs or workers. The two-node production campaign measured dynamic mode faster than static at four computing workers for the main integrand, while the separate synthetic stress study favored different schemes for different task-cost profiles. Those observations must be interpreted with the extra dynamic master rank, unlike CPUs, different stress-test timing boundaries, and limited worker counts.

1.3 Contributions and scope

The repository contains a serial entry point, a static MPI program, a dynamic MPI program with a master and workers, a shared local numerical kernel, four random-stream policies, structured result files, test cases, experiment drivers, visualization scripts, and a teaching dashboard [P1]. The report contributes an integrated account of those components and their evidence. It derives the exact integral and variance, explains the distribution and measurement rules, interprets the statistical and performance data, and states the limits of those observations.

1.4 Reading guide

Sections 2 and 3 state the mathematical and computing models. Sections 4 through 6 describe implementation, random streams, and verification. Section 7 defines the experiment design and metrics. Sections 8 and 9 give the separately frozen statistical and random-stream results. Sections 10 through 13 present the two-node environment and measured calibration, scaling, and scheduler-stress findings. Sections 14 through 16 discuss interpretation, limitations, and conclusions. The appendices provide reproducibility commands, data fields, and documented responsibility areas. Project-file citations use [P1] to [P9]; numbered citations refer to external literature and official technical documentation.

2 Mathematical model and estimator

2.1 Integral and exact value

Let $\Omega_d = [-1, 1]^d$, and let $f(x) = \sum_{j=1}^d x_j^2$. The quantity of interest is

$$I_d = \int_{[-1, 1]^d} \sum_{j=1}^d x_j^2 dx.$$

Each coordinate term has a one-dimensional integral of $2/3$; the remaining $d - 1$ coordinates contribute a volume factor of 2^{d-1} . By linearity,

$$I_d = d \left(\frac{2}{3} \right) 2^{d-1} = \frac{d 2^d}{3}.$$

At the principal dimension $d = 10$, the exact value is $10240/3$, or approximately 3413.333333 . This identity is used by the software to calculate signed, absolute, and relative error. It also gives a direct diagnostic for output from every execution mode. A value close to the target in one random trial is reassuring, but it is not a full validation of the random-number policy or of the variance.

2.2 Monte Carlo estimator

Let $x_1, \dots, x_N$ be independent points uniformly distributed on Ω_d . The domain has volume 2^d , so

$$\hat{I}_N = \frac{2^d}{N} \sum_{i=1}^N f(x_i).$$

For one uniform coordinate U on $[-1, 1]$, $E[U^2] = 1/3$. Hence $E[f(x_i)] = d/3$ and, under the sampling assumptions, $E[\hat{I}_N] = d 2^d / 3 = I_d$. The estimator is unbiased in this mathematical model. The program's empirical mean over a finite set of trials may fall above or below the exact value; that observed difference should be called empirical bias or mean error rather than proof of a systematic implementation bias.

The exact sample count matters. If N is not divisible by the number of ranks or by the dynamic chunk size, all remainder samples must still be processed. Static partitioning assigns the extra points to the first ranks. Dynamic scheduling creates one final short task. The program records `samples_processed`, allowing tests and later audits to check that this count equals the request `[P1]`.

2.3 Analytical variance and convergence

The fourth moment of a uniform coordinate is $E[U^4] = 1/5$. Thus

$$\text{Var}(U^2) = \frac{1}{5} - \left(\frac{1}{3}\right)^2 = \frac{4}{45}.$$

The coordinate terms are independent, giving $\text{Var}(f(X_i)) = 4d/45$. For independent sample points,

$$\text{Var}(\hat{I}_N) = \frac{4^d}{N} \frac{4d}{45}, \quad \text{SD}(\hat{I}_N) = 2^d \sqrt{\frac{4d}{45N}}.$$

At $d = 10$, the theoretical standard deviations are 9.6544, 3.0530, and 0.9654 at $N = 10^4$, 10^5 , and 10^6 . Multiplying N by 100 should divide the theoretical standard deviation by 10. Parallelism does not change this statistical law: it can reduce the wall time required to process more valid samples. If two workers use the same pseudo-random stream, their duplicated evaluations do not create the same independent information as distinct samples, even though the program may count both evaluations.

The relative theoretical standard deviation is $2/\sqrt{5dN}$. This expression helps compare uncertainty across dimensions and sample counts, but it is conditional on the ideal independent uniform sampling model. The project tests finite properties of its pseudo-random streams; it does not prove the assumptions for every future stream or workload.

2.4 Sampling error, implementation error, and timing

Three kinds of variation must be kept separate. Sampling error is the difference between an estimate and the exact integral caused by finite random sampling. Implementation error arises if the code uses the wrong domain volume, loses points, duplicates streams unexpectedly, or aggregates sums incorrectly. Timing variation arises from operating-system scheduling, cache state, cluster contention, binding, and other execution conditions. A single small absolute integration error does not imply fast code or high-quality random numbers. A fast run does not imply a correct estimator. The report therefore treats numerical checks, repeated statistical tests, and repeated performance trials as distinct evidence streams.

3 Computing architecture

3.1 Shared computation

All modes evaluate the same mathematical expression. A generator creates an array of sampled points, with rows representing samples and columns representing dimensions. The local kernel sums the squared coordinates for each row and returns one scalar sum. The serial process or MPI ranks then add their scalar sums and multiply by $2^d/N$. The core operation takes work proportional to $N \cdot d$; random-number generation and memory traffic also grow with that product. The implementation uses `float64` point arrays and floating-point accumulation [P1].

The local kernel is compiled with Numba. `worker_compute.py` has a `prange` kernel for MPI workers and a separate serial kernel for the true serial baseline. The `parallel=True` version is configured to use one thread per rank in primary pure-MPI measurements. That choice prevents a purported four-rank run from silently becoming a many-thread run. The serial path explicitly selects the serial kernel, so its baseline is one process and one active compute thread [P1, P2]. Numba documents both the parallel transformation and `prange` behavior [6].

Sample arrays are constructed per logical task or rank assignment. Consequently, memory consumption depends on the configured task or chunk size and dimension. It is incorrect to assume the current worker function enforces a fixed 32 MiB cap: `process_task_timed` allocates an array of shape `(chunk_size, dimensions)` before the kernel call. A production calibration must choose a chunk size that fits the allocation. A future bounded sub-batching implementation could reduce this risk, provided stream continuity and timing semantics remain documented.

3.2 True serial baseline

`serial_baseline.py`, exposed through `sequential.py`, runs one operating-system process. It sets the active Numba thread count to one, warms the kernels before timing, creates logical chunk states, and measures the loop that generates samples and evaluates the serial compiled kernel. Setup and warm-up are recorded but excluded from the main compute-phase timer. This separation supports a comparison of steady-state computation. It does not represent user-perceived latency including interpreter launch or JIT compilation [P1].

The serial `--chunk-size` controls the number of samples generated in one array. Its default is one million. Repeated local chunks have distinct stream identities under the default policy. When comparing with an MPI mode, the experiment must hold the estimator, dimension, requested total samples, RNG policy, thread count, and timing boundary as closely as possible. Random draws need not be bitwise identical across different partitionings for a fair performance comparison, but their statistical construction must be valid and recorded.

3.3 Static MPI partitioning

In static mode, all `P` MPI ranks compute. The assignment for rank `r` is `floor(N/P) + 1` if `r < N mod P`, and `floor(N/P)` otherwise. Rank zero prepares one stream state per rank and scatters the states. After a barrier, each rank generates its local samples and computes a scalar. An MPI sum

reduction combines these scalars at rank zero. The reported compute-phase time is the maximum elapsed time among ranks, because completion waits for the slowest rank [P1].

Static partitioning has one principal communication step after local work, plus setup and diagnostics outside the main timer. It is therefore a natural candidate for a homogeneous per-sample workload. Its weakness is limited adaptability: if a rank receives unusually slow work or a slow machine, other ranks may wait at the reduction. For the present integrand, the arithmetic cost of samples is nearly constant, so significant task imbalance is more likely to arise from system effects than from the function itself. That expectation requires measurements before it is turned into a result.

3.4 Dynamic master-worker scheduling

Dynamic mode requires at least two MPI ranks. Rank zero is a master. The other $W = P - 1$ ranks are workers. The master divides N into $\text{ceil}(N/C)$ logical tasks for chunk size C . It initially sends tasks to available workers and sends a completion signal to workers for which no task exists. When a result arrives from any active worker, the master adds the partial sum and sends that worker another task if one remains. A worker reports its task count, processed samples, random-generation time, kernel time, send time, and receive-wait time [P1].

The measured dynamic interval begins immediately before initial dispatch and ends when the last task result has been received and active workers have been sent stop messages. Later collection of worker diagnostic records is outside that interval. The program records a coarse worker utilization metric: task elapsed time divided by the master's measured interval, capped at one. That metric includes a task's local generator and kernel work; it does not isolate every MPI wait state. The reported `communication_time_seconds` field is a residual using the longest worker task time, not a profiler-derived MPI communication measurement. Any report of actual message time requires an event trace or more precise instrumentation.

Dynamic scheduling trades more messages and a dedicated coordinator for the ability to feed new work to the next worker that finishes. Too-small chunks can make dispatch and message overhead prominent. Too-large chunks can leave workers idle. If there are fewer tasks than workers, some workers cannot be active at all. The optimum depends on the workload, machine, interconnect, and worker count; the source code alone cannot establish it.

3.5 Resource accounting

Fair comparisons must separate computing capacity from total allocated ranks. Static mode uses $P = W$ total ranks for W computing workers. Dynamic mode uses $P = W + 1$ total ranks for the same W computing workers. A graph with total ranks on the horizontal axis gives the dynamic scheme one fewer computing worker at each P ; a graph with workers on the axis must still disclose the extra master resource. This report follows the project's primary worker-based comparison and also defines allocation-aware efficiency for the dynamic case [P2].

The main study is pure MPI, with one active Numba thread per rank. `performance_runtime.py` also constrains common OpenMP and BLAS thread-pool environment variables where applicable. Hybrid MPI plus multiple compute threads per rank is possible in principle, but it would be a

separate experiment. Its results cannot be mixed with the one-thread-per-rank series without recording core allocations and a new baseline.

4 Parallel random-number design

4.1 Why stream identity belongs in the method

Monte Carlo parallelism is useful only when additional workers contribute valid information. If each worker starts the same deterministic generator from the same seed and performs the same number of draws, their point sequences are repeated. Processing W copies of one stream costs roughly W times the computation but can provide much less than W times the independent information. A tiny absolute error in one duplicated run could hide this failure; it would be an accident of the chosen stream. The project therefore treats stream construction as part of the algorithm, not as an implementation footnote.

A pseudo-random generator is deterministic once its initial state is fixed. Reproducibility is desirable for debugging and rerunning experiments, but it is distinct from stream separation. A method can be reproducible and bad: the intentional `same_seed` policy always repeats the same stream. Conversely, different output prefixes are a useful check but do not prove that entire future sequences are independent. The analytical variance in Section 2 assumes independent samples, so the random-stream design and its limited checks are necessary to interpret statistical results.

4.2 Four policies in the code

Policy	Construction in <code>rng_strategies.py</code>	Purpose	Interpretation limit
<code>same_seed</code>	The same base seed initializes the default generator for every rank or task.	Deliberate duplicated-stream control.	Not a valid parallel policy for independent work.
<code>rank_seed</code>	<code>base_seed + stream_id</code> initializes the default generator.	Simple deterministic baseline.	Distinct integer seeds alone are not a proof of disjoint trajectories.
<code>seed_sequence</code>	A root <code>SeedSequence</code> spawns child states for the default generator.	Structured child-state baseline.	Finite tests cannot certify all possible streams.
<code>philox_seed_sequence</code>	Spawned child states initialize NumPy Philox generators.	Primary project policy.	Strong design and finite tests do not imply a mathematical proof of independence.

NumPy’s parallel-generation guidance recommends spawning child generators through `SeedSequence`; the child identity includes its position in the spawn tree [3]. NumPy documents Philox as a counter-based generator and demonstrates its use with spawned child sequences [4]. The broader counter-based approach was developed for scalable parallel random-number generation [2]. These sources support the design choice. The project’s own evidence is narrower: it checks selected prefixes and repeated runs, not full generator quality.

4.3 Logical stream identities

Static mode associates a child stream with the rank number. If the number of ranks changes, the partition of N changes and so do the sampled points. A repeat with the same rank count, seed, software version, and configuration should reproduce the stream assignments. The rank-based

construction is simple, but one should not interpret a result at `P = 2` and a result at `P = 4` as evaluations of exactly the same random points.

Dynamic mode associates each child stream with a logical task ID. A task can be sent to different workers on different runs, yet it retains its state and sample count. This removes worker-arrival order from stream selection. The aggregate floating-point result can still differ in last digits because task partial sums may reach the master in different orders, and floating-point addition is not associative. Reproducible streams therefore do not imply bitwise-identical final sums across every possible schedule. This distinction is particularly important when using asynchronous results as a debugging criterion.

The software prepares one child state per task for dynamic mode. This is convenient and makes identity explicit, but it gives the master state and task metadata proportional to the number of tasks. Very small chunks can therefore increase both scheduling traffic and master memory use. A scalable future design could derive a child state from a task ID as needed, while maintaining the same documented mapping and validating it against the current implementation.

4.4 What the RNG experiment tests

`scripts/statistics/rng_stream_validation.py` records a fixed root seed, stream IDs, short sample prefixes, rerun equality, and whether paired prefixes differ. Its summary tests 16 streams with prefixes of length 8. All four policies pass the fixed-seed rerun check. The two spawned policies and `rank_seed` show distinct tested prefixes; `same_seed` shows the intended duplication [P5]. These are implementation checks for the selected configuration. They do not establish zero correlation, non-overlap at every length, or a ranking of generators by scientific quality.

The project also runs 30 independent estimates at each sample count with the principal Philox policy. This provides an empirical variance comparison with the analytical model. Agreement is another useful check, but it remains finite evidence. Dedicated generator batteries, much larger repetitions, or application-specific correlation analysis would be needed for broader claims. The report avoids treating one-run absolute error as an RNG score.

5 Implementation and data flow

5.1 Program organization

The code separates the estimator, execution modes, experiment drivers, and presentation tools. `prng_math.py` contains the domain volume and exact value. `rng_strategies.py` constructs streams. `worker_compute.py` owns sample generation and local numerical reduction. `serial_baseline.py`, `static_mpi.py`, and `main_hpc.py` are the three execution entry points. `master_node.py` contains dynamic task dispatch and worker behavior. `results_io.py` defines structured output fields; benchmark and experiment scripts orchestrate repeated runs. This separation makes it possible to test mathematics and stream rules without launching MPI for every test [P1].

For a given run, command-line arguments identify the sample count, dimension, seed, RNG policy, execution mode, and relevant task or rank count. The result row also records the requested and processed samples, estimate, exact value, error measures, timing fields, and parallel resource counts. The experiment drivers add run class, repetition, ordering, and provenance fields. Storing the configuration alongside each observation is necessary because a runtime number without its process, thread, workload, and timer settings is not reproducible evidence.

5.2 Local task execution

`process_task_timed` receives a logical ID, stream state, sample count, dimension, and RNG policy. It constructs a generator, draws a `(count, dimension)` array uniformly over `[-1,1]`, and applies the selected compiled sum-of-squares kernel. It returns the scalar local sum and separate durations for generator work, kernel work, and the overall task. The serial baseline calls the serial kernel; MPI modes use the parallel-capable kernel with one configured thread in the main study [P1].

Separating generator time from kernel time is useful diagnostically, but the durations should not be added to MPI send, receive, and master intervals as though all activities were sequential and non-overlapping. A worker may wait while another worker calculates. The master may handle a result while a different task is in progress. The overall compute-phase wall time remains the primary performance measure; component fields help explain it.

The allocation shape is also a reproducibility and memory parameter. For `float64`, a point array alone uses approximately `8 N_task d` bytes, excluding Python objects, generator state, MPI buffers, and Numba work space. At `d = 10`, a task with one million points occupies about 80 million bytes for its point array. This estimate is a lower bound for peak process memory. A calibration that increases `N` should control the task size separately from the total workload rather than assuming the full problem can be materialized on every rank.

5.3 Static control flow

The static program validates inputs, configures threads, warms the local kernel, and creates one stream state per rank. A scatter distributes these states. Each rank computes its quotient-and-remainder share after a barrier. The local scalar sums are reduced to rank zero, and the maximum rank elapsed time is reported. Worker diagnostics are gathered afterward. Rank zero writes the result row and prints a human-readable summary [P1].

The final reduction is a small communication payload, but its elapsed cost includes waiting for the slowest rank as well as the collective operation. The program's residual `communication_time_seconds` cannot be interpreted as pure communication time. A clean analysis should show total runtime, task durations, and a profiler trace if the report needs to distinguish those components. The barrier aligns the starts of participating ranks; it does not remove operating-system noise during the timed region.

5.4 Dynamic control flow and termination

The dynamic master computes exact chunk sizes, including the final remainder. It sends task tuples containing ID, child state, count, dimension, policy, and seed. Workers generate and evaluate their chunks, then send a result dictionary. The master receives from any worker, adds the scalar, and assigns the next task to that worker. Once the queue is empty, it sends a stop tag. Workers that receive no initial task are stopped immediately. Each worker finally sends a statistics record [P1].

Several edge cases are important. If $N < C$, exactly one task exists. If the worker count exceeds the number of tasks, some workers are idle by design. If N is not divisible by C , the final short task must not be lost. If a worker has no task, it must not wait forever for a message that will never come. The tests exercise these cases. Any production run should also check that the sum of per-worker processed samples equals N and that task IDs are complete and unique.

The master is a possible bottleneck as task counts increase. It performs stream-state setup, dispatch, result receipt, scalar accumulation, and termination. Its use of blocking `send` and `recv` is a straightforward design with explicit tags. A future nonblocking version might overlap coordination, but would need careful correctness checks and a fair benchmark before it could be called an improvement.

5.5 Output provenance and auditability

The project distinguishes `smoke`, `exploratory`, and `production` run classes. This is more than a label: only production records with the approved workload, repetitions, environment metadata, and successful evidence gate enter final scaling claims. A stored Git commit connects data to code. Run IDs connect performance rows to worker rows. Stage 03 through Stage 05 data and logs are frozen in the two-node campaign archive; Stage 06 was separately preserved and hash-verified [P4, P6, P9].

There are older local benchmark plots in `results/figures/` and an older short PDF report. Their timings helped develop and review the software, but the later methodology rejects them as final cluster evidence. They are excluded from the two-node production series. This report uses canonical Stage 03 through Stage 05 figures for performance and approved Stage 06 figures for statistical interpretation [P6].

6 Correctness and verification strategy

6.1 Mathematical invariants

Several invariants follow from the estimator and are inexpensive to check. The exact integral must be $d^2 d/3$. Requested sample counts must be positive, and dimensions must be at least one. Every execution mode must process exactly N samples and compute the same estimator form. Static rank assignments must sum to N , even when $N < P$; dynamic chunk sizes must sum to N , including a remainder. The reported absolute error must be the absolute value of signed error, and relative error must divide by the magnitude of the exact result. These checks catch structural mistakes without depending on a lucky random outcome.

The codebase contains unit tests for mathematical constants, RNG construction, data schemas, benchmark behavior, scheduler edge cases, and statistical summaries. Opt-in MPI integration tests run real static and dynamic jobs and compare their outputs with independently reconstructed deterministic sample streams. That approach is stronger than asserting that the output is merely “near” the exact integral: a wrong stream assignment or lost remainder can still produce a superficially plausible integral. The repository includes CI configurations for quick checks, but this report does not claim that a hosted runner or a target cluster has been verified in the present workspace [P1, P8].

6.2 Random-stream and scheduling invariants

For a fixed seed, a given logical stream ID should create the same finite prefix on rerun. Under the intended spawned construction, tested different IDs should produce different tested prefixes. Under the deliberately bad `same_seed` control, their prefixes should match. In dynamic mode, a task’s stream ID should be independent of the worker that happened to receive it. The validation CSV records these finite checks [P5].

Scheduling adds accounting conditions. Every dispatched task should produce one result, and no result should be counted twice. The number of processed samples in worker diagnostics should sum to the requested total. An idle worker should receive a stop message and contribute a diagnostic record rather than blocking the job. Worker utilization should not conceal idle workers by averaging only those that received tasks. These properties are checked by code-level and integration tests, and remain useful acceptance checks for production datasets.

6.3 Statistical consistency

The completed Stage 06 experiment calculates 30 estimates at each of three preselected N values. For each level, it reports mean estimate, empirical bias, sample variance with $R - 1$ denominator, sample standard deviation, root mean squared error, analytical variance and SD, SD ratio, standard error of the mean, and a Student-t 95% interval for the repeated-run mean [P4]. These quantities answer different questions. The sample SD describes variation of a single estimate at that N ; the standard error describes uncertainty in the mean of 30 estimates. The Student-t interval is for that mean, not for every individual Monte Carlo run.

The analytical $N^{(-1/2)}$ trend provides an expected scale. With only 30 repetitions at each level, an empirical SD ratio different from exactly one is normal. The correct question is whether the observed pattern and magnitude are broadly consistent with the model, not whether every finite estimate exactly matches a formula. The data are analyzed in Section 8.

6.4 Limits of the checks

Passing deterministic tests shows that the tested cases and environments behaved as specified. It does not prove the absence of all bugs or all future stream overlap. The archived campaign records genuine two-host MPI execution, but it does not test a homogeneous, exclusively allocated supercomputer or larger worker counts. A statistical convergence plot does not validate strong scaling; the separate Stage 04 production runs support the scaling observations. The report records these boundaries because they determine what can be inferred from each type of evidence.

7 Experimental methodology

7.1 Fair comparison and timing boundary

The principal performance comparison is pure MPI with one active compute thread per rank. The serial baseline uses one process and one thread. Static mode uses one rank per computing worker. Dynamic mode uses one additional master rank. Every compared configuration must use the same dimension, total workload for strong scaling, estimator, RNG policy, and compatible timer boundary. The project uses elapsed wall time: Python’s monotonic performance counter for the serial loop and `MPI.Wtime()` for MPI intervals. The MPI standard defines `MPI_Wtime` as elapsed wall-clock seconds from a process-local reference point [5].

Numba compilation is warmed up before measured trials. This makes the primary number a steady-state compute-phase duration. It excludes interpreter and MPI launch, environment setup, CSV writing, and one-time compilation. That is a legitimate boundary for algorithmic comparison if applied consistently, but the report must never call it total end-to-end application latency. A separate launch-inclusive measurement would be required to answer a user’s first-run wait-time question.

The approved protocol calls for an unmeasured warm-up and seven measured repetitions of each final performance configuration. Configuration order is deterministically shuffled to reduce simple order bias. The summary is the median with interquartile range; failures are kept visible. A single very short timing is not enough to characterize either runtime or uncertainty. The environment record should include CPU and node type, core allocation, process placement, MPI implementation, Python and package versions, thread settings, command line, Git commit, and run ID [P2].

7.2 Calibration and crossover

The production workload is chosen by increasing the serial sample count until the median compute-phase runtime falls in an approximately 20- to 60-second band on the target hardware. This calibration avoids assuming that an arbitrary `N` is large enough to dominate noise and coordination. The selected `N_calibrated` then becomes the strong-scaling workload. If the target band is not reached, the production scaling campaign should not be called calibrated [P2, P6].

The crossover experiment measures serial, static, and dynamic compute-phase durations across increasing workloads. At each workload and worker count, repeated medians and uncertainty bands are compared. A crossover is an observed interval between tested `N` values where a parallel mode changes from slower to faster than serial under the same timing definition. Interpolation may be shown as a model, but a precise universal crossover value cannot be asserted from a sparse sweep. Because launch time is excluded, a compute-phase crossover would generally precede an end-to-end crossover; the latter requires its own measurements.

7.3 Strong scaling

Strong scaling holds the total $N = N_{\text{calibrated}}$ and $d = 10$ fixed while the number of computing workers W changes. Let T_1 be the matching serial median and T_W the median of an MPI configuration. Speedup is $S_W = T_1/T_W$; worker efficiency is $E_W = S_W/W$. The ideal compute line is $S_W = W$, but actual efficiency can decline because each worker gets less work while coordination and serial fractions persist. Amdahl's law is a useful interpretive model for such limits, not a fitted result in the absence of production data [7].

For dynamic mode, the master consumes a rank. Worker efficiency uses W , and a secondary allocation-aware efficiency uses $W + 1$. Both must be labeled. An equal-total-rank comparison and an equal-computing-worker comparison answer different resource questions. The report's main comparison uses the latter and discloses the additional master. Error bars or interquartile ranges should accompany repeated medians, and every plot should use integer worker ticks.

7.4 Weak scaling

Weak scaling holds work per **computing worker** fixed, so total samples are $N = W N_{\text{worker}}$. If runtime stayed constant as W increased, the worker-scaled throughput would grow proportionally to W . A simple weak-efficiency measure is T_1/T_W for matching per-worker work. Static mode requires W ranks; dynamic mode requires $W + 1$ total ranks. Counting the master as a computing worker would quietly change the amount of useful work per worker and invalidate the comparison.

The theoretical Monte Carlo uncertainty also changes in a weak-scaling experiment because total N grows with W . If streams are valid, larger N reduces estimator SD even if runtime stays approximately constant. This is a useful benefit of parallelism, but it should not be confused with a specialized variance-reduction method. Gustafson's scaled-work perspective helps frame the distinction between fixed-work speedup and growing-work throughput [8].

7.5 Chunk size and worker activity

The dynamic task size C governs the number of tasks $K = \text{ceil}(N/C)$. The chunk study keeps N , W , dimension, and RNG policy fixed while varying C . It records median runtime and repeated variability alongside task counts and worker diagnostics. A very small C increases scheduling and message frequency; a very large C may create too few tasks to use the workers. A coarse configuration with $K < W$ is a deliberately underfilled schedule, not evidence that all dynamic schedules scale badly.

The project defines worker task-time fractions over the master interval. These show whether workers had assigned work and how evenly it was distributed. They do not by themselves decompose communication, queue delay, or compute stalls. A profiler such as Score-P/Vampir or a more detailed event trace would be needed for a defensible communication breakdown. Section 12 reports the application-level fractions from canonical Stage 04 worker diagnostics.

7.6 Controlled scheduling stress

The normal quadratic integrand is nearly homogeneous. To evaluate the purpose of dynamic scheduling under imbalance, Stage 05 defines controlled heterogeneous task-cost profiles and runs them through real MPI code. The stress is synthetic by design: it probes scheduling behavior rather than claiming the benchmark integral naturally has those cost variations. Repeated makespan, task-event, and worker-metric files support the conditional comparisons in Section 13 [P2, P6].

The distinction between a scheduling simulation in a dashboard and an MPI execution is crucial. The Streamlit demonstration can explain queues and worker cards, but its simulated timeline is not a measured trace of the MPI program. Stage 05 supplies real-MPI task events and worker measurements from the two-node campaign. Its synthetic cost profiles and unequal static/dynamic stopwatch boundaries are disclosed with the results.

8 Statistical results

8.1 Data provenance and protocol

The completed statistical campaign is Stage 06. It uses the ten-dimensional benchmark, the `philox_seed_sequence` policy, root entropy 2026, and 30 independently identified repetitions at each of three sample counts. The raw file records run ID, Git commit, start time, sample count, repetition, spawn key, estimate, exact value, and error measures. The summary file records aggregate statistics and is marked `APPROVED_CANONICAL` in the data manifest [P4, P6]. These records allow the reported numbers to be traced to individual trial values rather than to a manually drawn trend line.

The three sample levels were 10,000, 100,000, and 1,000,000. The main question is whether the distribution of repeated estimates narrows at a rate broadly consistent with the known analytical SD. Thirty repetitions are enough to show a useful pattern and estimate a sample SD, but they are not so many that a ratio to theory should equal exactly one. The trial runs use different spawned stream identifiers; their recorded root entropy makes them reproducible.

8.2 Summary of repeated estimates

Samples N	Repeats	Mean estimate	Empirical bias	Sample SD	Analytical SD	SD ratio	RMSE
10,000	30	3413.0318	-0.3015	9.7989	9.6544	1.0150	9.6389
100,000	30	3413.9827	+0.6493	2.9903	3.0530	0.9795	3.0109
1,000,000	30	3413.3495	+0.0162	0.7869	0.9654	0.8151	0.7739

Table 1. Stage 06 repeated-estimate summary. The exact integral is 3413.3333. Sample SD uses the 29-degree-of-freedom sample variance; RMSE is calculated across the 30 errors at each N . Source: [results/stage06/summary/statistical_summary.csv](#) [P4].

The empirical sample SD falls from 9.7989 to 2.9903 when N increases tenfold, then to 0.7869 at one million. The analytical SD falls by exactly a factor of $\sqrt{10}$ for each tenfold increase. The empirical reductions are not exact because only 30 estimates were observed at each level. The SD

ratios of 1.0150, 0.9795, and 0.8151 are broadly compatible with the expected scale. The last ratio indicates a lower sample spread than the analytical value in this finite set; it does not demonstrate a faster asymptotic convergence law.

The mean estimates are near the exact integral, but their deviations are not monotone. The 100,000-sample mean is farther from the exact value than the 10,000-sample mean even though the distribution is narrower. This is not contradictory: an empirical mean from 30 random trials can land on either side of the truth. The RMSE combines spread and mean error across those trials and decreases with N in the observed data. A single error trace could have been misleading; repeated trials make the variance comparison more informative.

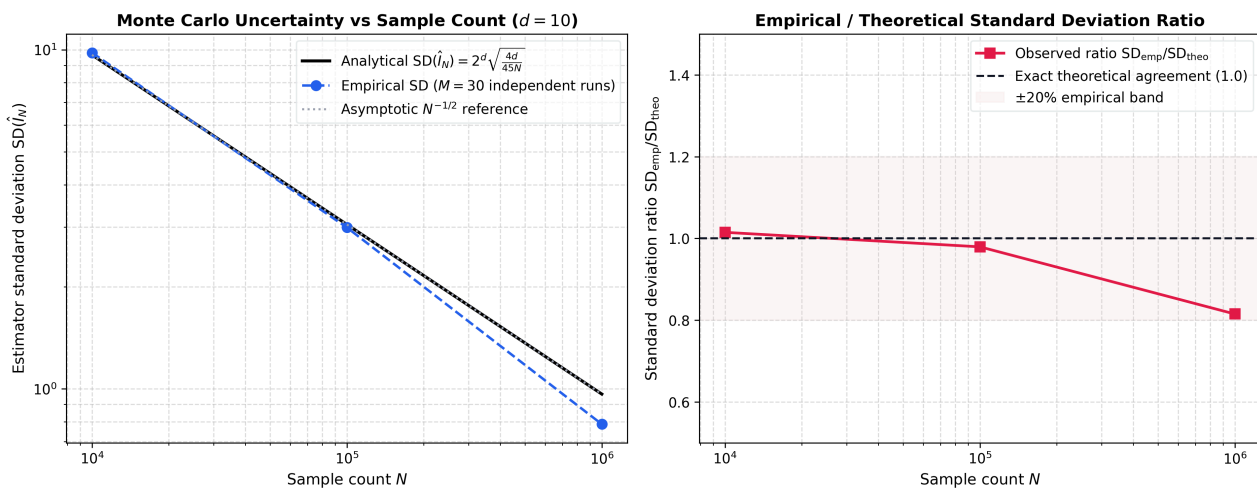


Figure 1. Stage 06 empirical and analytical standard deviation. The theoretical curve has the expected $N^{-1/2}$ form. The ratio panel is a finite-sample diagnostic, not a formal generator-quality test. Source: Stage 06 approved summary [P4].

8.3 Confidence intervals and interpretation

Samples N	Standard error of 30-run mean	95% Student-t interval for mean	Contains exact value?
10,000	1.7890	[3409.3729, 3416.6908]	Yes
100,000	0.5460	[3412.8661, 3415.0993]	Yes
1,000,000	0.1437	[3413.0557, 3413.6434]	Yes

Table 2. Student-t intervals for the mean of the 30 repeated estimates. These intervals describe uncertainty in the repeated-run mean, not the spread of one Monte Carlo estimate. Source: Stage 06 approved summary [P4].

The interval width shrinks with N because the sample SD shrinks. Each interval contains the exact value. This is consistent with the unbiased estimator model, but three interval inclusions cannot prove unbiasedness. The exact unbiasedness statement comes from the derivation in Section 2, conditional on the sampling assumptions. The empirical intervals serve as a separate implementation and finite-sample consistency check.

The absolute empirical bias is 0.3015, 0.6493, and 0.0162 across the three levels. Its non-monotone behavior is expected for a finite set of random runs. Bias and SD must not be interchanged: the former describes the signed difference between the average observed estimate

and the truth, whereas the latter describes spread among individual estimates. RMSE combines both aspects. The project's `rmse_convergence.png` displays these complementary quantities.

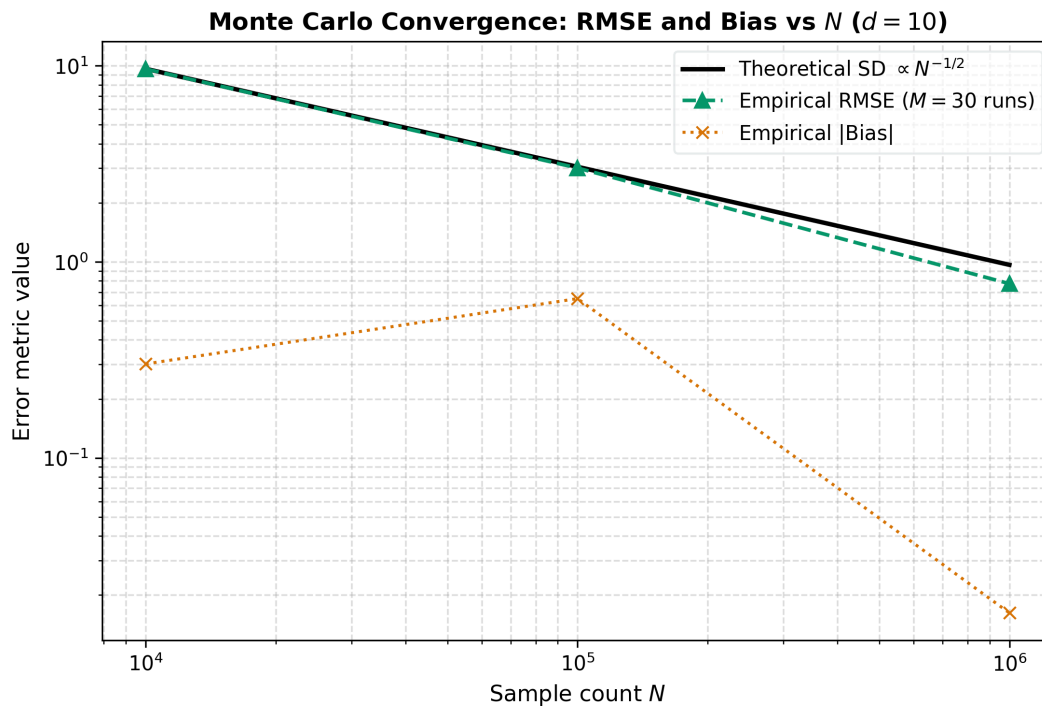


Figure 2. Stage 06 RMSE and absolute empirical bias. The single finite-run bias sequence is not expected to follow a smooth power law. Source: Stage 06 approved summary [P4].

8.4 Numerical scale and practical meaning

At $d = 10$ the integral is about 3413.33. The analytical SD at 10,000 samples is about 9.65, roughly 0.283% of the exact value. At one million samples it is about 0.965, roughly 0.0283%. The hundredfold sample increase therefore gives a tenfold theoretical precision gain, while requiring roughly a hundredfold number of point evaluations. This is the central computational pressure that motivates parallelism. A speedup would help pay for the larger sample count; it would not alter the estimator's statistical rate.

The observations are specific to the quadratic benchmark, $d = 10$, the selected sample counts, and the implemented generator. A different integrand may have much larger variance or a costly branching evaluation. The formulas for the present benchmark are exact, but one should not transfer the numerical SD values to a different problem without recomputing its variance.

9 Parallel random-stream results

9.1 Fixed-seed checks

The Stage 06 validation summary reports 16 tested streams and a prefix length of eight for each policy. Every policy is reproducible on rerun with the same base seed. This includes `same_seed`, which is intentionally invalid for parallel independent sampling. The other three policies have

distinct tested prefixes among the selected streams, whereas the `same_seed` baseline has duplicated prefixes [P5]. The contrast confirms that the test can detect the particular failure it was built to expose.

Strategy	Rerun reproducible	Tested prefixes distinct	Duplicate detected	Project interpretation
<code>same_seed</code>	Yes	No	Yes	Deliberate duplicate-stream control
<code>rank_seed</code>	Yes	Yes	No	Simple deterministic baseline
<code>seed_sequence</code>	Yes	Yes	No	Spawned default-generator streams
Spawned Philox	Yes	Yes	No	Primary <code>philox_seed_sequence</code> policy

Table 3. Stage 06 finite stream validation. “Distinct” is limited to the tested prefixes, not a claim about all infinite output sequences. Source: `results/stage06/summary/rng_validation_summary.csv` [P5].

9.2 Visual example and limits

The project’s RNG figure shows an illustrative subset of four stream IDs and six displayed values per stream. Under `same_seed`, the rows match. Under the other policies, the shown rows differ. The display makes the duplication mechanism easy to see, but it is less extensive than the summary validation and should not be described as a general statistical test. Prefix difference is necessary to rule out exact duplication over those positions; it is not sufficient to establish independence or absence of long-range correlations.

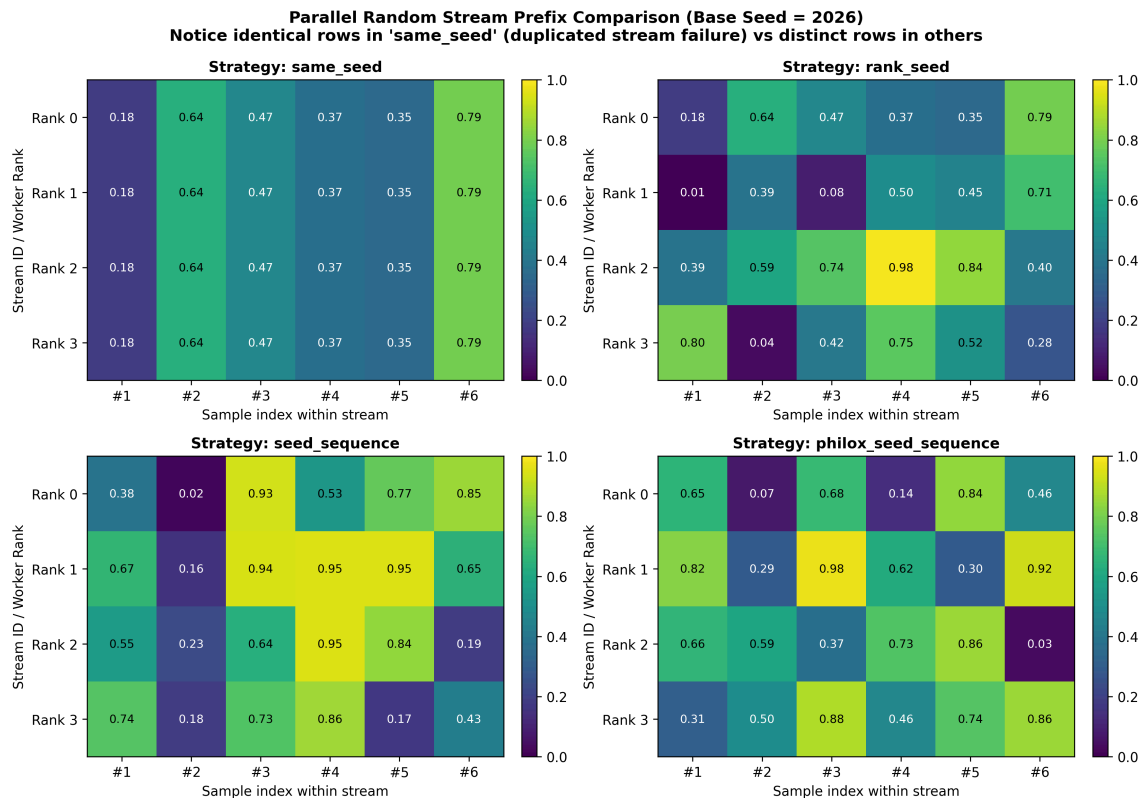


Figure 3. Illustrative Stage 06 stream prefixes. Identical `same_seed` rows display the intended failure case. The figure shows four IDs and six values; the summary validation tests 16 streams with eight-value prefixes [P5].

For the principal policy, the strongest defensible statement is that separately indexed spawned Philox streams were reproducible and had different tested prefixes, and that the 30-trial estimator spread was broadly consistent with the analytical variance. NumPy’s documentation provides the rationale for spawned stream construction [3, 4]. The project did not execute a comprehensive external randomness battery or prove statistical independence for all future trajectories. The report therefore does not claim a universal ranking among `rank_seed`, `seed_sequence`, and Philox based on a small set of integration errors.

10 Two-node cluster and evidence provenance

10.1 Measurement setting

The final performance campaign ran on two Windows hosts joined by MPI over Wi-Fi. NovaCore acted as the controller and used an AMD Ryzen 5 7235HS with four physical cores and eight logical processors. DELL used an Intel Core i7-12700H with 14 physical cores and 20 logical processors. Both hosts reported Windows 11 build 26200, Python 3.14.7, NumPy 2.4.5, Numba 0.67.0, mpi4py 4.1.2, and Microsoft MPI 10.1.12498.52. The measured Stage 03 through Stage 05 source commit was `23e27a2ee45f7c95bb9777d9dd0e76b3af0c5a56` [P6, P9].

Host	Role	Processor	Physical cores	Place in four-worker jobs
NovaCore	Controller	AMD Ryzen 5 7235HS	4	Three computing ranks; dynamic master also on this host.
DELL	Second node	Intel Core i7-12700H	14	One computing rank.

Table 4. Two-node placement for the four-worker runs. At two workers, one computing rank ran on each host. The dynamic mode always used one additional master rank. The Wi-Fi link speed was not measured [P9].

The campaign confirmed actual two-host execution before scientific timing. An archived rank-and-host check shows the processes reporting the same source SHA on the two machines; a cross-host MPI collective returned the expected sum. The operating-system firewall remained enabled with restricted peer access for the MPI sockets. These records support the claim that the timing data came from two machines rather than from multiple local ranks on one host. The campaign also preserved exact commands, run logs, rank placement, and file hashes [P9].

No physical cores were oversubscribed, and the campaign set `NUMBA_NUM_THREADS`, `OMP_NUM_THREADS`, `MKL_NUM_THREADS`, and `OPENBLAS_NUM_THREADS` to one. However, processes were not bound to exclusive cores, and the hosts were not reserved from other system activity. The CPUs differ substantially. These facts matter for the observed nonmonotone scaling, especially the two-worker static case. They also prevent extrapolating the results to a homogeneous cluster with a measured high-speed interconnect.

10.2 Campaign design and separation of evidence

Stage 03 was an exploratory workload sweep and calibration. It completed 147 of 147 trials, seven repetitions for each configuration. Stage 04 was the production scaling campaign: 26 configurations, seven measured repetitions each, 182 successful trials, and 322 linked dynamic-worker records. Stage 05 used real MPI processes for four synthetic cost profiles and two

scheduling modes, seven repetitions per case: 56 successful runs, 3,584 task events, and 224 worker records. Stage 06 was collected earlier, preserved, and hash-verified rather than rerun on the two-node cluster [P6, P9].

The numerical records have distinct scopes. Stage 03 establishes calibration and explores a crossover. Stage 04 supports production runtime, strong and weak scaling, chunk-size, and worker-activity observations. Stage 05 tests controlled scheduler response to synthetic heterogeneity. Stage 06 tests statistical convergence and finite random-stream behavior. Combining them into one undifferentiated dataset would obscure different seeds, code commits, timing definitions, and scientific questions. The current claim-evidence matrix identifies those boundaries explicitly [P6].

The campaign retained unsuccessful setup attempts. An initial 100-million-sample run failed on a DELL NumPy allocation of about 1.86 GiB. The successful calibration used 60 million samples and a memory-aware placement. Two Stage 04 partial starts with unknown Git provenance were archived and excluded from production results. This is a useful quality-control practice: a clean final CSV should not imply that every attempted command succeeded or that unsuitable data disappeared without explanation [P9].

10.3 Timing and derived measures in the final data

All primary runtimes are wall-clock seconds after Numba warm-up. Stage 04 fixed-workload speedup is the **median of seven same-repetition ratios** $T_{\text{serial}}/T_{\text{MPI}}$, not generally the ratio of two separately pooled medians. Worker efficiency divides that paired speedup by the number of computing workers. Dynamic mode's extra master rank is disclosed and is excluded from worker efficiency. The separate allocation-aware efficiency $S/(W+1)$ can be calculated to describe total-rank use, but it should not be substituted for the reported worker metric [P2, P6].

The median and interquartile range (IQR) summarize runtime distributions without assuming normality. Seven repetitions still leave considerable uncertainty, especially when the hosts are not isolated. Differences of a few tenths of a second with overlapping spreads should be described as observed orderings, not general optima. The figures show uncertainty where the source series supports it. No curve in this report is presented as a model fitted to unseen machines.

11 Stage 03 workload calibration and crossover

11.1 Calibrated serial workload

The method called for a serial runtime between approximately 20 and 60 seconds on the measurement system. Stage 03 tested increasing sample counts and first reached that band at 60,000,000 samples. The median true-serial compute-phase time was 22.511 seconds, with an IQR of 3.040 seconds. `results/stage03/summary/calibration.json` records `target_band_reached=true` and `calibrated_points=60000000` [P6]. This calibrated value then became the fixed total workload for Stage 04 strong scaling.

The calibration has two roles. It reduces the risk that an extremely short run is dominated by timer resolution or transient disturbances, and it gives the performance study a workload tied to the actual hardware rather than to an arbitrary round number. It is not a performance result for every computer. Different CPUs, memory capacities, process placement, or library versions would require a new calibration. The failed 100-million-sample allocation attempt also shows that runtime selection must be paired with memory checks.

11.2 Exploratory serial and static MPI comparison

Mode at 60 million samples	Computing workers	Total MPI ranks	Median runtime (s)	IQR (s)	Paired median speedup
True serial	1	1	22.511	3.040	1.000
Static MPI	2	2	33.806	2.830	0.706
Static MPI	4	4	13.622	0.372	1.661

Table 5. Stage 03 exploratory results at the calibrated workload. The paired speedups use the Stage 03 serial observations, not the later Stage 04 serial median [P6, P9].

Four-worker static MPI was faster than the serial program at this calibrated workload; two-worker static MPI was slower. This is an important observed counterexample to a simple “more ranks means faster” story. The two-worker arrangement placed one computing rank on each unlike host. Differences in CPU speed, network path, memory behavior, scheduling, or system activity could contribute. The campaign does not isolate a single cause, so the report does not ascribe the slowdown to one mechanism.

The first observed paired-median speedup above one in the exploratory sweep occurred at one million samples for two static workers and at 100,000 for four static workers. Timing distributions overlap at those candidate points, and the two-worker mode was slower again at the larger calibrated workload. A single precise crossover threshold would therefore be unstable and misleading. The defensible statement is that the calibrated four-worker configuration gave an observed benefit under this placement, whereas the two-worker configuration did not at that workload [P6].

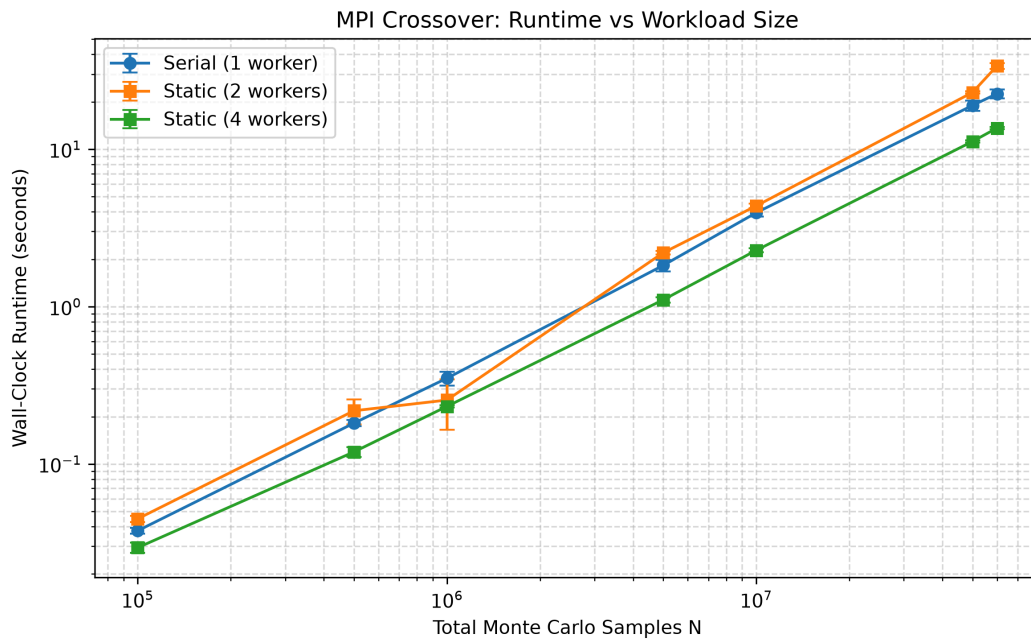


Figure 4. Exploratory workload sweep on the two-node system. Error bars summarize repeated timing variation. Apparent low-workload crossings are not treated as stable thresholds [P6].

11.3 What the crossover means

The measured Stage 03 durations exclude launch and warm-up. They describe a steady-state computational boundary. A user who starts a fresh Python and MPI job for only a small calculation will also pay interpreter, process-launch, and compilation costs. The end-to-end workload threshold for an actual first invocation may therefore differ from the compute-phase threshold. Stage 03 includes a separate lower-workload end-to-end sweep in the archive, but it is not merged into the calibrated compute-phase comparison because its timing boundary is different [P9].

The crossover also depends on worker count and placement. The four-worker mode here placed three ranks on NovaCore and one on DELL, while the two-worker mode placed one on each. These are not a simple nested addition of identical processors. A final diagram or verbal conclusion should therefore name the system and rank placement with the threshold statement. The production campaign in the next section supplies a more controlled fixed-workload comparison at the calibrated N .

12 Stage 04 production scaling and scheduling

12.1 Strong scaling at fixed workload

Stage 04 held the total workload at 60 million samples and used seven measured repetitions for each configuration. The true-serial production median was 23.748 seconds. At four computing workers, static mode had median 13.180 seconds and dynamic mode 9.470 seconds. The dynamic

job used five total ranks, including its master. The complete series shows a nonmonotone static result: two static workers took 32.244 seconds, slower than both one static rank and the true-serial program [P6].

Execution mode	Workers	Total ranks	Runtime median (s)	Q1-Q3 (s)	Paired speedup	Worker efficiency
True serial	1	1	23.748	22.784-23.835	1.000	1.000
Static MPI	1	1	24.499	24.119-24.852	0.948	0.948
Static MPI	2	2	32.244	29.112-33.979	0.716	0.358
Static MPI	4	4	13.180	13.165-13.207	1.802	0.450
Dynamic MPI	1	2	31.404	30.589-31.554	0.756	0.756
Dynamic MPI	2	3	19.367	18.871-19.588	1.201	0.600
Dynamic MPI	4	5	9.470	9.307-9.588	2.483	0.621

Table 6. Stage 04 production strong scaling at $N = 60,000,000$. Speedup is the median of paired seven-repetition serial/MPI ratios; efficiency divides by computing workers. The dynamic master is an additional allocated rank [P6].

The one-rank static time is slightly above true serial because they are distinct program paths. The dynamic one-worker configuration takes still longer and allocates an extra master. At two workers, dynamic mode is faster than serial but static mode is slower. At four workers, both are faster than serial, and dynamic has the lower observed median. No measured configuration approaches ideal fourfold speedup. These results should not be smoothed into a monotone curve or interpreted as a universal statement that dynamic scheduling beats static scheduling on a homogeneous workload.

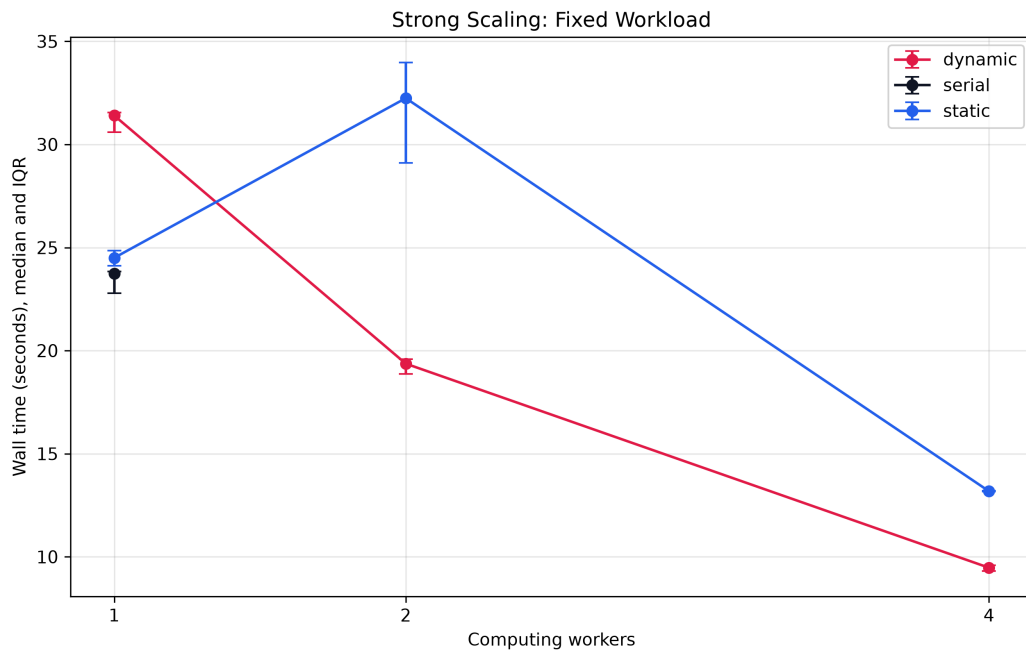


Figure 5. Strong-scaling runtime. The static two-worker slowdown is retained because it is present in the production data. Dynamic uses one more total MPI rank than the worker-axis value [P6].

12.2 Speedup and efficiency

The Stage 04 paired four-worker speedups are 1.802 for static and 2.483 for dynamic. Worker-based efficiencies are 0.450 and 0.621. The dynamic allocation-aware efficiency is approximately $2.483/5 = 0.497$, because five ranks were allocated. That derived value answers a different question from the worker metric; it is not an alternative way to claim that each computing worker was 49.7% utilized. Amdahl's law helps explain why serial work and overhead can limit fixed-work speedup, but these data cannot isolate a unique serial fraction [7].

The speedup series is not simply the serial median divided by each mode's median. The code pairs each MPI repetition with the corresponding serial repetition before taking the median of the ratios. This matters when runtime varies: $\text{median}(a_i/b_i)$ need not equal $\text{median}(a_i)/\text{median}(b_i)$. The report uses the project's stored paired-speedup summary [P6].

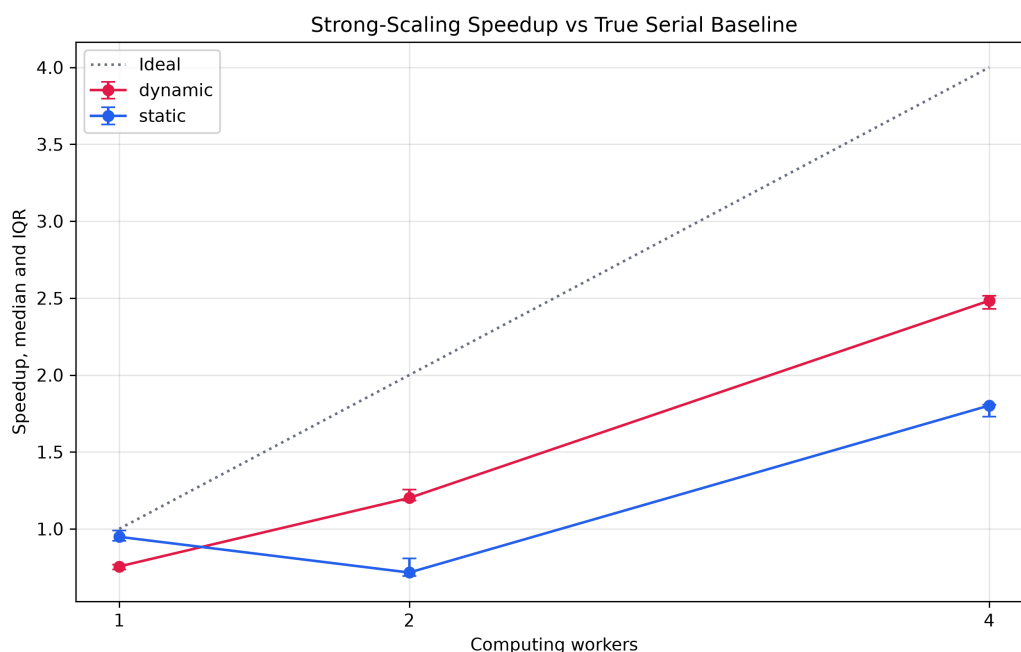


Figure 6. Fixed-workload paired speedup, with ideal worker scaling shown as context. The ideal line is a reference rather than a measured expectation for the two-host allocation [P6].

The two-worker static result is especially diagnostic: its median speedup of 0.716 means it ran slower than serial. This cannot be explained by a smooth overhead curve alone without examining host placement and workload distribution. The worker-level records, machine differences, and absence of exclusive binding are relevant. The data show the anomaly; they do not establish its cause. Future repetitions on homogeneous, bound cores would be needed to test whether it persists.

12.3 Weak scaling

The weak-scaling campaign held 15 million samples per computing worker, so total samples were 15, 30, and 60 million as w increased from one to two and four. Weak efficiency uses a same-mode, same-repetition one-worker reference; it is therefore a different metric from the true-serial

strong speedup. Static median runtime grew from 5.733 to 13.274 seconds and remained about 13.270 at four workers. Dynamic grew from 7.643 to 9.822 and then was 9.397 seconds at four workers [P6].

Mode	Workers / total ranks	Total samples	Median runtime (s)	Weak efficiency
Static	1 / 1	15,000,000	5.733	1.000
Static	2 / 2	30,000,000	13.274	0.436
Static	4 / 4	60,000,000	13.270	0.432
Dynamic	1 / 2	15,000,000	7.643	1.000
Dynamic	2 / 3	30,000,000	9.822	0.787
Dynamic	4 / 5	60,000,000	9.397	0.822

Table 7. Stage 04 weak scaling. Efficiency uses a paired one-worker reference within the same mode. Total samples are proportional to computing workers, and dynamic total ranks include the master. Ideal weak scaling would keep time near the one-worker value [P6].

The dynamic series is closer to a flat runtime than the static series on this allocation, but neither achieves ideal weak scaling. A four-worker dynamic weak efficiency of 0.822 means the median paired same-mode baseline time was about 82.2% of the four-worker time at four times the samples. The result is descriptive for unlike CPUs, Wi-Fi, and the selected rank placement. It does not predict behavior at eight, 16, or more workers on a different cluster.

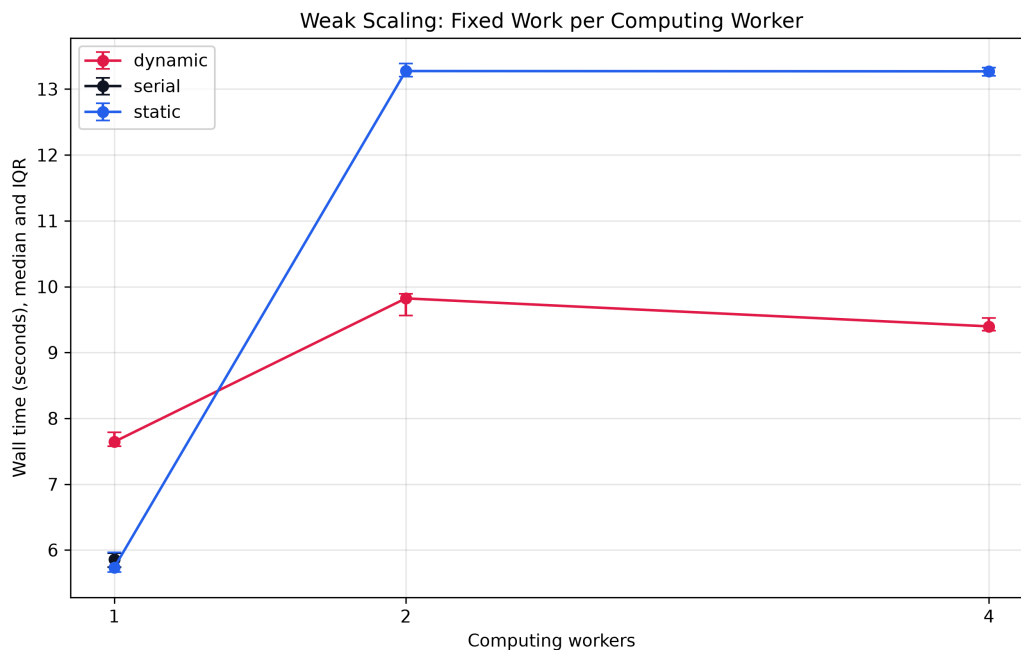


Figure 7. Weak-scaling comparison. The horizontal axis counts computing workers; dynamic mode has one further coordinator rank [P6].

12.4 Static versus dynamic at equal computing-worker count

At $W = 4$, the observed homogeneous-integrand medians were 13.180 seconds for static and 9.470 for dynamic. Dynamic was faster in this production campaign even though a uniform-cost integrand might suggest static's lower-message architecture. The two programs have different

paths, and dynamic uses a fifth rank. Heterogeneous host speeds can also make dynamic assignment useful without any per-sample cost variation. The present data do not isolate which contribution dominates. A fair statement is that dynamic had a lower observed median at the tested equal computing-worker count on this two-node system [P6].

The opposite result in the synthetic homogeneous Stage 05 control does not invalidate Stage 04. The workload, timer, and task pipeline differ. It instead shows that an architecture label alone does not determine performance. The discussion in Section 14 compares the studies only within their respective measurement definitions.

12.5 Chunk-size sensitivity and worker activity

For dynamic $w = 4$ at 60 million samples, the chunk-size campaign tested 234,375, 468,750, 937,500, and 1,875,000 samples per task. The medians were 9.207, 9.271, 9.446, and 9.515 seconds. The smallest tested chunk had the lowest observed median in this set. The seven-run IQRs overlap, so the data do not establish a stable optimum or a rule to use the smallest possible chunk [P6].

Chunk samples	Approximate task count	Median runtime (s)	Q1-Q3 (s)
234,375	256	9.207	9.158-9.409
468,750	128	9.271	9.217-9.308
937,500	64	9.446	9.137-9.518
1,875,000	32	9.515	9.439-9.596

Table 8. Stage 04 dynamic chunk experiment at four computing workers. Task counts equal N/C for these selected values; timing intervals overlap [P6].

The median per-run mean worker task-time fraction in strong scaling was 0.990, 0.977, and 0.956 at one, two, and four dynamic workers. At four workers, the median per-run minimum and maximum worker fractions were 0.927 and 0.988. These are accumulated task elapsed times divided by the master's scheduling interval and capped at one, not CPU occupancy or a direct estimate of MPI message time. The records also retain per-worker samples, chunks, and receive-wait durations for deeper analysis [P6].

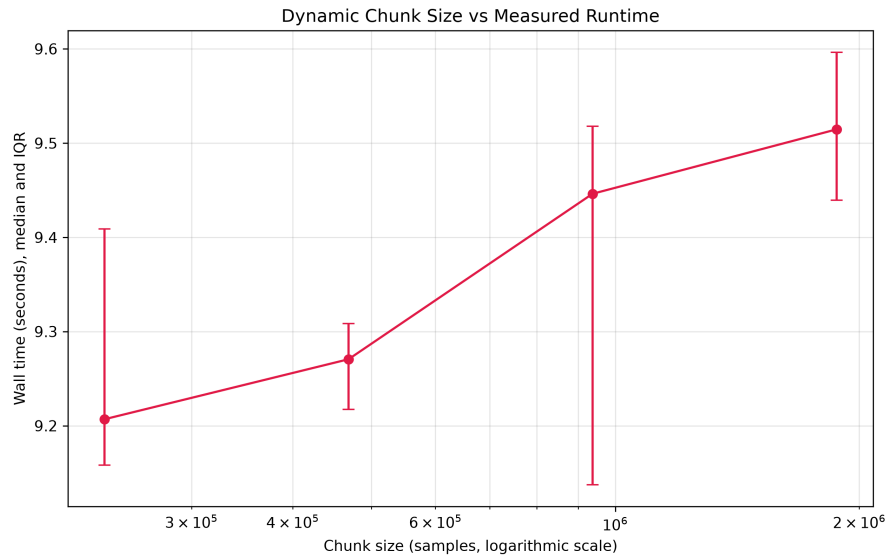


Figure 8. Chunk-size sensitivity. The lowest observed median is at 234,375 samples, with overlapping spreads across the tested settings [P6].

13 Stage 05 real-MPI scheduling stress

13.1 Controlled task-cost profiles

The production integrand has nearly uniform per-sample arithmetic cost. Stage 05 therefore introduced synthetic task-cost profiles to ask whether dynamic dispatch helps under planned imbalance. Each real-MPI run processed 64 seeded tasks with four computing workers. Static mode used four ranks; dynamic used four workers plus one master, five ranks. The profiles were homogeneous, alternating, clustered, and long-tailed. Each mode-profile combination had seven repetitions [P6, P9].

These runs were not merely dashboard animations. The archive contains 3,584 task events and 224 worker records, with task assignment, work units, and activity measures. The homogeneous profile serves as a control. The other profiles place expensive tasks in different patterns, which affects whether a static assignment leaves one worker finishing much later than its peers. Dynamic mode can reassign new tasks after a completion, but the master and messages cost time.

13.2 Makespan and utilization results

Synthetic profile	Static median makespan (s)	Dynamic median makespan (s)	Dynamic change	Static / dynamic mean utilization
Homogeneous	0.726	0.910	+25.3%	0.944 / 0.873
Alternating	1.441	1.010	-29.9%	0.495 / 0.798
Clustered	0.694	0.903	+30.1%	0.941 / 0.905
Long-tailed	0.595	0.540	-9.3%	0.836 / 0.850

Table 9. Seven-run median real-MPI synthetic stress results. Negative dynamic change means the dynamic median was lower. The two modes have different stopwatch boundaries, so the percentages are descriptive rather than clean causal scheduler-overhead estimates [P6].

The alternating profile shows the largest observed dynamic benefit: median makespan fell from 1.441 to 1.010 seconds, and the application-level mean utilization was higher. Static allocation in this profile left a large imbalance. Dynamic mode also had a modest lower median for long-tailed work. By contrast, static mode was faster for the homogeneous control and the clustered profile. The result supports a conditional scheduling interpretation: dynamic assignment can help particular distributions of expensive tasks, but it is not universally faster. It should not be projected onto the smooth quadratic integrand as though that integrand naturally had alternating costs.

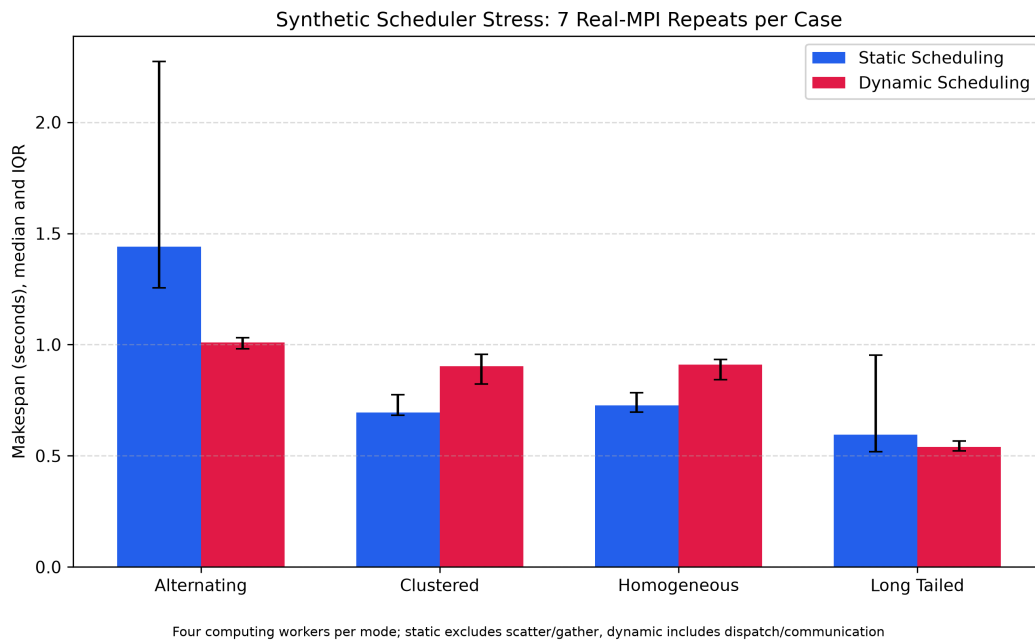


Figure 9. Real-MPI scheduler stress. The plot itself notes the unequal static and dynamic timing boundaries. Bars show observed medians, and whiskers show IQRs [P6].

13.3 Stopwatch boundary and causal caution

Static stress makespan is the largest post-barrier worker task time and excludes scatter/gather. Dynamic stress makespan begins at first dispatch and ends after final result receipt, including dispatch and result traffic but excluding process startup. These are not identical boundaries. Differences in Table 9 therefore describe the implemented profiles and reported metrics. They do not by themselves estimate a pure scheduler overhead or prove that one policy would win under a common end-to-end stopwatch [P6, P9].

Stage 05 utilization is accumulated task-kernel time divided by the run's reported makespan; inferred idle time is the difference between makespan and active time. This is an application accounting measure, not a measurement of operating-system CPU occupancy. It is useful for comparing the designed task distributions but should be accompanied by the boundary disclosure. The representative Gantt and active-versus-idle figures in the archive can help inspect why a particular profile behaved differently, but the report's central evidence remains the repeated makespan table and raw task events.

14 Discussion

14.1 What was demonstrated

The project demonstrated a numerically checkable estimator, two working MPI distribution strategies, genuine two-host MPI execution, and a staged evidence chain. The Stage 03 serial calibration reached its specified timing band at 60 million samples. Stage 04 showed that four computing workers accelerated the true-serial program in both static and dynamic modes on the tested allocation, with paired speedups of 1.802 and 2.483. Stage 05 showed that the benefit of dynamic assignment depends on the synthetic task-cost pattern. Separately frozen Stage 06 results support the expected decline of estimator spread and the finite random-stream checks. These conclusions rest on different files and measurement definitions and should remain separately traceable [P4, P5, P6, P9].

The two-worker static slowdown is part of the result. At the calibrated workload, it was slower than serial in Stage 03 and Stage 04. Its presence means a simple story of monotone speedup would misrepresent the experiment. A plausible explanation involves unlike CPUs, placement, memory, or unbound system load, but the current campaign does not isolate cause. The correct next scientific test would hold process placement, core affinity, memory pressure, and host class more tightly while repeating the same workload.

14.2 Why the architecture comparison is conditional

The static design has one final reduction and uses every rank for numerical work. The dynamic design has repeated task/result messages and a dedicated master. On an equal-worker axis, dynamic therefore consumes an extra rank. Despite that cost, dynamic $w = 4$ had the lower production median on the heterogeneous two-node allocation. A faster worker can accept more tasks while a slower worker accepts fewer; the program's task-based design permits that adjustment. This is a reasonable interpretation, but the campaign does not prove it was the sole cause of the observed difference.

The synthetic stress cases add nuance. Dynamic wins strongly for alternating cost but loses in homogeneous and clustered cases under the reported stopwatch rules. The pattern shows that load-balancing value depends on where expensive tasks fall, how tasks are assigned, and what communication is counted. It does not show that the production quadratic integrand has a 29.9% dynamic scheduler advantage. Nor does the Stage 04 outcome imply static MPI is inherently worse. Both conclusions would exceed the measured scope.

14.3 Linking performance to numerical precision

At the principal dimension, the analytical standard deviation is proportional to $1/\sqrt{N}$. More samples reduce uncertainty but increase work. Four-worker acceleration can help reach a larger N in a similar compute-phase time, if the streams remain valid and memory limits are respected. The Stage 06 statistical experiment and Stage 04 performance campaign together motivate a time-to-precision question. They do not directly measure one combined time-to-target-SD experiment,

because Stage 06 was gathered separately and the campaigns use different commits and environments. A future study could compute theoretical target N and then measure end-to-end wall time at that N for each mode on the same frozen system.

The random-stream controls remain essential in such a study. Repeated streams would spend time on nominally additional samples without the expected gain in independent information. The `same_seed` control makes this failure visible at the prefix level. The spawned Philox construction and observed Stage 06 variance give project-level support for the intended policy, with finite-test limitations.

14.4 Presentation of uncertainty and negative results

The academic value of a performance report depends on showing variation, not just attractive medians. Stage 03 crossover candidates have overlapping distributions; Stage 04 chunk-size spreads overlap; Stage 05 long-tailed difference is modest. A four-worker dynamic median of 9.470 seconds is meaningful for the tested allocation, but its seventh trial is still only one observation under a noisy environment. The report therefore uses medians, IQRs, named baselines, and explicit resource counts. It retains slower configurations and failed setup attempts rather than selecting only the best outcomes.

15 Limitations and future work

15.1 Hardware and scaling scope

Only two unlike Windows hosts were available for the cluster campaign. The Wi-Fi link speed was not measured. Processes stayed within physical-core counts but lacked exclusive processor affinity and exclusive machine reservations. Only one, two, and four computing workers were used in the main scaling series; the dynamic mode added one master. These facts prevent claims about large homogeneous supercomputers, multi-rack networks, or asymptotic strong-scaling limits. The work demonstrates a concrete two-host result.

The 100-million-sample allocation failure shows a practical memory boundary. The present worker materializes each chunk's sample array, so peak memory depends on task size and dimension. Stage 03 selected a feasible 60-million-sample total after a failed larger attempt. This does not prove that arbitrary larger N will run without modification. Bounded internal sub-batching, while preserving stream identity and sample counts, would be a useful engineering extension.

15.2 Interpretation limits of the experiments

Stage 03 is exploratory. Its apparent early crossover points have overlapping timing distributions and are not stable thresholds. Stage 04 compares distinct serial, static, and dynamic program paths; a practical paired speedup includes their implementation differences, not only MPI overhead. Dynamic worker efficiency excludes the extra coordinator rank unless allocation-aware

efficiency is named separately. Stage 05 profiles use controlled synthetic costs and unequal static/dynamic stopwatch boundaries. These limitations qualify the reported comparisons rather than making the observations unusable.

Stage 06 uses only 30 independent estimates at each N , and its RNG test checks finite prefixes of 16 streams. The empirical SD ratios need not equal one exactly. Tested prefix differences cannot prove global independence, zero correlation, or universal generator quality. The Stage 06 files were preserved from a different source commit than the two-host campaign. The report therefore uses them for statistical and stream behavior, not as direct timing evidence on NovaCore and DELL.

15.3 Next research steps

A controlled follow-up should repeat strong and weak scaling on homogeneous hosts with explicit rank binding, a measured network, and exclusive allocations. It should collect launch-inclusive and compute-phase times side by side, so users can understand first-run latency as well as steady-state throughput. A common end-to-end stopwatch for static and dynamic scheduler stress would make a cleaner causal comparison. Per-task and MPI event tracing could separate computation, message transfer, master service time, and worker waiting more precisely.

Further numerical work could test discontinuous or branch-heavy integrands whose costs vary naturally, and separately evaluate variance-reduction methods such as control variates, stratification, or quasi-Monte Carlo. Those would be new algorithms and require new accuracy analysis. The present results should remain the baseline for the existing smooth quadratic integrand rather than being generalized to those future problems in advance.

16 Conclusion

The group built a distributed Monte Carlo integration system with a true one-thread serial baseline, static MPI partitioning, dynamic master-worker scheduling, structured random streams, and testable numerical output. Its exact ten-dimensional answer is $10240/3$, and the analytical estimator SD supplies a reference for repeated trials. The project also recorded genuine two-node execution, frozen production data, and separate statistical validation.

On the tested heterogeneous two-host Windows cluster, the calibrated serial workload was 60 million samples. At that workload, Stage 04 production four-worker static and dynamic MPI were faster than true serial, with paired speedups of 1.802 and 2.483; dynamic used an extra master rank. Two-worker static MPI was slower than serial. Weak scaling was closer to flat in dynamic mode on this placement. In synthetic real-MPI stress, dynamic had lower observed makespans for alternating and long-tailed profiles, while static had lower observed makespans for homogeneous and clustered profiles. The results therefore support conditional, configuration-specific conclusions rather than a universal scheduler ranking.

Separately, 30 repeated estimates at each of three sample counts showed decreasing spread broadly consistent with the analytical $N^{(-1/2)}$ law. Finite random-stream checks reproduced intended child streams and detected the deliberately duplicated `same_seed` control. These are

limited but important statistical checks. Together, the numerical and performance evidence show why a useful parallel Monte Carlo result needs both valid new samples and a measured reduction in elapsed time, with the tested hardware, timing boundary, and allocation stated alongside every speedup.

References

- [1] N. Metropolis and S. Ulam. “The Monte Carlo Method.” *Journal of the American Statistical Association* 44, no. 247 (1949): 335-341. DOI: [10.1080/01621459.1949.10483310](https://doi.org/10.1080/01621459.1949.10483310).
- [2] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. “Parallel Random Numbers: As Easy as 1, 2, 3.” *Proceedings of SC '11* (2011). DOI: [10.1145/2063384.2063405](https://doi.org/10.1145/2063384.2063405).
- [3] NumPy Developers. “Parallel Random Number Generation.” *NumPy Reference Manual*. numpy.org/doc/stable/reference/random/parallel.html. Accessed 30 September 2026.
- [4] NumPy Developers. “Philox Counter-Based RNG.” *NumPy Reference Manual*. numpy.org/doc/stable/reference/random/bit_generators/philox.html. Accessed 30 September 2026.
- [5] MPI Forum. *MPI 4.1 Standard*, Section on `MPI_Wtime`. mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf. Accessed 30 September 2026.
- [6] Numba Developers. “Automatic Parallelization with `@jit`.” *Numba Documentation*. numba.readthedocs.io/en/stable/user/parallel.html. Accessed 30 September 2026.
- [7] G. M. Amdahl. “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.” *AFIPS Conference Proceedings* 30 (1967): 483-485. DOI: [10.1145/1465482.1465560](https://doi.org/10.1145/1465482.1465560).
- [8] J. L. Gustafson. “Reevaluating Amdahl’s Law.” *Communications of the ACM* 31, no. 5 (1988): 532-533. DOI: [10.1145/42411.42415](https://doi.org/10.1145/42411.42415).

Project sources

- [P1] Project repository implementation: `prng_math.py`, `rng_strategies.py`, `worker_compute.py`, `serial_baseline.py`, `sequential.py`, `static_mpi.py`, `main_hpc.py`, `master_node.py`, `performance_runtime.py`, `results_io.py`, and the test files. The measured Stage 03-05 source commit is recorded separately in [P9].
- [P2] Project methodology: `docs/methodology/performance-methodology.md`, `docs/methodology/statistical-methodology.md`, `docs/methodology/results-communication.md`, and `docs/methodology/README.md`.
- [P3] Project experiment guides: `docs/experiments/README.md` and `docs/experiments/statistical-rng-validation.md`.

[P4] Approved Stage 06 statistical evidence: `results/stage06/raw/statistical_runs.csv`, `results/stage06/summary/statistical_summary.csv`, and the statistical figures under `results/stage06/figures/`.

[P5] Approved Stage 06 RNG evidence: `results/stage06/raw/rng_stream_checks.csv`, `results/stage06/summary/rng_validation_summary.csv`, and `results/stage06/figures/rng_reproducibility.png`.

[P6] Frozen claim and status controls: `docs/analysis/cluster-claim-evidence-matrix.md`, `docs/analysis/cluster-research-answers.md`, `docs/analysis/cluster-limitations.md`, `docs/analysis/STAGE07_STATUS.md`, `docs/analysis/stage07-preflight.json`, `docs/analysis/canonical-data-manifest.csv`, and `docs/FINAL_SUBMISSION_MANIFEST.md`.

[P7] Experiment drivers and job definitions: `scripts/crossover/crossover_experiment.py`, `scripts/benchmarking/benchmark.py`, `scripts/scheduling/scheduler_stress_experiment.py`, `scripts/statistics/statistics_experiment.py`, `scripts/statistics/rng_stream_validation.py`, and `jobs/monte_carlo_production.slurm`.

[P8] Repository documentation and verification: `README.md`, `.gitlab-ci.yml`, `.github/workflows/windows_mpi_ci.yml`, and the `tests/test_*.py` suite.

[P9] Two-node measurement archive: `results/cluster_campaign_20260926_154803/CAMPAIGN_REPORT.md`, archived rank and collective proofs, exact commands, raw logs, frozen Stage 03-05 data and figures, and their SHA256 manifests. Measured source commit: `23e27a2ee45f7c95bb9777d9dd0e76b3af0c5a56`.

Appendix A Work sharing

The following table summarizes the primary responsibility areas documented during the project. Software development, experimentation, analysis, and preparation of the final report were collaborative activities. The listed areas indicate primary responsibility rather than exclusive authorship [P2].

Group member	Student number	Documented primary methodology area
Carl Etornam Yevutsey	17051097	Performance and MPI methodology, including timing boundaries and comparison design.
Evans Boateng	22376083	Results and communication methodology, source register, figures, and claim discipline.
Ernest Osei Amofo	13347342	Mathematics, estimator variance, repeated statistical protocol, and RNG interpretation.

The software, experiments, analysis, and report are presented as group work. The table above records documented primary responsibility areas and does not imply exclusive ownership of implementation, data collection, writing, or presentation tasks.

Appendix B Reproducibility protocol

B.1 Software and machine record

A reproducing team should first identify a Python installation with NumPy, Numba, mpi4py, pandas, matplotlib, SciPy, Streamlit, and pytest versions compatible with `requirements.txt`. It should also record the MPI runtime and launcher, operating system, CPU model, node count, allocated cores, process binding, memory, and the Git commit used. The Stage 06 raw file records its earlier source commit. Stage 03 through Stage 05 carry the later measured source SHA and two-host environment metadata in the frozen archive [P2, P4, P9].

Thread settings are part of the configuration. Primary comparisons use one active Numba thread and constrain common BLAS and OpenMP pools to one where applicable. The benchmark child processes should inherit these settings before importing numerical libraries. The command-line `-threads-per-rank 1` option documents the intended active count, but an allocation record is still needed to establish that processes did not oversubscribe physical cores.

B.2 Minimal numerical smoke test

From the project root, the following commands demonstrate the three execution paths on a machine with a working MPI installation. They are examples of how to run the code, not substitutes for the production protocol. On Windows, use an MPI launcher such as `mpiexec`; on a cluster, use the launcher specified by the allocation.

```
python sequential.py --points 100000 --dimensions 10 --seed 2026 --output results/serial_check.csv
mpiexec -n 4 python static_mpi.py --points 100000 --dimensions 10 --seed 2026 --output
results/static_check.csv
mpiexec -n 5 python main_hpc.py --points 100000 --chunk-size 10000 --dimensions 10 --seed 2026 --output
results/dynamic_check.csv
```

The first command is one process. The second gives static mode four computing ranks. The third gives dynamic mode four computing workers plus one master. The requested and processed sample counts should match, and all estimates should be interpreted relative to the exact value $10240/3$. Because partitioning changes streams, the three estimates need not be bitwise equal. A smoke test checks execution and basic accounting, not performance or statistical validity by itself.

B.3 Statistical campaign

The Stage 06 experiment can be regenerated using the documented scripts, provided output locations are chosen so that the approved existing files are not overwritten unintentionally. The intended campaign uses 30 repetitions at each sample level and then summarizes its raw records. A fresh run should have a new run ID and its own manifest. If its numbers differ slightly, that difference must be explained in terms of seed, code, package versions, or execution settings before comparisons are made.

```
python -m scripts.statistics.statistics_experiment --points 10000 100000 1000000 --repetitions 30 --
output-dir results/stage06_reproduction
python -m scripts.statistics.rng_stream_validation --output-dir results/stage06_reproduction
python -m scripts.statistics.statistical_visualizer
```

For a strict reproduction, compare the raw and summary schemas, exact integral, sample counts, root entropy, repetition IDs, spawn keys, and recorded checksums. Verify that the theoretical SD column equals $2^d \sqrt{4d/(45N)}$ for each row. A visually similar plot is not enough if the data source or averaging rule changed.

B.4 Production performance sequence

The completed two-node campaign followed Stage 03 calibration with seven-repeat exploratory timing, then used the approved 60-million-sample workload for Stage 04 production scaling. Its weak-scaling unit was 15 million samples per computing worker. Stage 05 ran four controlled synthetic profiles through real MPI. Exact Windows command files, launcher details, logs, frozen summaries, and figures are retained under the timestamped campaign directory [P6, P9]. On a different machine, the correct reproduction sequence begins with a new serial calibration rather than assuming that 60 million samples will again run for 20 to 60 seconds.

The repository's Stage 07 preflight is `python generate_stage07_artifacts.py --require-complete`. The stored current status reports that the canonical datasets passed automated checks and that final submission blockers are empty [P6]. That process check supports data integrity; the interpretations in Sections 11-15 additionally disclose variability, stopwatch differences, rank allocation, and system limitations.

Appendix C Data definitions and analysis checks

C.1 Core variables

Symbol or field	Meaning	Unit or constraint
<code>d</code>	Dimension of the integration domain.	Positive integer; principal study uses 10.
<code>N</code> or <code>points</code>	Requested total Monte Carlo samples.	Positive integer.
<code>P</code> or <code>total_ranks</code>	MPI processes in the allocation.	Static: $P = W$; dynamic: $P = W + 1$.
<code>W</code> or <code>workers</code>	Ranks that generate and evaluate samples.	Positive integer.
<code>C</code> or <code>chunk_size</code>	Dynamic logical task size.	Positive integer; final task may be shorter.
<code>K</code>	Number of dynamic tasks.	$\text{ceil}(N/C)$.
<code>T_1</code>	Matching true-serial compute-phase runtime.	Seconds; paired by repetition for Stage 04 speedup.
<code>T_W</code>	Runtime for w computing workers.	Seconds; paired by repetition for Stage 04 speedup.
<code>S_W</code>	Strong-scaling speedup T_1/T_W .	Dimensionless; report the median paired ratio.
<code>E_W</code>	Worker efficiency S_W/W .	Dimensionless; median paired result divided by workers.
<code>E_alloc</code>	Dynamic allocation efficiency $S_W/(W+1)$.	Dimensionless; report alongside <code>E_W</code> .
<code>s</code>	Sample SD of repeated integral estimates.	Integral units.
<code>sigma_N</code>	Analytical SD at sample count N .	Integral units.

These definitions prevent the most common axis and denominator errors. A speedup must always identify its baseline and timer boundary. A worker-efficiency value above one can occur from measurement variation or cache effects; it should not immediately be interpreted as a new parallel law. An allocation-efficiency value for dynamic mode includes the master resource and answers a different question from worker efficiency.

C.2 Stage 06 columns

The raw statistical records include run identity, timestamp, Git commit, dimensions, points, repetition, root entropy, spawn key, RNG strategy, estimate, true value, signed error, absolute error, and relative error. The summary adds the number of repetitions, mean estimate, empirical bias, sample variance, sample SD, RMSE, theoretical variance and SD, SD ratio, standard error of the mean, and Student-t interval limits [P4]. The detailed schema allows every number in Tables 1 and 2 to be recomputed from the raw estimates.

The RNG records include strategy, base seed, two stream IDs, prefix length, first displayed values, repeat equality, and stream distinction. The summary records the tested stream count, prefix length, boolean verdict fields, and interpretation. A failed separation check is expected for `same_seed`; it should not be silently filtered as a test failure when summarizing the designed control [P5].

C.3 Production performance checks

Before accepting a Stage 04 plot, verify that each displayed point corresponds to the intended `run_class = production`, the same code revision or a documented equivalent, the calibrated workload, seven valid repetitions, one thread per rank, a recorded allocation, and the correct worker count. Check that `samples_processed = points`. Confirm that dynamic total ranks equal workers plus one. Recompute medians, quartiles, speedup, and efficiency from raw observations. Inspect failure logs and anomalous runs rather than excluding them without a reason.

For a static-versus-dynamic comparison, decide whether the primary question is equal computing capacity or equal total allocation and state that choice on the plot. For the chunk study, show `K` as well as `C`, because a one-task configuration cannot engage several workers. For a utilization plot, include every worker, including those with zero tasks. For communication claims, use measured MPI events or explicitly label a residual as a residual. These checks are the bridge between a technically functioning benchmark script and a defensible scientific performance result.

Appendix D Evidence map for final submission

Report claim or section	Supporting artifact in this snapshot	Evidence class
Integral, exact value, and variance derivation	<code>prng_math.py</code> ; statistical methodology; independent derivation in Section 2	Mathematical and implementation
Serial and MPI architecture	Entry points, <code>worker_compute.py</code> , <code>master_node.py</code> , tests	Implementation
Timing and resource definitions	Performance methodology, <code>performance_runtime.py</code> , result schema	Methodology and implementation
Stage 06 SD, RMSE, and confidence intervals	Raw and summary statistical CSVs, approved figures	Completed measured evidence
Stage 06 stream-prefix results	Raw and summary RNG CSVs, approved figure	Completed finite test evidence
Workload calibration and exploratory crossover	Stage 03 raw, summary, calibration, and figures	Frozen seven-repeat evidence
Strong/weak scaling, efficiency, chunk tuning	Stage 04 raw/worker CSVs, summaries, and figures	Frozen production evidence
Controlled scheduler stress	Stage 05 task events, worker metrics, and summaries	Frozen real-MPI synthetic evidence
Two-host provenance	Campaign report, rank proof, collective proof, source SHA	Frozen environment evidence

Stage 03-05 and Stage 06 use different source commits; their evidence remains separate [P6, P9].