IOPS

Reproducible benchmarking, from one YAML to a report

Luan Teylo & Abdraman Mahamat

COMPAS 2026 · IOPS v3.5.8

Integrated Orchestration for Parametric Studies

Part 1

Meet IOPS

what it is · the YAML · search · run · report

What is IOPS?

Integrated **O**rchestration for **P**arametric **S**tudies — an open-source tool that turns **one YAML file** into a full benchmarking campaign.

You describe

- the variables to sweep
- the command / job script
- a small parser for the metric

IOPS does

- run every configuration (local or SLURM)
- cache results, never repeat a run
- collect a CSV and an interactive report

💡 If it runs from a script and prints a number, IOPS can sweep it, cache it, and plot it.

```
pip install iops-benchmark
```

The problem: one run is never enough

You want the *best* performance from a benchmark. So you ask:

- Which **problem size** and **block size** run fastest?
- How many **threads** or **ranks** should I use?
- Does the winner at **1 node** still win at **4**?
- Is the gain **stable**, or just one lucky run?

The problem: one run is never enough

You want the *best* performance from a benchmark. So you ask:

- Which **problem size** and **block size** run fastest?
- How many **threads** or **ranks** should I use?
- Does the winner at **1 node** still win at **4**?
- Is the gain **stable**, or just one lucky run?

By hand this becomes a pile of scripts:

- one script per configuration
- copy/paste loops, magic suffixes
- `grep`/`awk` to scrape numbers
- a spreadsheet glued together by hand

⚠ Hard to reproduce, hard to share, easy to forget *which* script produced *which* number.

The smallest useful config

```yaml
benchmark:
  name: "IOR sweep"
  workdir: "./workdir"
  executor: "local"
  repetitions: 3

vars:
  block_size:               # 3 values
    type: int
    sweep: { mode: list, values: [256, 512, 1024] }
  transfer_size:            # 2 values
    type: int
    sweep: { mode: list, values: [1, 4] }

command:
  template: "ior -w -b {{ block_size }}m -t {{ transfer_size }}m"

scripts:
  - name: ior
    script_template: |
      #!/bin/bash
      mpirun -np 4 {{ command.template }}
    parser:
      parser_script: |
        def parse(p):
            ...

output:
  sink: { type: csv }       # csv, parquet, or sqlite
```

Five sections:

- `benchmark`: where & how
- `vars`: the space
- `command`: the run
- `scripts`: job + parser
- `output`: the sink

ℹ By default IOPS runs **every combination**:

`3 x 2 = 6` configs × 3 reps = **18 runs**, a script generated for each.

vars: how a variable gets its value

```
vars:
  # 1. SWEPT - the axes of the study
  nodes:
    type: int
    sweep: { mode: list, values: [1, 2, 4, 8] }
  ppn:
    type: int
    sweep: { mode: range, start: 8, stop: 32,
     step: 8 }

  # 2. DERIVED - computed from others (Jinja2)
  ntasks:
    type: int
    expr: "{{ nodes * ppn }}"

  # 3. CONSTANT - a literal expr, fixed for all
     runs
  block_size:
    type: int
    expr: "1024"
```

- **sweep** defines the axes (`list` or `range`)
- **expr** derives a value with full Jinja2
- a literal **expr** = constant

The matrix is the *cross-product* of the swept axes:

`4 nodes x 3 ppn = 12` points (x repetitions)

You don’t always want the full cross-product

Method	Coverage	Best for	Deterministic?
`exhaustive`	every combination	small spaces, full picture	yes
`random`	sampled subset	large spaces, statistics	seed
`bayesian`	guided to optimum	expensive runs, find best	no

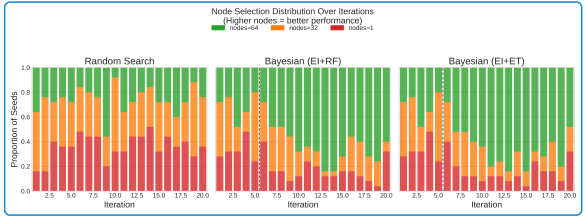
🎚 Switch strategy with *one line* (`search_method`): the rest of the config is untouched.

Bayesian: find the best setting cheaply

```
benchmark:
  search_method: "bayesian"
  bayesian_config:
    objective_metric: "gflops"
    objective: "maximize"
    n_initial_points: 6
    n_iterations: 24
```

IOPS builds a surrogate model of the `objective_metric` and **spends runs where they pay off**, instead of testing everything.

◎ ~90% of the optimum after only ~7% of the space. Needs `pip install scikit-optimize`.



Bayesian (centre/right) quickly concentrates on the fast region (green); random (left) keeps sampling everywhere.

Metrics: the numbers IOPS measures

A **metric** is a named measurement. You **declare** it in YAML and a `parse()` **produces** it from the output:

```yaml
parser:
  file: "{{ execution_dir }}/result.json"
  metrics:                          # declare what you measure
    - { name: gflops, type: float }
  parser_script: |
    import json
    def parse(path):
        d = json.load(open(path))
        # an invalid (broken / cheated) run scores zero
        return {"gflops": d["value"] if d["valid"] else 0.0}
```

Metrics are IOPS's currency: they become `results.csv` columns and report plots, and drive the Bayesian `objective_metric`. The keys returned by `parse()` must match the declared names.

output: where the results go

One `output.sink` gathers every parsed metric into a single dataset:

```
output:
  sink:
    type: csv                  # csv, parquet, or sqlite
    path: "{{ workdir }}/results.csv"
```

- one row per (config, repetition)
- all variables *and* metrics as columns
- results are always **appended**

Formats: `csv`, `parquet`, `sqlite`.
Ready to plot or diff.

Run: one command, a tidy workdir

`iops run cfg.yaml` builds the tree below. IOPS hands each folder to your templates as a **built-in variable**:

```
run_001/                  # {{ workdir }}
  logs/                   # {{ log_dir }}
  runs/
    exec_0001/            # one config: {{ execution_id
      }}
      repetition_001/     # {{ execution_dir }}
        run.sh            # the script that ran
        stdout  stderr    # captured output
        result.json       # this run's output
  results.csv             # parsed metrics (sink)
  report.html             # interactive report
```

```
{{ workdir }}         # run root run_NNN/
{{ execution_dir }}   # this rep's folder
{{ log_dir }}         # the logs/ folder
{{ execution_id }}    # config id
{{ repetition }}      # repetition number
```

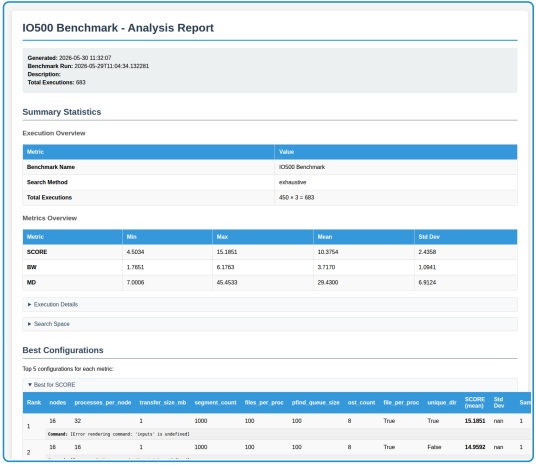
🗁 **Reserved** names. The starters set `result_file` under `execution_dir`, so each run's output stays in its own folder.

From CSV to an interactive report

```
iops report run_001                    # one self-contained HTML file
```

The report bundles:

- summary **statistics** per metric
- the **best configurations** found
- the **search space** explored
- interactive **plots**



IO500 Benchmark - Analysis Report

Generated: 2026-05-30 11:32:07
Benchmark Run: 2026-05-29T11:04:34.132281
Description:
Total Executions: 683

Summary Statistics

Execution Overview

Metric	Value
Benchmark Name	IO500 Benchmark
Search Method	exhaustive
Total Executions	450 × 3 = 683

Metrics Overview

Metric	Min	Max	Mean	Std Dev
SCORE	4.5034	15.1851	10.3754	2.4358
BW	1.7651	6.1763	3.7170	1.0941
MD	7.0006	45.4533	29.4300	6.9124

▸ Execution Details

▸ Search Space

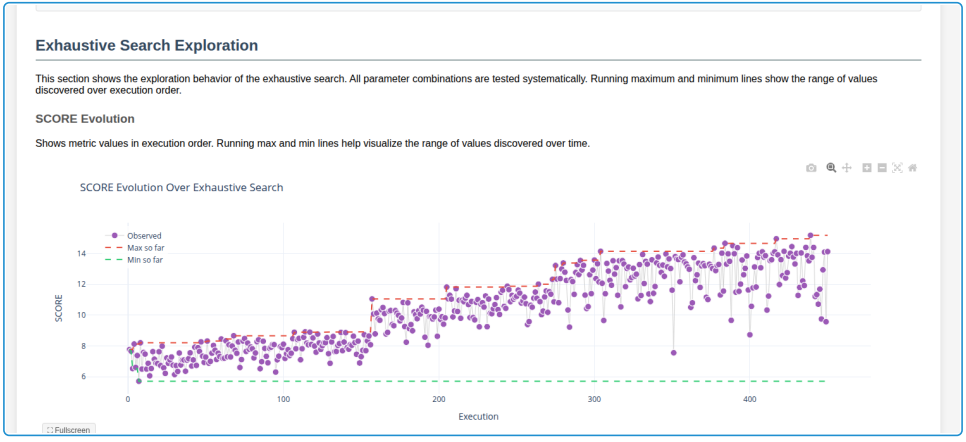
Best Configurations

Top 5 configurations for each metric:

▾ Best for SCORE

Rank	nodes	processes_per_node	transfer_size_mb	segment_count	files_per_proc	pfind_queue_size	ost_count	file_per_proc	unique_dir	SCORE (mean)	Std Dev	Sam
1	16	32	1	1000	100	100	8	True	True	**15.1851**	nan	1
	Command: [Error rendering command: 'inputs' is undefined]											
2	16	16	1	1000	100	100	8	True	False	**14.9592**	nan	1

Report: explore the search



Interactive charts: scatter, bars, heatmaps, parallel coordinates, and search-evolution views.

Caching: never compute the same point twice

```
benchmark:
  cache_file: "./cache.db"
  cache_exclude_vars: ["result_file"]  # path-
    like
```

```
iops run cfg.yaml --use-cache   # skip
    cached
iops cache stats  cache.db
iops cache list   cache.db nodes=8
```

- key = **parameter values** + repetition
- only SUCCEEDED runs cached
- interrupted? rerun and it **resumes**

⚠ The key is the *parameters*, not the command. Change what a flag *does* ⇒ clear the cache.

Archiving a run

Bundle a whole run into one portable, checksummed archive to share or submit:

```
iops archive create ./run_001 -o study.tar.gz   # pack a run
iops archive extract study.tar.gz -o ./out      # unpack it
iops find study.tar.gz                           # inspect, no extract
```

📦 The archive carries a **manifest** and **checksums**, so a study stays self-contained and verifiable when moved between machines — exactly what the submission site checks.

Part 2

The competition

benchmarks · scoring · submission · your workflow

First: install IOPS and sign up

1. Install IOPS (Python 3.10+)

```
pip install iops-benchmark
```

2. Create your account on the submission site

tutorial.iops-benchmark.com

What you will do now

You are handed a suite of benchmarks. For each one, **find the parameter combination that gets the most performance** and submit it.

Two things are scored

- the **metric** you reach (GFLOP/s, GB/s, …)
- the **core-hours** you spent finding it

🎛 The loop you repeat:

edit `vars` → `iops run` → `iops report` → `iops archive` → upload.

🏆 Best metric *and* fewest core-hours wins. Searching smart beats brute force.

Installing the benchmark suite

IOPS you already have. Now get the **benchmark binaries**:

```
unzip benchmarks.zip && cd benchmarks
./install.sh                          # build + install into ~/.local/bin
export PATH="$HOME/.local/bin:$PATH"

allreduce --help                      # every benchmark prints its own
    flags
```

ℹ The YAMLs call each benchmark **by name**, so the binaries must be on your `PATH`. On the cluster they may already be in a shared module — ask the instructor.

The benchmark suite

Single-node (OpenMP) — one process, `processes_per_node` = threads, `nodes`=1.

Benchmark	Stresses	Metric	Headline knobs
`gemm`	dense FP / vectorization	`gflops`	size, tile, threads
`triad`	memory bandwidth	`gbps`	size, stride, non-temporal, threads
`stencil`	cache blocking	`mlups`	size, block bx×by, threads
`gups`	random access + atomics	`gups`	table size, updates, method, threads

Node-scaling (MPI) — pure MPI, one thread per rank, `cores = nodes × processes_per_node`.

Benchmark	Stresses	Metric	Headline knobs
`halo`	MPI neighbour exchange	`gbps`	subdomain, iters, comm mode, ranks
`iobench`	parallel I/O (IOR-like)	`bw`	block, transfer, FPP/shared, API
`alltoall`	MPI all-to-all	`gbps`	message size, iters, ranks
`allreduce`	MPI collective reduction	`gbps`	vector length, iters, op, ranks
`nbody`	distributed compute + ring	`ginteractions`	particles, steps, ranks

All metrics are **maximize**; each benchmark self-verifies, so an invalid run scores **zero**.

How you are scored

Per benchmark, your **best valid result** is scored against the **best result reached so far** (`best_so_far`); your **core-hours** are penalized:

```
metric_score = your_best / best_so_far      # 1.0 if you lead
search_score = 1 / (1 + total_core_hours)   # fewer hours, nearer 1
rank         = 0.7 * metric_score + 0.3 * search_score
overall      = mean(rank over the benchmarks you entered)
```

- **70%** your performance vs. the current best
- **30%** how few core-hours it took
- an `invalid` run counts as metric `0`
- enter **more benchmarks** to raise your mean

💡 `best_so_far` moves as people improve, so scores update live. A huge sweep can *lower* yours — every run adds core-hours.

Archive and submit

Bundle the run into one checksummed archive, then upload it:

```
iops archive create run_001/ -o allreduce.tar.gz   # pack the run

# upload from the web UI (one card per benchmark), or by curl:
curl -F file=@allreduce.tar.gz -F token=$IOPS_TOKEN \
     http://<host>:8000/api/submissions/allreduce
```

- submit the **archive**, not the CSV
- the benchmark in the URL must match `benchmark.name`
- resubmit anytime — your **best accepted** entry counts

The site re-verifies the manifest & checksums. An edited `results.csv` is rejected.

Your workflow, end to end

1. `iops check <bench>.yaml` — config valid?
2. `iops run <bench>.yaml --use-cache` — execute (resumable)
3. `iops report run_NNN` — inspect best & validity
4. `iops archive create run_NNN/ -o sub.tar.gz` — package
5. upload to the site — climb the leaderboard

🚀 Then tighten the `vars` around the best region and repeat. The cache keeps the old runs, so each iteration is cheap.

Reference

The benchmark suite

what each one measures, and the knobs you tune

Reference: single-node (OpenMP)

gemm — GFLOP/s

Dense $N \times N$ matrix multiply with cache blocking; stresses peak floating-point and vectorization.

- `-size` — matrix dimension N
- `-tile` — cache-block size (locality sweet spot)

triad — GB/s

STREAM triad `a = b + s*c` over large arrays; stresses sustained memory bandwidth.

- `-size` — elements per array (exceed cache for DRAM)
- `-stride` — touch every S-th element
- `-nt 0|1` — non-temporal (streaming) stores

Plus `processes_per_node` = OpenMP threads on both.

Reference: single-node (OpenMP)

stencil — MLUP/s

2D 5-point Jacobi sweeps over an $N \times N$ grid; cache blocking decides performance.

- `-size` — grid dimension N
- `-bx`, `-by` — block width / height

gups — GUP/s

Updates at pseudo-random indices over a 2^K-entry table; stresses random DRAM latency and atomics.

- `-logsize` — table = 2^K entries (cache vs DRAM)
- `-method 0|1|2` — racy (invalid, many threads) | atomic | private

Plus `processes_per_node` = OpenMP threads on both.

Reference: node-scaling (MPI)

halo — GB/s

Ranks on a 2D grid swap edges with their neighbours; stresses point-to-point network bandwidth.

- `--sub` — subdomain edge (sets message size)
- `--mode 0|1|2` — sendrecv | nonblocking | persistent

iobench — MiB/s

IOR-like parallel write; every rank writes its share, reports the aggregate bandwidth.

- `--block` MiB/rank, `--transfer` KiB/write
- `--mode 0|1` — file-per-proc | shared file
- `--api 0|1|2` — POSIX | MPI-IO indep | coll
- `--dir` — scratch filesystem

Plus `nodes` × `processes_per_node` = ranks (and topology) on both.

Reference: node-scaling (MPI)

alltoall — GB/s

Every rank exchanges with every other; stresses bisection bandwidth.

- `--count` — doubles to each peer

allreduce — GB/s

Collective sum/max of a vector across all ranks; stresses the reduction tree.

- `--count` — vector length
- `--op 0|1` — SUM | MAX

nbody — Ginter/s

All-pairs forces each step; ranks pass particles in a ring. Compute + neighbour comm.

- `-n` — particle count

ℹ All three also tune `nodes` × `processes_per_node` (ranks). Discover any benchmark's exact flags with `<bench> --help`.