

IOPS



Reproducible benchmarking, from one YAML to a report

Luan Teylo

June 2026 · IOPS v3.5.6

Integrated Orchestration for Parametric Studies

The problem: one run is never enough

You want to know how GYSELA writes to the PFS. So you ask:

- Which **stripe count** gives the best write bandwidth?
- What about **subfiling**?
- Does it hold at **64 nodes** as well as 128? With more tasks per node?
- Is the gain **stable**, or just one lucky run?

The problem: one run is never enough

You want to know how GYSELA writes to the PFS. So you ask:

- Which **stripe count** gives the best write bandwidth?
- What about **subfiling**?
- Does it hold at **64 nodes** as well as 128? With more tasks per node?
- Is the gain **stable**, or just one lucky run?

In practice this becomes a pile of hand-edited scripts:

- one sbatch 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 core idea

One YAML to rule them all.

You write

- the variables to sweep
- templates (run command / job script)
- a small Python parser

```
pip install iops-benchmark
```

IOPS gives you

- automatic execution (local or SLURM)
- a centralized CSV of results
- a nice interactive report

Python 3.10+, single tool, git-style subcommands

The pipeline: one config, five stages



The **cache** sits underneath all of this: a result, once computed, is never computed again, and can be *reused* when the study grows.

Part 1

Describe the experiment

benchmark · vars · command · scripts · output

The smallest useful config (IOR)

```
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 required 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.

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)

Conditional variables: sweep only when relevant

A variable can be swept only **when** another makes it meaningful, with a default otherwise:

```
vars:
  use_compression:
    type: bool
    sweep: { mode: list, values: [true, false] }
  compression_level:
    type: int
    sweep: { mode: list, values: [1, 5, 9] }
    when: "use_compression"    # only when true
    default: 0                 # otherwise fixed

command:                        # full Jinja2 in templates
  template: >
    mybench {% if use_compression %}
      --zlevel {{ compression_level }}{% endif %}
```

i Without when: $2 \times 3 = 6$ runs, half redundant.

With when: $3 + 1 = 4$ runs (the three levels, plus one “no compression”).

</> Templates get the **full Jinja2**: {% if %}, loops, filters, expressions.

Constraints: drop combinations that make no sense

Filter out invalid points before anything runs (here: IOR needs `block_size` to be a multiple of `transfer_size`):

```
constraints:  
  - name: block_alignment  
    rule: "block_size % transfer_size == 0"  
    violation_policy: skip      # skip | warn | error
```

i The rule is any boolean expression over your variables; violations are skipped (or warn/error).

You do not 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
adaptive	step to a limit	find a threshold / cliff	yes

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

Bayesian: find the best I/O setting cheaply

benchmark:

```
search_method: "bayesian"  
bayesian_config:  
  objective_metric: "bandwidth"  
  objective: "maximize"  
  n_initial_points: 5  
  n_iterations: 20
```

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.

More details: <https://iops.gitlabpages.inria.fr/blog/bayesian-optimization-study/>



Bayesian (centre/right) quickly concentrates on `nodes=64` (green = fast configs); random (left) keeps sampling everywhere.

exhaustive_vars: never miss a value

Search the expensive axes, but **fully sweep** the one knob you study:

```
benchmark:
  search_method: "bayesian"    # or "random"
  exhaustive_vars: ["stripe_count"]
vars:
  nodes:
    sweep: { values: [1, 2, 4, 8] }    # searched by BO
  ppn:
    sweep: { values: [16, 32] }        # searched by BO
  stripe_count:
    sweep: { values: [1, 2, 4, 8] }    # fully run at each point
```

BO picks	IOPS runs	Back to BO
nodes=4, ppn=16	stripe=1,2,4,8	best write_bw
nodes=8, ppn=32	stripe=1,2,4,8	best write_bw



BO picks a (nodes, ppn) point; IOPS runs *all* stripe_count values (kept for the report). BO is told only the **best** per point.

Metrics: the numbers IOPS measures

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

```
parser:
  file: "{{ execution_dir }}/stdout"
  metrics:                                # declare what you measure
    - { name: write_bw, type: float }
    - { name: read_bw, type: float }
  parser_script: |
    import re
    def parse(path):
      text = open(path).read()
      write = re.search(r"Max Write:\s+([\d.]+)", text).group(1)
      read = re.search(r"Max Read:\s+([\d.]+)", text).group(1)
      return {"write_bw": float(write), "read_bw": float(read)}
```

i Metrics are IOPS's currency: they become `results.csv` columns and report plots, and drive the Bayesian `objective_metric` and `adaptive stop_when`. 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.
You choose where it lives: ready to plot or diff.

Part 2

Run it and see the results

dry-run · run · watch · report

Preview the plan before running

```
iops run cfg.yaml --dry-run --time-estimate "60,120,300"
```

A dry run reports, without launching anything:

- how many **tests** and **executions**
- estimated **core-hours** and wall-clock
- a **scenario** table for different per-run times

```
=====
EXECUTION PLAN SUMMARY
=====

Execution Plan:
  Unique tests: 720
  Repetitions per test: 3
  Total executions: 2160

Core Configuration:
  Cores expression: {{ ntasks }}
  Min cores per execution: 16
  Max cores per execution: 576
  Avg cores per execution: 109.7

Estimated Core-Hours (for 2160 executions to run):
  Estimated time per execution: 60.0 seconds (1.0 minutes)
  Total core-hours: 3948.00

Estimated Wall-Clock Time (sequential):
  Total: 36.00 hours (1.50 days)

=====
SCENARIO ANALYSIS (3 time estimates)
=====

Time/Exec      Core-Hours      Wall-Clock      Budget %      Execs Fit
-----
60s (1.0m)     3948.00         36.00h          N/A           2160/2160
120s (2.0m)    7897.60         72.00h          N/A           2160/2160
300s (5.0m)    19744.00        180.00h         N/A           2160/2160

=====
GENERATING REPORT
=====
✓ Report saved to: /home/lgouveia/workdir_io500/dryrun_001/dry-run-report.txt

=====
DRY-RUN COMPLETE - Nothing was executed
```

Run: one command, a tidy workdir

```
iops run cfg.yaml           # execute the plan
```

```
run_001/  
  exec_0001/           # one folder per config  
    repetition_001/  
      run_ior.sh        # the script that ran  
      stdout stderr    # captured output  
      ior.out           # benchmark output  
      results.csv       # parsed metrics  
      report.html       # interactive report
```

 Fully **reproducible**: the script that produced each number sits right next to its stdout, stderr, and output files.

Watch it run, live

```
iops find run_001 --watch
```

```
# live table + progress bar
```

IO500 Benchmark														
720 tests = 3 * test = 2160 total														
run_001 04:00:15 11:17:17														
[LIVE] progress output														
test	block_size_mb	file_per_proc	files_per_proc	nodes	ntasks	ost_count	pfind_queue_size	processes_per_node	segment_count	total_size_gb	transfer_size_mb	unique_dir	Reps	Status
exec_0001	1920	True	100	2	16	2	100	0	1000	30	1	True	S H N WAIT	25.0s
exec_0002	1920	True	100	2	16	2	100	0	1000	30	1	False	S H N WAIT	24.0s
exec_0003	1920	False	100	2	16	2	100	0	1000	30	1	True	S H N WAIT	49.0s
exec_0006	1920	True	100	2	16	4	100	0	1000	30	1	False	S H N WAIT	25.0s
exec_0008	1920	False	100	2	16	4	100	0	1000	30	1	False	S H N WAIT	35.0s
exec_0009	1920	True	100	2	16	7	100	0	1000	30	1	True	S H N WAIT	24.0s
exec_0010	1920	True	100	2	16	7	100	0	1000	30	1	False	S S N WAIT	42.5s
exec_0014	1920	True	100	2	16	8	100	0	1000	30	1	False	S H N WAIT	23.0s
exec_0016	1920	False	100	2	16	8	100	0	1000	30	1	False	S S N WAIT	31.0s
exec_0017	1920	True	100	2	16	2	100	0	1000	30	8	True	S S N WAIT	26.5s
exec_0019	1920	False	100	2	16	2	100	0	1000	30	8	True	S H N WAIT	49.0s
exec_0020	1920	False	100	2	16	2	100	0	1000	30	8	False	S H N WAIT	36.0s
exec_0023	1920	False	100	2	16	4	100	0	1000	30	8	True	S H N WAIT	37.0s
exec_0025	1920	True	100	2	16	7	100	0	1000	30	8	True	S H N WAIT	23.0s
exec_0027	1920	False	100	2	16	7	100	0	1000	30	8	True	S H N WAIT	39.0s
exec_0034	1920	True	100	2	16	2	100	0	1000	30	16	False	S S N WAIT	24.0s
exec_0038	1920	True	100	2	16	4	100	0	1000	30	16	False	S H N WAIT	22.0s
exec_0039	1920	False	100	2	16	4	100	0	1000	30	16	True	S H N WAIT	35.0s
exec_0040	1920	False	100	2	16	4	100	0	1000	30	16	False	S S N WAIT	30.0s
exec_0042	1920	True	100	2	16	7	100	0	1000	30	16	False	S H N WAIT	23.0s
exec_0043	1920	False	100	2	16	7	100	0	1000	30	16	True	S H N WAIT	30.0s
exec_0045	1920	True	100	2	16	8	100	0	1000	30	16	True	S H N WAIT	26.0s
exec_0046	1920	True	100	2	16	8	100	0	1000	30	16	False	S H N WAIT	25.0s
exec_0048	1920	False	100	2	16	8	100	0	1000	30	16	False	S H N WAIT	20.0s
exec_0049	1920	True	100	2	16	2	100	0	1000	30	32	True	S H N WAIT	20.0s
exec_0051	1920	False	100	2	16	2	100	0	1000	30	32	True	S H N WAIT	49.0s
exec_0052	1920	False	100	2	16	2	100	0	1000	30	32	False	S H N WAIT	32.0s
exec_0054	1920	True	100	2	16	4	100	0	1000	30	32	False	S H N WAIT	24.0s
exec_0055	1920	False	100	2	16	4	100	0	1000	30	32	True	S S N WAIT	35.0s
exec_0056	1920	False	100	2	16	4	100	0	1000	30	32	False	S H N WAIT	34.0s
exec_0058	1920	True	100	2	16	7	100	0	1000	30	32	False	S H N WAIT	25.0s
exec_0059	1920	False	100	2	16	7	100	0	1000	30	32	True	S S N WAIT	20.5s
exec_0060	1920	False	100	2	16	7	100	0	1000	30	32	False	S S N WAIT	20.0s
exec_0061	1920	True	100	2	16	8	100	0	1000	30	32	True	S H N WAIT	24.0s
exec_0062	1920	True	100	2	16	8	100	0	1000	30	32	False	S H N WAIT	25.0s
exec_0064	1920	False	100	2	16	8	100	0	1000	30	32	False	S H N WAIT	27.0s
exec_0065	1920	True	100	2	16	2	100	0	1000	30	64	True	S H N WAIT	27.0s
exec_0068	1920	False	100	2	16	2	100	0	1000	30	64	False	S S N WAIT	43.0s
exec_0070	1920	True	100	2	16	4	100	0	1000	30	64	False	S H N WAIT	23.0s
exec_0071	1920	False	100	2	16	4	100	0	1000	30	64	True	S H N WAIT	33.0s
exec_0073	1920	True	100	2	16	7	100	0	1000	30	64	True	S H N WAIT	24.0s
exec_0075	1920	False	100	2	16	7	100	0	1000	30	64	True	S S N WAIT	19.0s
exec_0081	960	True	100	2	32	2	100	16	1000	30	1	True	S H N WAIT	20.0s
exec_0084	960	False	100	2	32	2	100	16	1000	30	1	False	S H N WAIT	40.0s
exec_0087	960	False	100	2	32	4	100	16	1000	30	1	True	S S N WAIT	40.5s
(463 completed hidden, 238 pending off-screen)														
31.4%														
avg: 38.9s/test -16.0h left														
executed: 483 Pending: 1477														
Core-hours: 1134.64														

Per-execution status and metrics update in place, with a progress bar and core-hours used.

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: 2020-06-30 11:32:07
Benchmark Run: 2020-06-29T11:04:34.132Z
Description:
Total Executions: 663

Summary Statistics

Execution Overview

Metric	Value
Benchmark Name	IO500 Benchmark
Search Method	exhaustive
Total Executions	450 × 3 = 663

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

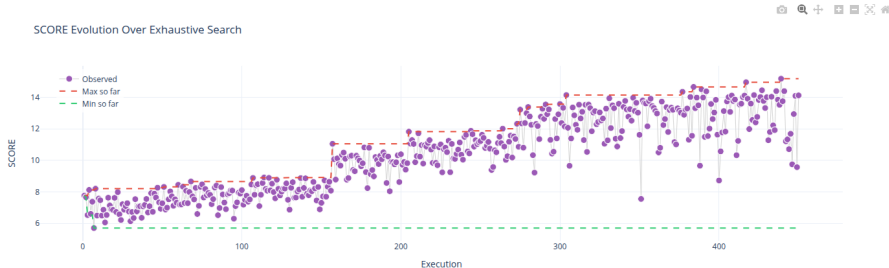
Report: explore the search

Exhaustive Search Exploration

This section shows the exploration behavior of the exhaustive search. All parameter combinations are tested systematically. Running maximum and minimum lines show the range of values discovered over execution order.

SCORE Evolution

Shows metric values in execution order. Running max and min lines help visualize the range of values discovered over time.



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

Make the report part of the config

Add a `reporting` section and the report is generated automatically after every run, with exactly the plots you care about:

```
reporting:
  enabled: true
  metrics:
    write_bw:
      plots:
        - { type: bar,      x: stripe_count }      # write_bw vs stripe count
        - { type: heatmap, x: nodes, y: ppn }
```

 A single bar chart of `write_bw` vs `stripe_count` answers “which stripe count wins?” at a glance, refreshed every time the study runs.

Part 3

Going further

probes · caching

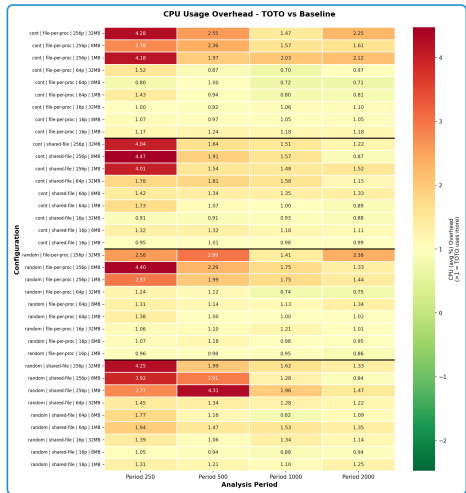
Resource probes: CPU, memory, GPU

Turn on lightweight samplers to record the **resource footprint** of each run:

```
benchmark:  
  probes:  
    resource_sampling: true    # CPU +  
                               memory  
    gpu_sampling: true        # GPU (  
                               nvidia-smi)  
    sampling_interval: 1.0    # seconds
```

- low-priority sampler **per node**
- per-core CPU %, memory, GPU use
- folded into the report as heatmaps

🔧 Sampling adds a little overhead: take a **baseline** first, then enable probes.



CPU-usage overhead per configuration (sampling
vs. baseline). 19/22

Caching: never compute the same point twice

```
benchmark:  
  cache_file: "./workdir/iops_cache.db"  
  cache_exclude_vars: ["output_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
- retry failures safely

 The key is the *parameters*, not the command. Change the command's behaviour $\Rightarrow$ clear the cache or bump a `cache_version` var.

Not only I/O

IOPS **started as an I/O benchmarking tool** (IOR, MDTEST, IO500), then we made the engine **generic**. It is a working, fully documented tool with **growing adoption** beyond its original scope:

CEA · NUMPEX

Parametric performance studies on production HPC systems.

IO500 collaboration

INRIA, JGU, Sandia, and the IO500 community: smart parameter-space search to optimize IO500 scores.

INRIA (Concace) · Airbus

Under evaluation for industrial HPC performance studies.

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

Get involved

IOPS is an open-source project created at **TADaam** (Inria).

🌐 iops.gitlabpages.inria.fr | 🐱 gitlab.inria.fr/lgouveia/iops | 📦 `pip install iops-benchmark`

Want to use, contribute, or share ideas? Email luan.teylo@inria.fr or open an issue.

Contributors



Luan Teylo

Inria



Francieli Boito

Inria



Abdraman Mahamat

Inria



Mihail Popov

Inria



Méline Trochon

Inria/DDN



Jérôme Charousset

CEA



Alexandre Lou-Poueyou

Inria



Mohamad Almousa

Inria

Extra

parallel execution · adaptive · growing the search space · multi-machine · SLURM · CI/CD · archive

Parallel execution

By default tests run one at a time. One setting changes that:

```
benchmark:  
  parallel: 4          # up to 4 tests concurrently
```

iops run cfg.yaml --parallel 4 (CLI overrides YAML)

Method	Max parallelism	Why
exhaustive / random	unlimited	tests are independent
bayesian	1	each result feeds the next suggestion
adaptive	#probes	probes independent, steps sequential

Works with *both* backends: parallel subprocesses locally, or many SLURM jobs submitted and polled at once. Budget tracking stays accurate.

Growing the parameter space without re-running

You ran a full sweep. Now you want to add a *new* axis, compression, but keep the old results for compression=off.

```
# Add a constant variable to every existing cache entry
```

```
iops cache rebuild cache.db --add compression:bool=false -o new_cache.db
```

```
vars:
```

```
  nodes:      { type: int,  sweep: { mode: list, values: [1, 2, 4, 8] } }
```

```
  compression: { type: bool, sweep: { mode: list, values: [false, true] } }
```

🔧 Now `iops run -use-cache` finds hits for all `compression=false` points and only runs the *new* `compression=true` half. The old runs are reused, not repeated.

Catch regressions when commits land



1. On each new commit, the CI job runs the *same* config with the small ci profile.
2. Results **append** to a central `history.csv`, building a record over time.
3. Compare against the baseline; **fail the pipeline** if the metric regresses beyond a threshold.

A CI machine profile + pipeline

A fast ci profile that writes to a **central CSV**:

```
machines:
  ci:
    benchmark:
      repetitions: 1
    vars:
      nodes:
        sweep: { values: [1, 4, 8] }
    output:
      sink:
        type: csv
        path: "/shared/iops/history.csv"
```

```
# .gitlab-ci.yml (sketch)
io-benchmark:
  rules:
    - if: $CI_PIPELINE_SOURCE ==
      "merge_request_event"
  script:
    - pip install iops-benchmark
    - iops run cfg.yaml --machine
      ci
    - python compare.py
      /shared/iops/history.csv
```

👉 You decide where results live: every commit **appends** to one `history.csv`, so performance over time sits in a single file you can diff. (IOPS_MACHINE=ci works too.)

The SLURM backend (and beyond)

```
benchmark:
  executor: "slurm"
  cores_expr: "{{ nodes * ppn }}"
  max_core_hours: 1000          # budget guard
  slurm_options:
    commands:                   # override the
      defaults
      submit: "sbatch"
      status: "squeue -j {job_id} --format=%T"
      info: "scontrol show job {job_id}"
      cancel: "scancel {job_id}"
  poll_interval: 30             # seconds
```

IOPS handles the lifecycle:

- submit, poll, status
- per-job status tracking
- core-hour budget

≡ Swap these for *any* scheduler or wrapper (the {job_id} placeholder is filled at runtime). Not tied to SLURM.

Archiving a run

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

```
iops archive create ./workdir/run_001 -o study.tar.gz # pack a run
iops archive extract study.tar.gz -o ./extracted      # 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.

Multi-machine support

The **same experiment** on different clusters, in **one file**: override only the site-specific setup.

```
# Base: the experiment,  
# shared by every machine  
benchmark:  
  workdir: "./workdir"  
  repetitions: 5  
vars:  
  nodes:  
    sweep: { values: [1,2,4,8] }  
  ppn:  
    sweep: { values: [16,32] }  
command:  
  template: "ior -w -b 1g -t 2m"
```

...

same

file

```
machines:                # only the site-specific setup  
  differs  
  plafrim:  
    benchmark: { workdir: "/scratch/$USER/b" }  
    scripts:  
      - name: run  
        script_template: |  
          #SBATCH --partition=bora  
          module load openmpi  
          srun {{ command.template }}  
  grid5000:                # OAR, not SLURM  
    benchmark: { workdir: "/home/$USER/b" }  
    scripts:  
      - name: run  
        script_template: |  
          #OAR -l nodes={{ nodes }},walltime=1:00:00  
          module load openmpi  
          mpirun {{ command.template }}
```

```
ioops run config.yaml --machine plafrim
```

```
ioops run config.yaml --machine grid5000
```

```
# run on PlaFRIM
```

```
# run on Grid'5000
```

Adaptive: probe for a limit

When the question is “*how far can we push it?*” rather than “*which is best?*”:

```
benchmark:
  search_method: "adaptive"
vars:
  nodes:                                # swept as usual
    type: int
    sweep: { mode: list, values: [1, 2, 4] }
  problem_size:
    type: int
  adaptive:
    initial: 1000
    factor: 2                            # double each step
    stop_when: "exit_code != 0"         # until it breaks
    max_iterations: 15                  # safety net
```

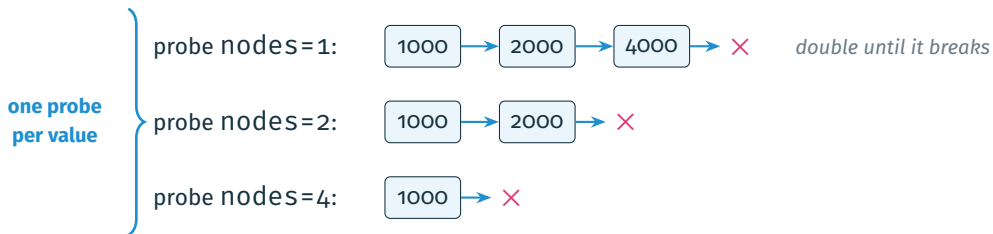
What stop_when can find:

- **max problem size** before it crashes
"exit_code != 0"
- a **performance cliff**
"gflops < 50"
- a **time budget** limit
"execution_time > 300"

↔ Steps (double, add, or a custom expression) until the condition is met. One independent probe per swept combination.

Swept vars + adaptive: one probe per combination

With swept variables *and* an adaptive one, IOPS runs **one independent probe** per swept combination. Example: nodes: [1, 2, 4] + adaptive problem_size.



i Each probe steps its adaptive value independently until its stop_when fires, finding the limit *for that combination*.