Parallel simulations

Simulation studies — parameter recovery, scenario sweeps, prior sensitivity — are loops of *independent* fits. Written as a plain `for` loop they run one at a time, so a single core works while the rest of the machine sits idle.

`run_simulations` spreads that loop across worker processes. Each worker is a separate Python process with its own JAX backend, so every simulation gets its own `bf()` instance, its own share of the cores and its own seed.

Crucially, **the scenario settings travel with the results**: every row of the returned table carries the parameters that produced it, so comparing true against estimated values needs no bookkeeping of your own.

---

1. The pattern

Write a function that runs **one** simulation and returns the numbers you care about. `run_simulations` calls it once per scenario, in parallel, and stacks the results into a `pandas` DataFrame.

```
from BayesForge import bf, grid, run_simulations

def one_sim(<your scenario parameters>, seed, **_):
    m = bf(rand_seed=seed, print_devices_found=False)  # one bf per simulation

    data = <simulate a dataset using the scenario parameters>
    m.fit(<model>, obs=data, progress_bar=False)

    return {"estimate": ..., "truth": ...}   # -> one row of the result table

scenarios = grid(<param>=[...], <param>=[...], reps=<n>)   # or just an int
res = run_simulations(one_sim, scenarios)                  # -> DataFrame
```

Three things the runner injects into every call, on top of your own parameters:

injected	meaning
`sim`	global index of the simulation
`rep`	replicate index within its scenario cell
`seed`	per-simulation seed, derived deterministically from the base seed

Pass `seed` into `bf(rand_seed=seed)`. Without it every worker draws the *same* data, because `bf()` seeds the global NumPy and Python RNGs from that value.

Always give `one_sim` a `**_` catch-all so extra injected keys cannot break it.

! In a `.py` script, use a main guard

Workers are *spawned*, which re-imports the calling script. Without the guard the module level runs again in every worker:

```
if __name__ == "__main__":
    res = run_simulations(one_sim, scenarios)
```

Notebooks and the REPL need no guard.

⚠ Return numbers, not the model

A fitted `bf` object cannot cross a process boundary — `m.diag` holds closures over `m`, and `m.dist` is an unpicklable proxy. Return `m.summary()`, `m.sampler_stats()`, or a plain dict. Returning `m` raises an explicit error.

---

2. A concrete example

Recovering the slope of a linear regression across two sample sizes and three true slopes, four replicates each — 24 simulations.

```
import numpy as np
from BayesForge import bf, grid, run_simulations

def one_sim(N, b_true, seed, **_):
```

```
    m = bf(rand_seed=seed, print_devices_found=False)

    # --- simulate one dataset ------------------------------------
    x = np.asarray(m.dist.normal(0.0, 1.0, sample=True, shape=(N,)))
    y = np.asarray(m.dist.normal(1.5 + b_true * x, 1.0, sample=True))

    # --- the model -----------------------------------------------
    def linreg(x=None, y=None):
        a = m.dist.normal(0.0, 5.0, name="a")
        b = m.dist.normal(0.0, 5.0, name="b")
        s = m.dist.exponential(1.0, name="s")
        m.dist.normal(a + b * x, s, name="y", obs=y)

    # --- fit, then report truth against estimate -----------------
    m.fit(linreg, obs=dict(x=x, y=y),
          num_warmup=500, num_samples=500, num_chains=1,
          progress_bar=False)

    est = m.summary()
    return {"b_hat":  float(est.loc["b", "mean"]),
            "b_true": b_true,
            "r_hat":  float(est.loc["b", "r_hat"])}

if __name__ == "__main__":
    scenarios = grid(N=[100, 500], b_true=[-0.8, 0.0, 0.8], reps=4)   # 24 sims
    res = run_simulations(one_sim, scenarios)

    print(res[["sim", "N", "b_true", "b_hat", "elapsed_s"]].head())
```

`res` is a tidy DataFrame — the grid columns `N` and `b_true` are joined onto every row automatically:

```
   sim    N  b_true  b_hat  elapsed_s
0    0  100    -0.8  -0.64      8.952
1    1  100    -0.8  -0.80      9.099
2    2  100    -0.8  -0.93      8.598
3    3  100    -0.8  -0.62      8.620
4    4  100     0.0  -0.13      8.184
```

So the analysis is a `groupby`, with nothing to join by hand:

```
(res.b_hat - res.b_true).abs().groupby([res.N, res.b_true]).mean()
```

```
N    b_true
100  -0.8      0.118
      0.0      0.073
      0.8      0.055
500  -0.8      0.015
      0.0      0.025
      0.8      0.018
```

Alongside your own columns, every row carries `sim`, `rep`, `seed`, `worker`, `elapsed_s` and `error`.

---

3. Chains, workers and cores

Chains and simulations compete for the same cores, so `num_chains`, `workers` and the fit's `chain_method` must be chosen together. There are three sensible setups. All produce the same number of genuine chains and equivalent ESS — they differ in speed and in how many simulations are in flight at once.

fit's `chain_method`	pass `num_chains=`	workers	cores/worker	sims at once
(single chain)	omit	`n_cpu`	1	`n_cpu`
`'parallel'`	`N`	`n_cpu // N`	`N`	`n_cpu // N`
`'vectorized'`	omit	`n_cpu`	1	`n_cpu`

Single-chain fits are the default and the quickest way through a large batch when you do not need a per-fit R-hat.

`chain_method='parallel'` puts one chain on one JAX device, so it needs a device per chain. Pass `num_chains` to the runner and take it back in your function, so the core budget and the fit can never disagree:

```
def one_sim(N, seed, num_chains, **_):
    ...
    m.fit(model, obs=data, num_chains=num_chains,
          chain_method="parallel", progress_bar=False)

res = run_simulations(one_sim, scenarios, num_chains=4)
```

On a 24-core machine `num_chains=4` gives each worker 4 cores and 4 JAX devices, pins it to its own block of CPUs, and runs **6 simulations at a time** instead of 24 — simulation concurrency traded for chain concurrency.

`chain_method='vectorized'` vmaps the chains inside a *single* device, so no cores are reserved for them and every core stays free for a different simulation. Do **not** pass `num_chains` for this one: it would reserve cores that vectorized chains never use.

Measured on 24 simulations over 24 cores, each drawing 4 chains at equal ESS:

setup	wall
`workers=24`, `cores_per_worker=1`, vectorized	**17.5 s**
`num_chains=4` (6 workers × 4 cores), parallel	21.9 s
`workers=6`, `cores_per_worker=4`, vectorized	29.1 s

> The silent trap
>
> Asking a fit for `num_chains=4` with `chain_method='parallel'` *without* telling the runner leaves the worker with one device. NumPyro then draws the chains **sequentially** — correct results, needlessly slow, with a warning per fit. Passing `num_chains` to `run_simulations` prevents it.

---

4. Useful options

argument	what it does
`workers`	How many simulations run at once. Defaults to one per core, divided by `cores_per_worker`.
`fn_kwargs`	Constants passed to every call. Serialised **once per worker** rather than once per scenario — put large shared arrays here.
`seed`	Base seed. Per-simulation seeds derive from it deterministically, so a rerun reproduces regardless of worker count or completion order.
`on_error`	`'record'` (default) puts the traceback in the row's `error` column and keeps going; `'raise'` aborts the run.

argument	what it does
`backend`	`'serial'` runs in-process for debugging, so `pdb` and full tracebacks work.

Inspecting failures without losing the rest of the batch:

```
res = run_simulations(one_sim, scenarios, on_error="record")
res[res.error.notna()][["sim", "N", "error"]]
```

> Define the model at module level when you can
>
> Each worker JIT-compiles the model on its first fit. When the model is built fresh inside `one_sim` (a closure, as in the example above), *every* simulation pays full compilation because the JIT cache key changes each time. Defining it at module level lets a worker reuse its compiled executable across all the simulations it handles — often the single largest speedup available.