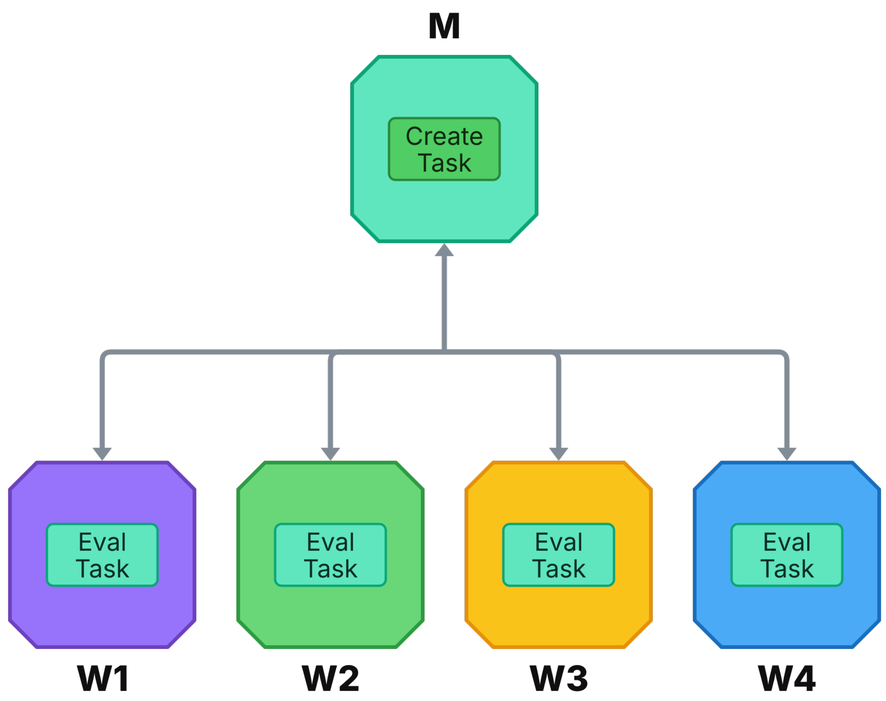
rush: Scalable Asynchronous Distributed Computing via Shared State in R

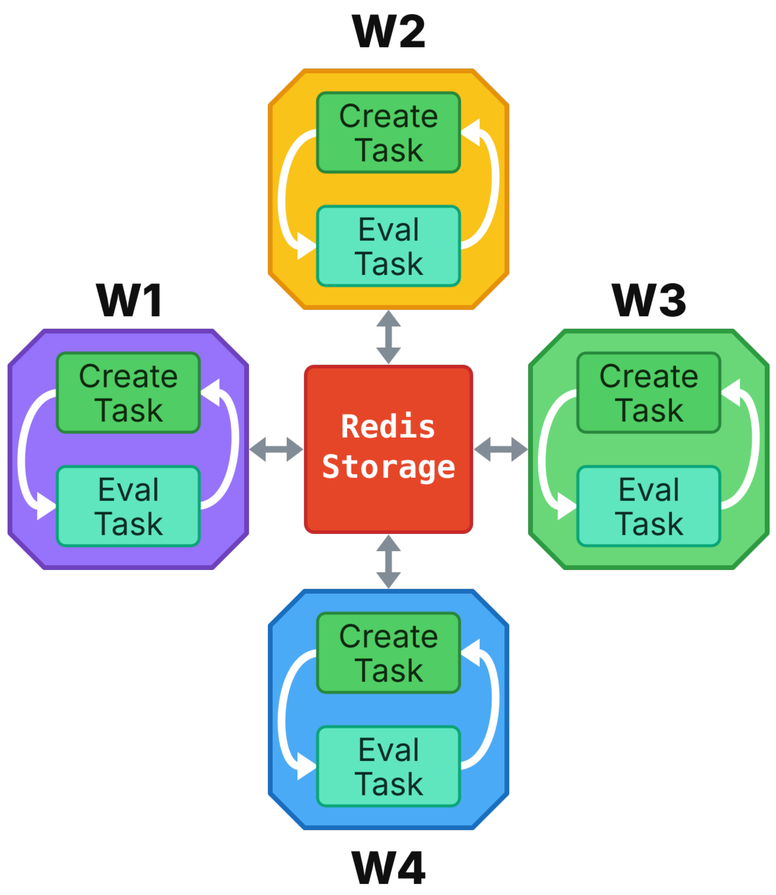
Marc Becker (Department of Statistics, LMU Munich · Munich Center for Machine Learning)

Motivation

Many iterative algorithms in **hyperparameter optimization** and **black-box optimization** no longer fit the classic **centralized controller-worker** architecture for parallel computing. Instead, workers run concurrently, each autonomously deciding what to compute next, coordinating through a shared state. Modern Python frameworks (`Optuna`, `DeepHyper`, `Hyperopt`) embrace this **decentralized design** through ask-and-tell interfaces. But every popular R package for parallel computing follows the **centralized** design: one process proposes all tasks, dispatches them, and collects results.



The main process (M) in the centralized design can become a bottleneck: the controller cannot propose tasks fast enough, leaving workers (W) idle.

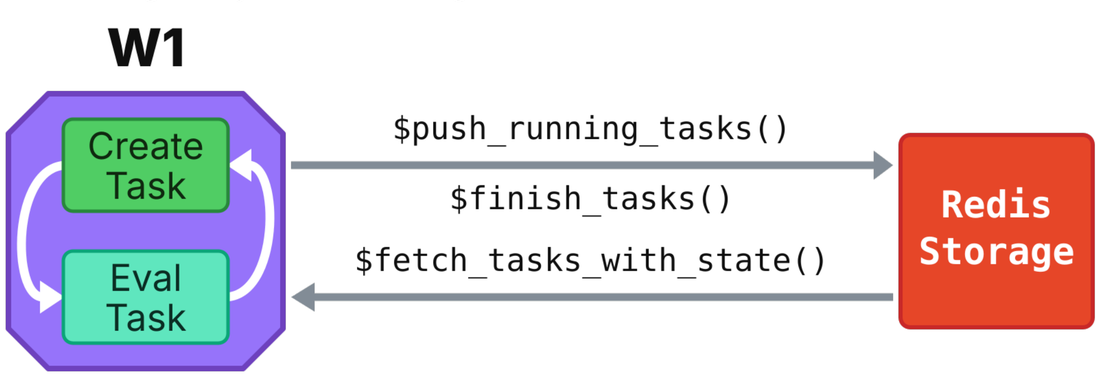


The decentralized architecture overcomes the central bottleneck by letting each worker (W) propose tasks independently.

How rush works

`rush` is a coordination layer for asynchronous iterative algorithms.

- Workers coordinate **through shared state**: a fast in-memory `Redis` database holding all task inputs, outputs, and states.
- Each worker runs its **own loop** independently, reading the shared state, proposing points, evaluating the objective, and writing results back.



`rush` provides a high-level API for communicating with the `Redis` database.

Key features

- **Asynchronous communication** between workers through a shared `Redis` database.
- **Powers asynchronous optimization** in the mlr3 ecosystem via `bbotk` and `mlr3tuning`.
- **Sub-millisecond overhead** per task for high-throughput workloads.
- **Efficient caching** that minimizes database reads and writes.
- **Task queue** for centralized task distribution when needed.
- **Scales** to large remote worker networks via `mirai`.
- **Robust error handling** across the worker network.
- **Minimal dependencies** for lightweight integration.

A minimal network

```
library(rush)

# What every worker does, over and over, on its own
worker_loop = function(rush) {
  while (rush$n_finished_tasks < 100L) {
    # Read the shared state
    archive = rush$fetch_tasks_with_state(states = c("running", "finished"))
    # Propose new point based on the archive
    xs = compute_task_inputs(archive)
    # Register the task in Redis
    keys = rush$push_running_tasks(xss = list(xs))
    # Evaluate the objective
    ys = compute_task_results(xs)
    # Write the results
    rush$finish_tasks(keys, yss = list(ys))
  }
}

# Connect the manager to Redis
config = redux::redis_config()
rush   = rsh(network_id = "my_network", config = config)

# Launch workers
mirai::daemons(2L)
rush$start_workers(worker_loop = worker_loop, n_workers = 2L)
rush$wait_for_workers(2L)

# Collect results
rush$fetch_finished_tasks()
```

Case study: ADBO at scale

We demonstrate `rush` by implementing **asynchronous decentralized Bayesian optimization (ADBO)** on top of it and tuning **nine hyperparameters of LightGBM** across **four datasets** on **448 workers** for 10 minutes per dataset. We compare three parallelization strategies:

- **CL** — synchronous batch BO (constant liar): one central optimizer; all workers wait for the slowest evaluation in each batch.
- **ACBO** — asynchronous *centralized* BO: no batch barrier, but surrogate fitting and proposal stay on a single controller.
- **ADBO** — asynchronous *decentralized* BO on rush: every worker fits its own surrogate and proposes independently.

Effective CPU utilization is $T_{\text{optimization}}/(T_{\text{walltime}} \cdot n_{\text{workers}})$, where $T_{\text{optimization}}$ is the cumulative time spent on surrogate fitting, proposing configurations, and model training, summed across all workers.

Finished evaluations and effective CPU utilization on 448 workers (CL / ACBO / **ADBO**).

Task	Evaluations	CPU utilization
credit-g	440 / 468 / **15,958**	0.3% / 0.3% / **95%**
KDDCup09_appetency	132 / 432 / **6,201**	1.1% / 4.4% / **94%**
adult	308 / 431 / **10,375**	1.4% / 2.2% / **96%**
airlines	88 / 276 / **648**	9.4% / 29% / **99%**

ADBO on `rush` sustains 94–99% CPU utilization on 448 workers, while centralized strategies fall to single digits on short-running tasks. This yields up to **~47× as many evaluations** in the same wall-clock budget.

Availability

- **Paper:** Becker & Bischl (2026), arXiv:2606.21430.
- **Source:** github.com/mlr-org/rush
- **Docs:** rush.mlr-org.com
- **Benchmarks:** github.com/slds-lmu/benchmark_2026_rush

mlr3 Machine Learning in R

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mcml Munich Center for Machine Learning

Scan for the repository