Tutorial Nix

CARLA22

Quentin GUILLOTEAU, Adrien FAURE, Jonathan BLEUZEN,
Millian POQUET, Olivier RICHARD

Université Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

2022-09-26

The goal of this tutorial

- Present Reproducibility issues
- Show that current solutions do not tackle those issues
- Present Nix as a promising solution
- Show you how we use it in the Datamove team
- Let you play with it!

The Reproducibility Problem

Different Levels of Reproducibility

1. **Repetition**: Run exact same experiment
2. **Replication**: Run experiment with different parameters
3. **Variation**: Run experiment with different environment

↪ **Share the experimental environment and how to build/modify it**

How to share a Software Environment in HPC?

- Containers? ⇝ need `Dockerfile` to rebuild/modify. might not be repro (e.g., `apt update`, `curl`, commit)
- Modules? ⇝ cluster dependent. how to modify?
- Spack? ⇝ share through modules...

Why is it important?

Control your software environment!

- Use/develop/test/distribute software
  - Manually install many dependencies?
  - Shared env for whole team (tunable) and test machines
  - Bug only on my machine? Means this is hardware or OS related
- Reproducible research
  - Repeat experiment in exact same environment
  - Introduce or test variation

Listing the versions of the dependencies is not enough!

$\hookrightarrow$ How to easily rebuild from there?

Nix and NixOS

The Nix Package Manager

- Functional Package Manager
- Packages are functions
  - Inputs = dependencies
  - Body of function = how to build
- Nix Lang $\simeq$ JSON + λ
- ($\simeq$) Solves Dependencies Hell
- Reproducible by design
- No side effects:
  - fails if undeclared dep.
  - new pkg cannot break existing ones
- Started in 2003
- 12k commits, 47k C++ LOC

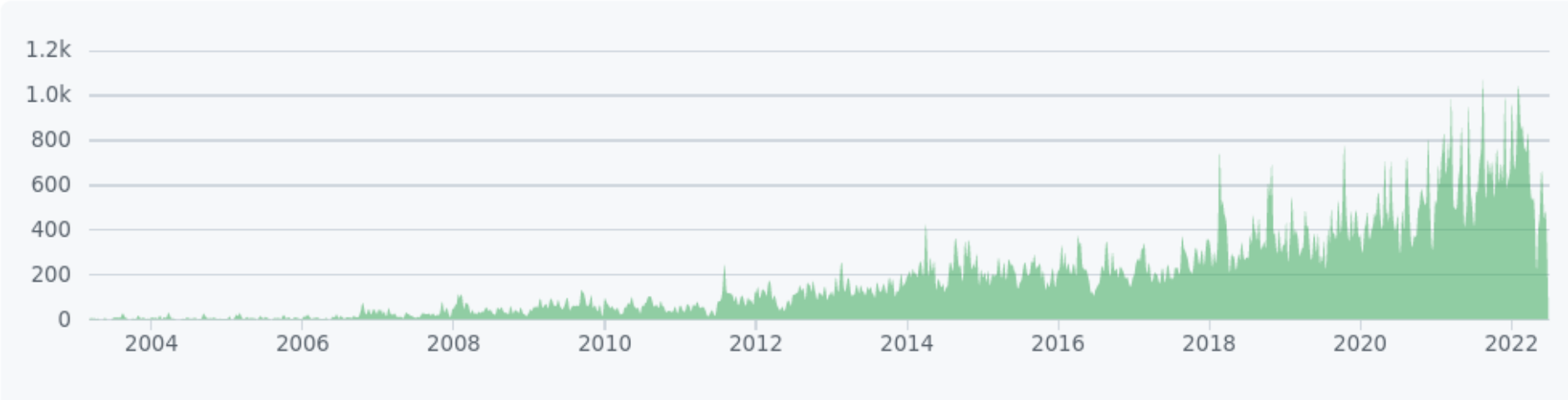
The NixOS Linux Distribution

- Based on Nix
- Declarative approach
- Complete description of the system (kernel, services, pkgs)

Nixpkgs: The Nix Packages Repository

`nixpkgs: https://github.com/NixOS/nixpkgs`

- Git repository
- Only contains Nix Expressions of the packages
- 340k commits, 85k packages[1]
- **Pinning the commit of `nixpkgs` ensure reproducibility of build**
- Binary cache of stable packages $\rightsquigarrow$ faster builds



[1] `https://repology.org/repositories/statistics`

Main ideas on building a Nix package

Build in a sandbox

- pure env variables
- no network access (src fetched by Nix, not by user code)
- no inter-process communication
- isolated filesystem

Build phases

- `unpackPhase`
- `patchPhase`
- `configurePhase`
- `buildPhase`
- `checkPhase`
- `installPhase`

Example of Package

```
1  { stdenv, fetchgit, simgrid, boost, cmake }:
2
3  stdenv.mkDerivation rec {
4    pname = "chord";
5    version = "0.1.0";
6
7    src = fetchgit {
8      url = "https://gitlab.inria.fr/me/chord";
9      rev = "069d2a5bfa4c40...";
10     sha256 = "sha256-ff4f...";
11   };
12
13   buildInputs = [ simgrid boost cmake ];
14
15   # configurePhase = "cmake .";
16   # buildPhase = "make";
17   # installPhase = "mkdir -p $out/bin && mv chord $out/bin";
18 }
```

Dependencies

Sources

Build Info

Derivation

Override Inputs

```
{ pkgs ? import (fetchTarball {
    url = "https://github.com/NixOS/nixpkgs/[...].tar.gz";
    sha256 = "sha256:[...]";}) {}
}:
let
  packages = rec {
    chord = pkgs.callPackage ./chord.nix { };
    chord_custom = chord.override {
      simgrid = simgrid-330;
      boost = boost-167;
    };

    boost-176 = ...;
    boost-167 = ...;
    boost = boost-176;

    simgrid-330 = ...;
    simgrid-331 = ...;
    simgrid = simgrid-331;
  };
in packages
```

Pinning

Pkg Def

Override

```
nix-build -A chord_custom
```

Nix Store

All packages in */nix/store*

- Isolated packages
- *Hash(inputs, source code)-packagename*
- Package names known before build → binary cache

```
/nix/store
└── hash-packagename
    ├── bin
    │   └── packagename
    └── lib
        └── libpackagename.so
```

How to store the packages?

Usual approach: **Merge them all**

- Conflicts
- `PATH=/usr/bin`

```
/usr
├── bin
│   └── myprogram
└── lib
    ├── libc.so
    └── libmylib.so
```

Nix approach: **Keep them separated**

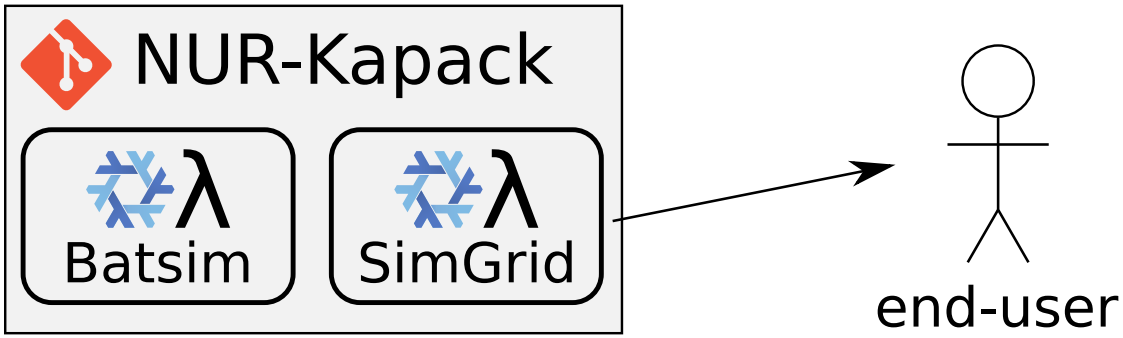
+ Pkg variation
+ Isolated
+ Well def. `PATH`
+ Read-only

```
/nix/store
├── y9zg6ryffgc5c9y67fcmfdkyyiivjzpj-glibc-2.27
│   └── lib
│       └── libc.so
└── nc5qbagm3wqfg2lv1gwj3r3bn88dpqr8-mypkg-0.1.0
    ├── bin
    │   └── myprogram
    └── lib
        └── libmylib.so
```

How we use Nix

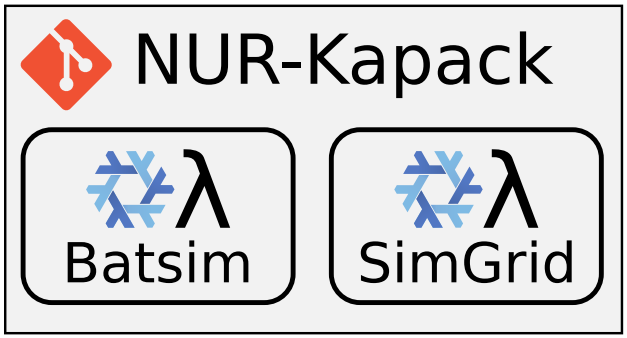
Write Nix expressions.

How we use Nix



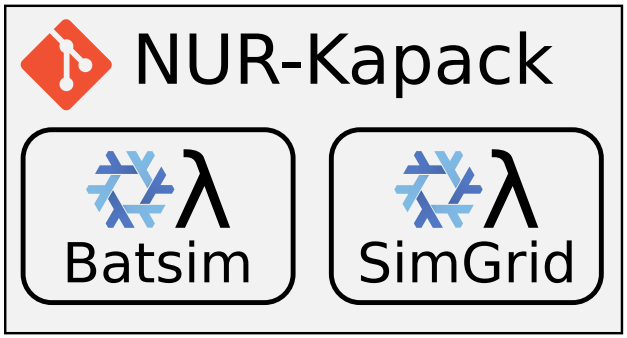
Put them in a Git repository.

How we use Nix

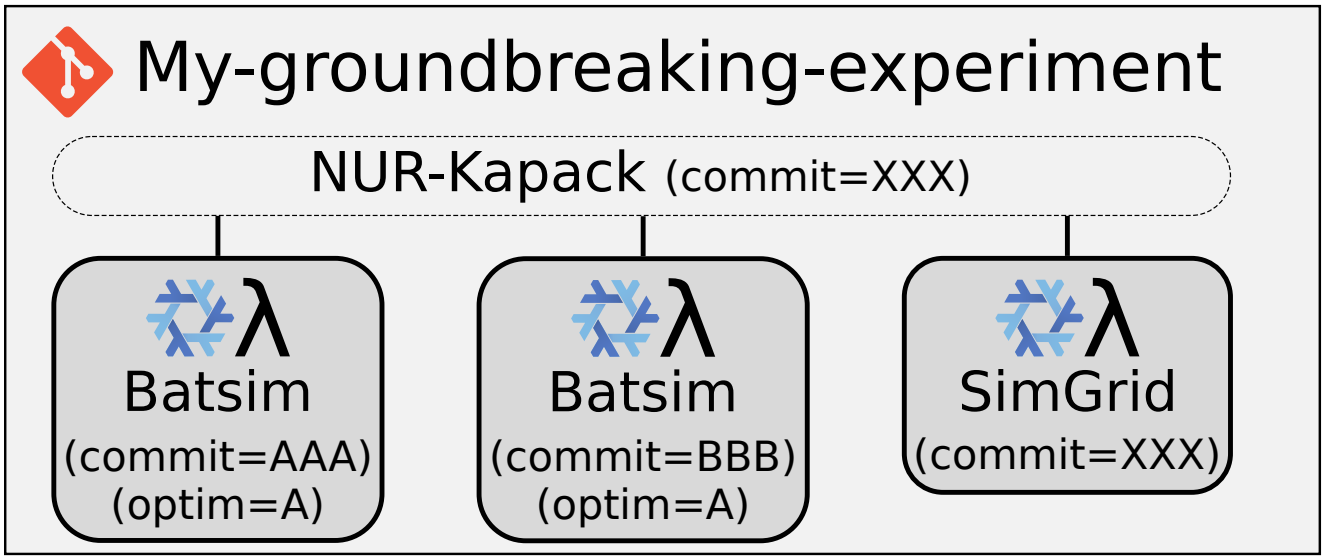


Use them for your experiment.

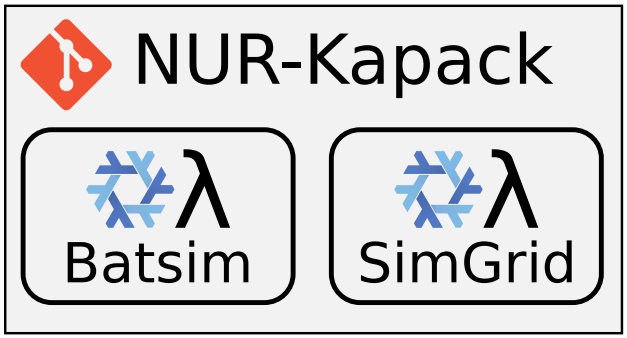
How we use Nix

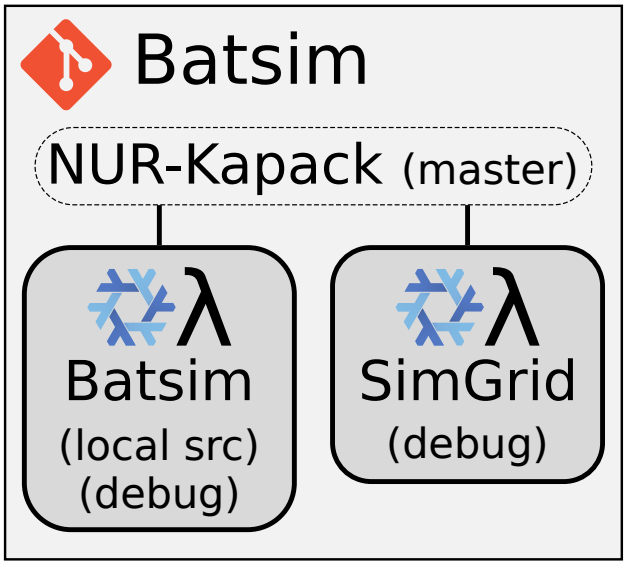


Customize them.

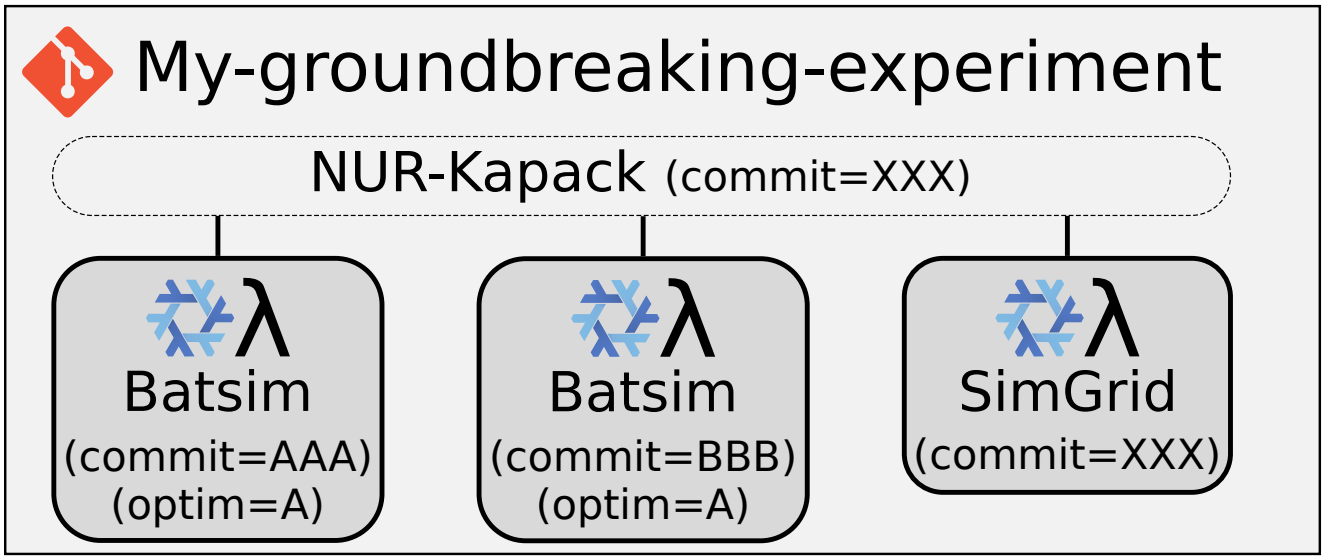


How we use Nix





Use them for dev, including CI.



Nix critique

Strengths

+ No missing dependencies, local build likely works anywhere
+ Traceable dependencies (pinned Nixpkgs)
+ `nix-shell` = multi-language `virtualenv`
+ Minimal size docker container generation is trivial
+ Distributed Nix expressions

Weaknesses

- Contaminant: dependencies must be expressed in Nix
- Learning curve + change in pratice
- Implicit behaviors to build packages (looks magic at first sight)
- External storage (github, gitlab,...)

Take home message

Nix in a nutshell

- Define pure packages (build in sandbox)
- Control and isolate your environments
- Sharing of packages/expressions:
  `https://github.com/oar-team/nur-kapack`

Steep learning curve, but worth it

- If you want to make sure your code runs in 5 years
- If you want to escape dependency hell

Your turn!

What you will do now

1. install Nix on your machines (or G5K)
2. package the `chord` application (distributed hashtable)
3. build a docker image of `chord`
4. start a package repository
5. use this repository to start experiments

`https://tinyurl.com/TutoReproCarla22`

If you have time

- **package your own applications!** (don't hesitate to ask for help)
- play with Nix Flakes