Playing with Nix in adverse HPC environments
FOSDEM 2023

Rodrigo Arias Mallo
rodarima@gmail.com

Raúl Peñacoba Veigas
raul.penacoba.veigas@gmail.com

2023-02-05
What do we do

- We develop a concurrent task-based runtime for High Performance Computing (HPC)
- LLVM-based compiler to interpret the `#pragma` directives (like OpenMP)
- Performance is **critical**
- We typically execute benchmarks on ~1000 CPUs
Example of a performance problem

Nbody normalized time. Particles=49152

Good case (normal):

Bad case (outlier):

Allocators took too long
Typical HPC scenario

- Login + compute nodes controlled by SLURM
- No root permission in any node
- Old LTS kernels 4.4 and software stack (5 years old)
- `LD_LIBRARY_PATH` used to change versions of libraries
- Hard to reproduce results after 1 year in the same machine
Question: Can we benefit from Nix?

- Up-to-date package versions, including glibc (finally)
- Explicit control over all build configuration options
- No more LD_LIBRARY_PATH evil dance
- Traceability: What library version did you use for experiment X?
Question: Can we benefit from Nix?

• Up-to-date package versions, including glibc (finally)
• Explicit control over all build configuration options
• No more LD_LIBRARY_PATH evil dance
• Traceability: What library version did you use for experiment X?
• Problem: Root daemon is not allowed (considered unsafe, refused to install by sysadmin)
The big picture

- Typically go back and forth between cycles
- Each iteration must be fast (i.e. cached builds)
- We'll focus on the Development and Execution cycles
Multi-user nix store with auxiliary machine

- Individual user installation can be done with user namespaces.
- We can use an auxiliary machine to do the builds.
- Use `post-build-hook` script to issue `nix copy`.
- Use a wrapper in the login node to mount the namespace.
- Then run the `nix-store --serve` to receive the derivation.
- Limited success with patching nix-daemon to run as a normal user (no systemd user services allowed).
Experimentation cycle

Program

Experimentation

Execution

Results
Requirement 1: safe execution environment

- Assume the program is **already built** with `nix-build` in a sandboxed environment.
- At runtime, the program may **load other libraries** (outside the nix store) with `LD_LIBRARY_PATH` or `dlopen()`.
- We can create a safe execution environment similarly to the sandbox in `nix-builds` that **prevents** accesses to dangerous paths like `/usr` and `/opt`.
- It needs to work with SLURM too.
Requirement 2: fast MPI intra-node communication

- Intra-node communication uses the `process_vm_readv()` system call.
- It only works if both processes are in the *same namespace*.
- Reenter the same namespace in the node if exists, otherwise create a new one.
Reentering the mount namespace

SLURM will allocate the nodes and fork a process in the first rank.
We enter the namespace from the first compute node to load /nix.
Reentering the mount namespace

Run `srun` from nix store so we can link MPI against the same PMI2 version.
Reentering the mount namespace

SLURM will fork a step process outside the safe environment.
Reentering the mount namespace

We reenter the namespace or create a new one if it doesn't exist.
Reentering the mount namespace

Finally, we can safely run the workload in parallel using one user namespace per node.
Requirement 3: custom software

- We provide an overlay with our custom packages extending nixpkgs.
- Sometimes we need to add patches or specific versions to upstream packages.
- Avoid rebuilding other packages keeping the scope inside an attribute set with a custom callPackage.

```nix
last: prev:
  with last.lib;
let
  inherit (last.lib) callPackageWith;
  inherit (last.lib) callPackagesWith;

  _custom = makeExtensible (custom:
    let
      callPackage = callPackageWith (last // custom);
      in
      {
        inherit callPackage;
        xyz = callPackage ./custom/xyz/default.nix { };# Select default MPI library
        mpi = custom.mpich;
        mpich = callPackage ./custom/mpich/default.nix { };
        mpichDebug = custom.mpich.override {
          enableDebug = true;
        };
        ...
      };
    in
    { custom = _custom; }
```

[1]https://pm.bsc.es/gitlab/rarias/bscpkgs
Requirement 4: custom compilers

- We need to use several compilers to build our benchmarks
- Custom LLVM compiler added with the `wrapCCWith` mechanism and setup hooks
- We also packaged some proprietary compilers with the help of `rpmextract`, `autoPatchelfHook` and *some* patience.
Development cycle

- Source code
- Development
- Build
- Program
Requirement 1: fast development cycle

Issuing a full nix-build:

```bash
~ $ time -p nix-build llvm..
...
real 645.33
user 35119.64
```

Issuing a full nix-build + ccache:

```bash
~ $ time -p nix-build llvm..
...
real 102.86
user 1618.21
```

Reusing the previous build:

```bash
~/llvm $ vim src/...
~/llvm $ time -p ninja
...
real 8.03
user 57.01
```

- Developer wants to do fast changes and recompile without building the whole application again.
- Rebuilding the entire project with `nix-build` takes a lot of time in each cycle.
- Reuse the current source and configuration without copying to the nix store
- Cache previous builds with `make`
Requirement 2: prevent accesses to /opt or /usr

```cpp
find_program(
    HIP_HIPCC_EXECUTABLE
    ...
    ENV HIP_PATH
    /opt/rocm /opt/rocm/hip
    PATH_SUFFIXES bin
    NO_DEFAULT_PATH
)
```

- The nix-shell environment is **not isolated**
- Assume that software always tries to find software in the system
- Silent contamination with system software is not easy to detect (it builds and runs fine)
- Patching every package is doable but very time consuming
- We need to provide an **isolated development environment** without access to the system software.

$ grep /opt CMakeOutput.log
Found HIP installation: /opt/rocm
Isolated development shell

Create a isolated mount namespace

```bash
~ $ nix-wrap
wrap ~ $ ls /usr
not found
wrap ~ $ nix-build -E '(import <nixpkgs> {}).
    runCommand "test" {} "ls /"'
...
bin build dev etc nix proc tmp
```

Enter a development nix-shell:

```bash
wrap ~/llvm $ nix-shell
nix-shell ~/llvm $ vim src/...
nix-shell ~/llvm $ time -p ninja
real 10.43
user 54.68
```

Test a compiled application:

```bash
nix-shell ~/apps/bin $ srun nix-wrap ./app1
```

- The `nix-wrap` script uses **bubblewrap** to create the user namespace with isolated mount (no /usr or /opt)
- The `nix-build` command can create a **nested namespace** to perform the build in a sandbox.
- Use nix-shell with extra development packages.
- Configure phase cannot access to /opt or /usr
Requirement 3: tune for specific CPUs

- We need to build **only a subset** of packages tuned for the target CPU
- Hand crafted arguments to `-march`, `native` doesn’t even work (non-reproducible too)
- Build a stdenv with the proper `hostPlatform`
- Prevent massive rebuild with `stdenv` inside our custom scope only

```plaintext
gccDetails = {
    arch = "armv8.2-a+crypto+fp16fml";
    tune = "tsv110";
    cpu = "tsv110";
};

stdenvNoCC = self.stdenvNoCC // rec {
    hostPlatform = self.stdenvNoCC.hostPlatform // {
        gcc = gccDetails;
    };
    targetPlatform = hostPlatform;
};
gcc11' = self.gcc11.override {
    stdenvNoCC = custom.stdenvNoCC;
};

stdenv = self.overrideCC self.stdenv custom.gcc11';
```
Conclusions

• We can benefit from the properties provided by Nix but with some drawbacks.
• Development and experimentation cycles can still be done quickly.
• A rootless nix daemon with a shared nix store will solve most of the problems among trusted peers in HPC.
Conclusions

- We can benefit from the properties provided by Nix but with some drawbacks.
- Development and experimentation cycles can still be done quickly.
- A rootless nix daemon with a shared nix store will solve most of the problems among trusted peers in HPC.

Thank you!