Monadic Composition
for Deterministic, Parallel Batch Processing

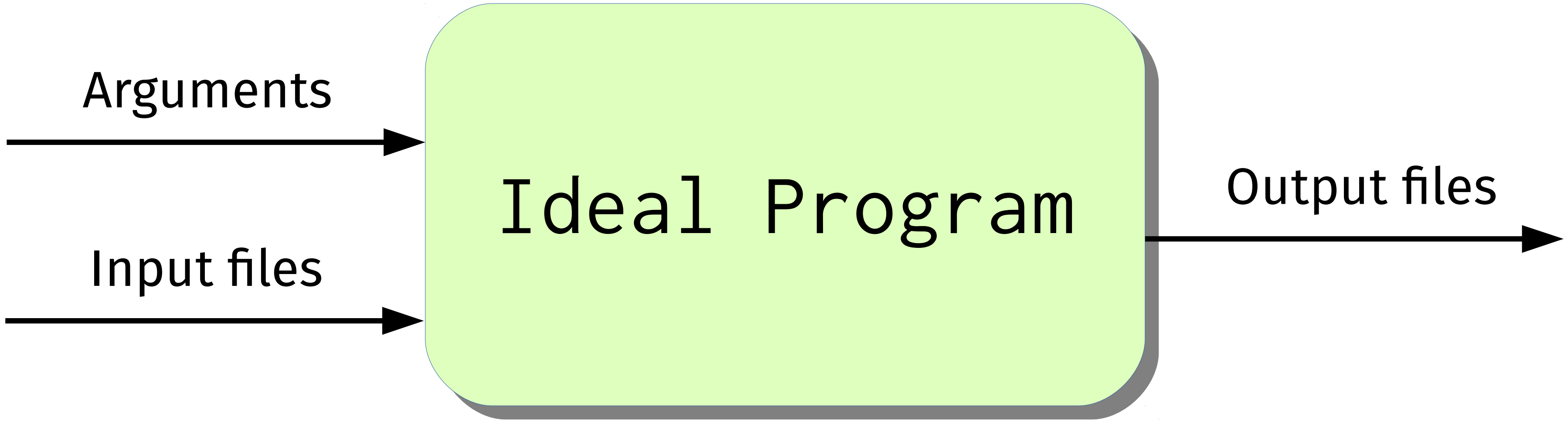
Ryan Scott[1]
Ryan Newton[1]

Omar Navarro Leija[2]
Joe Devietti[2]

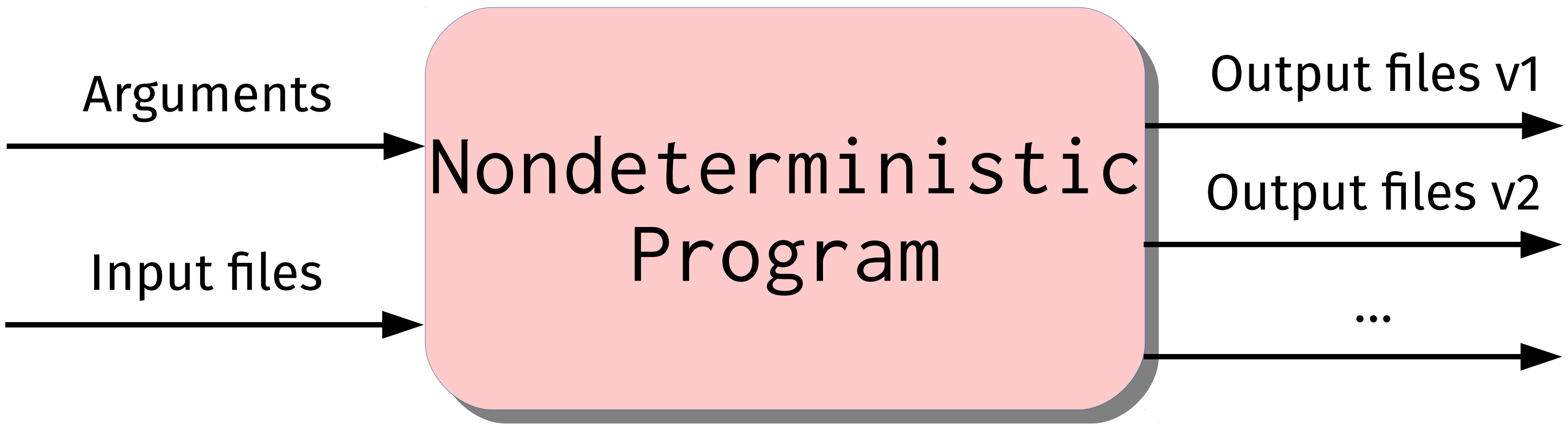
[1]Indiana University
[2]University of Pennsylvania

rgscott@indiana.edu
github.com/RyanGlScott

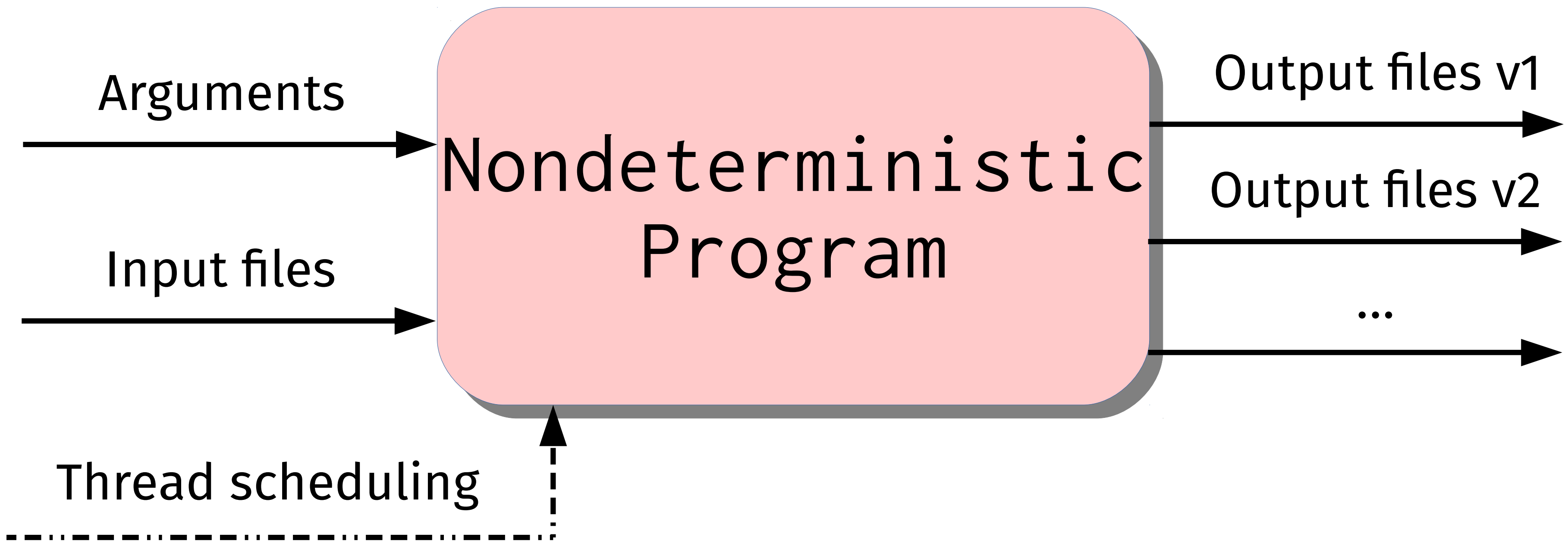
Nondeterminism



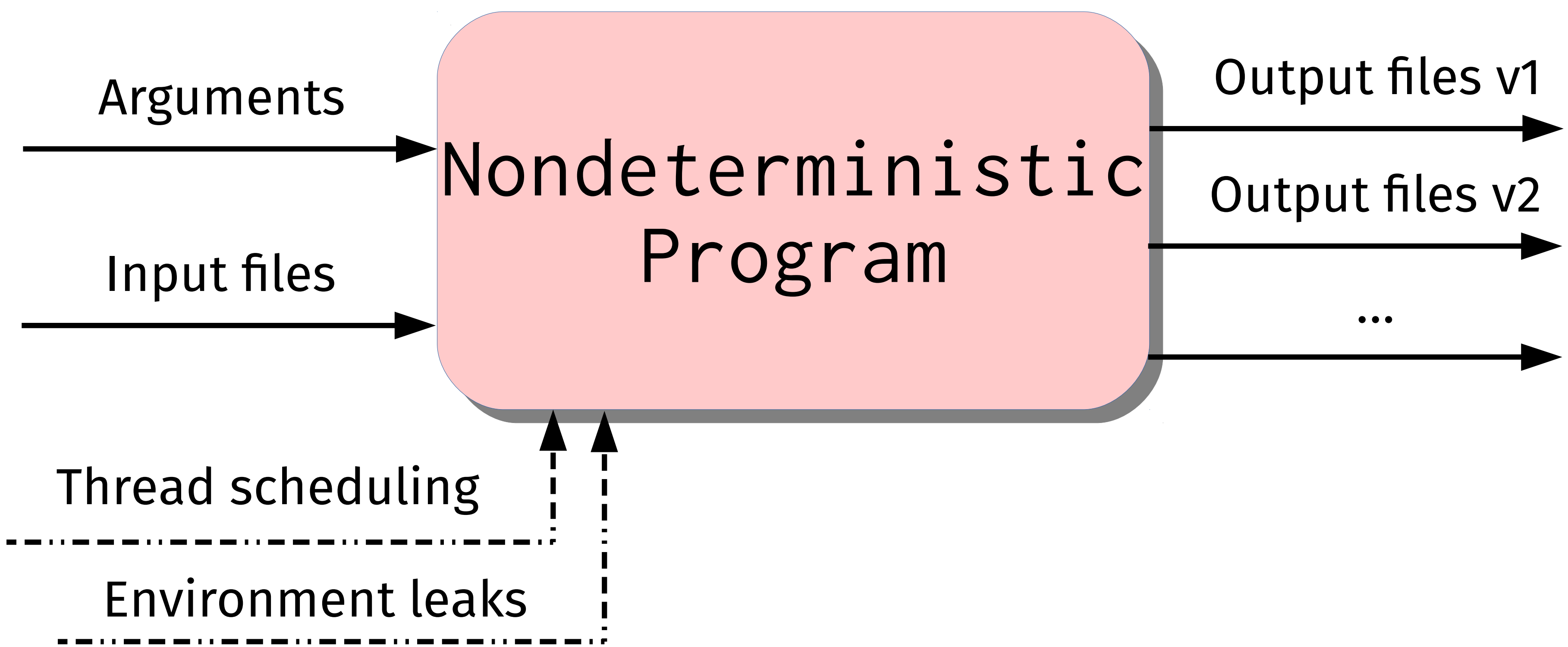
Nondeterminism



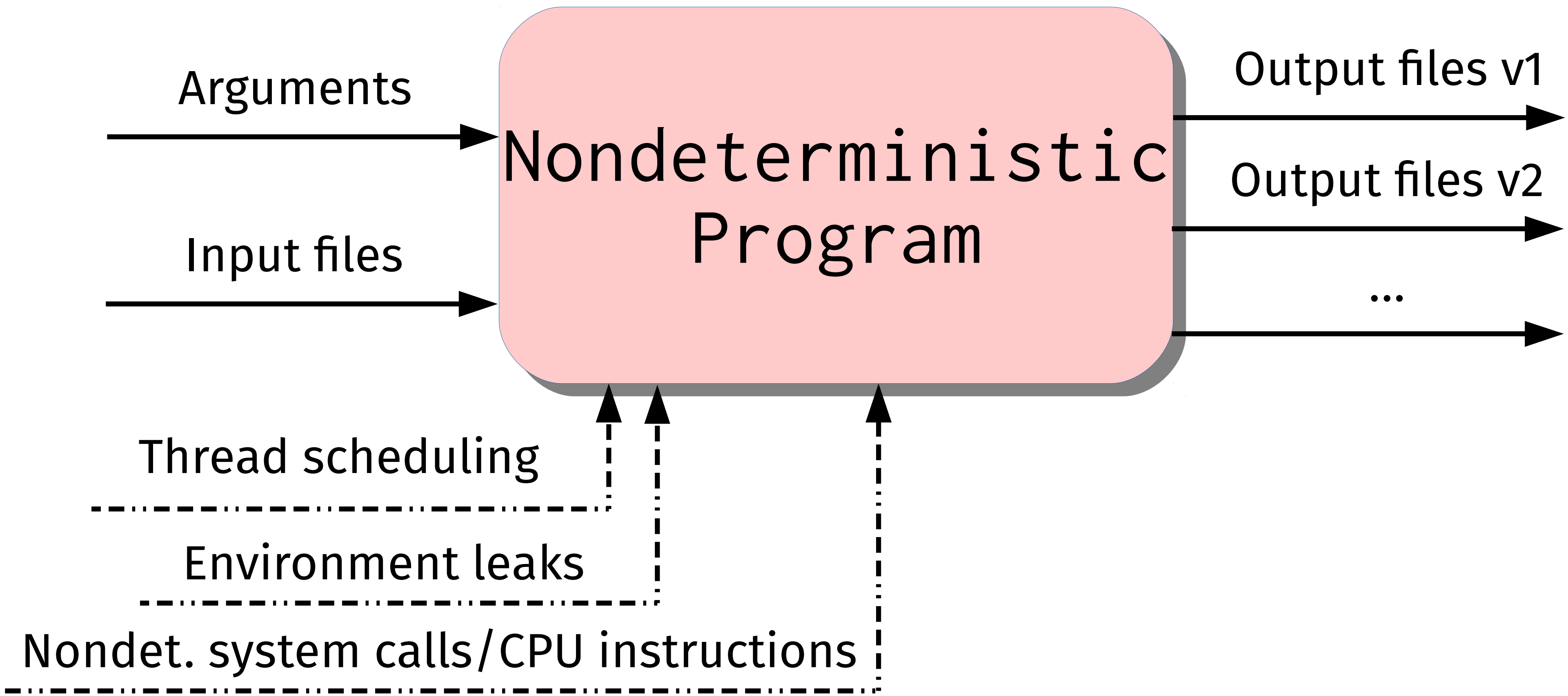
Nondeterminism



Nondeterminism



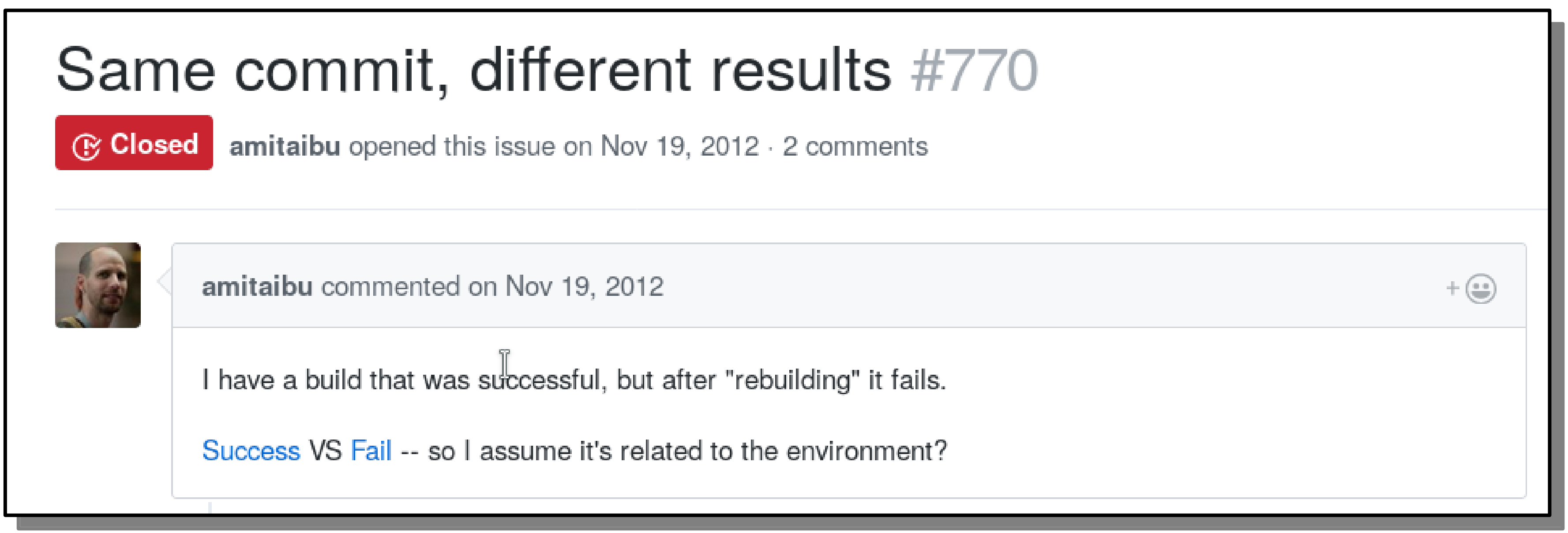
Nondeterminism



Where nondeterminism hurts

Where nondeterminism hurts

Continuous integration



Where nondeterminism hurts

Parallel workflows

```
all: a b c
a b c:
	@for n in 1 2 ; do \
		echo $@-$$n && sleep 1 ; \
	done
```

Serial execution

```
$ make
a-1
a-2
b-1
b-2
c-1
c-2
```

Parallel execution

```
```

Serial execution

```
$ make
a-1
a-2
b-1
b-2
c-1
c-2
```

Parallel execution

```
$ make -j2
a-1
a-2
b-1
b-2
c-1
c-2
```

Serial execution

```
$ make
a-1
a-2
b-1
b-2
c-1
c-2
```

Parallel execution

```
$ make -j2
a-1
b-1
b-2
a-2
c-1
c-2
```

Serial execution

```
$ make
a-1
a-2
b-1
b-2
c-1
c-2
```

Parallel execution

```
$ make -j2
a-1
b-1
a-2
b-2
c-1
c-2
```

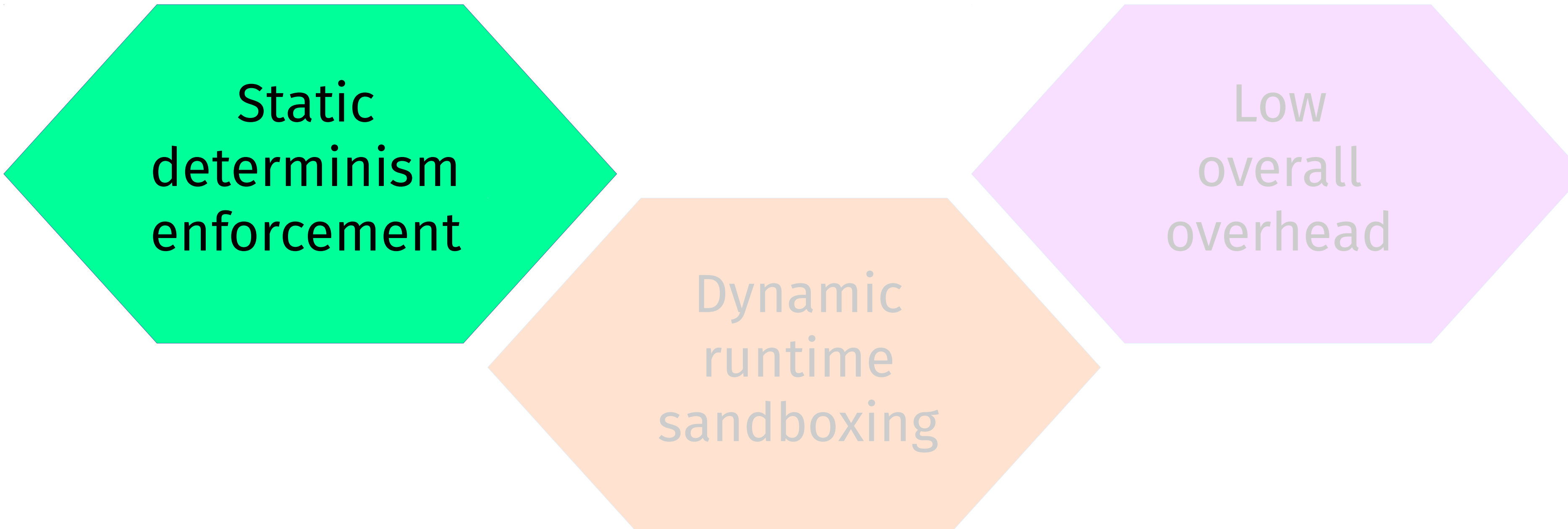
detflow

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

detflow



Static determinism enforcement
Dynamic runtime sandboxing
Low overall overhead

Entrypoints

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

Haskell

```
main :: IO ()
```

Entrypoints

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

Haskell

```
main :: IO ()
main = do
  Parallel.mapM_ putStrLn [1..10]
```

Entrypoints

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```haskell
main :: IO ()
main = do
  Parallel.mapM_ putStrLn [1..10]
  -- Already nondeterministic!
```

DetIO

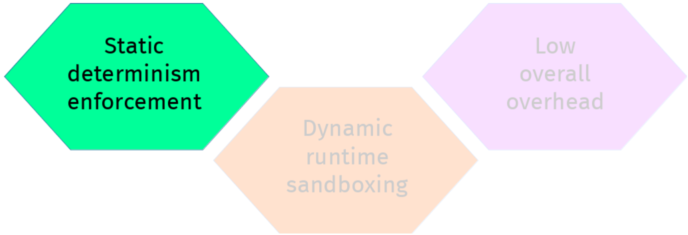
Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

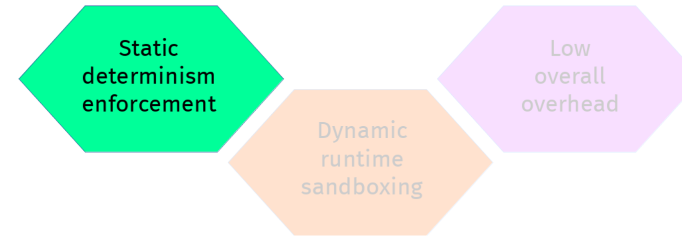
```
newtype DetIO a = MkDetIO (IO a)
```

DetIO




```haskell
newtype DetIO a = MkDetIO (IO a)
-- Expose only deterministic API calls
getLine   :: DetIO String
putStrLn :: String -> DetIO ()
-- etc.
```

DetIO




```haskell
newtype DetIO a = MkDetIO (IO a)
-- Expose only deterministic API calls
getLine  :: DetIO String
putStrLn :: String -> DetIO ()
-- etc.
```

Key idea: Only expose deterministic operations that can be *composed* in a deterministic fashion

DetIO

Static determinism enforcement

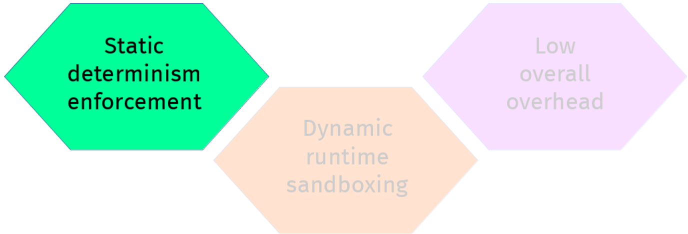
Dynamic runtime sandboxing

Low overall overhead

```haskell
newtype DetIO a = MkDetIO (IO a)
-- Expose only deterministic API calls
getLine   :: DetIO String
putStrLn :: String -> DetIO ()
-- etc.
```

```haskell
main :: DetIO ()
main = do
  x <- getLine
  putStrLn x
```

Parallelism



- `detflow` uses the filesystem for shared-memory parallelism
- Should this be allowed?

```
readFile  :: FilePath -> DetIO String
writeFile :: FilePath -> String
             -> DetIO String
```

Parallelism

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

Thread 1

```
do writeFile “foo.txt”
     “Hello, World”
```

Thread 2

```
do foo <- readFile “foo.txt”
   if foo == “Hello, World”
      then ...
      else ...
```

Parallelism

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

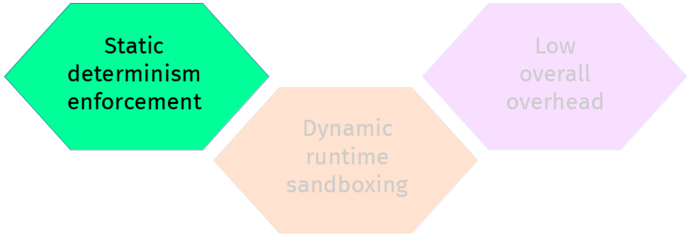
Thread 1

```
do writeFile “foo.txt”
     “Hello, World”
```

Thread 2

```
do foo <- readFile “foo.txt”
   if foo == “Hello, World”
      then ...
      else ...
```

Solution: permissions



- Every thread holds separate permissions on system filepaths

Static
determinism
enforcement

Dynamic
runtime
sandboxing

Low
overall
overhead

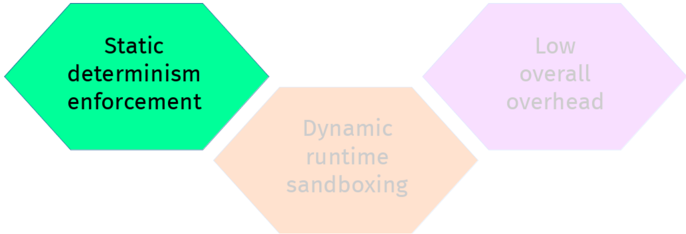
Solution: permissions

- Every thread holds separate permissions on system filepaths

`/abcdef/ghijkl/mnopqr`

Thread 1	`R 0.5`	`R 0.5`	`RW 1.0`
Thread 2	`R 0.5`	`R 0.5`	

Parallelism, revisited




```
data Perm -- (R/RW) + path + fraction

forkWPerms :: [PathPerm] -> DetIO a
              -> DetIO (Thread a)
joinThread :: Thread a -> DetIO ()
```

- `readFile` and `writeFile` must respect the permissions in a thread’s local state

Permissions checkout

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```
pgm :: DetIO ()
pgm :: do -- Assume parent starts with R 1.0 on /a
          th1 <- forkWPerms [R “/a”]
                            computation1

          th2 <- forkWPerms [R “/b”]
                            computation2

          joinThread t1

          joinThread t2
```

Permissions checkout

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```
pgm :: DetIO ()
pgm :: do -- Assume parent starts with R 1.0 on /a
          th1 <- forkWPerms [R “/a”]
                            computation1
          -- Parent has R 0.5 on /a
          th2 <- forkWPerms [R “/a”]
                            computation2

          joinThread t1

          joinThread t2
```

Permissions checkout

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```
pgm :: DetIO ()
pgm :: do -- Assume parent starts with R 1.0 on /a
          th1 <- forkWPerms [R “/a”]
                            computation1
          -- Parent has R 0.5 on /a
          th2 <- forkWPerms [R “/a”]
                            computation2
          -- Parent has R 0.25 on /a
          joinThread t1

          joinThread t2
```

Permissions checkout

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```
pgm :: DetIO ()
pgm :: do -- Assume parent starts with R 1.0 on /a
         th1 <- forkWPerms [R “/a”]
                           computation1
         -- Parent has R 0.5 on /a
         th2 <- forkWPerms [R “/a”]
                           computation2
         -- Parent has R 0.25 on /a
         joinThread t1
         -- Parent has R 0.75 on /a
         joinThread t2
```

Permissions checkout

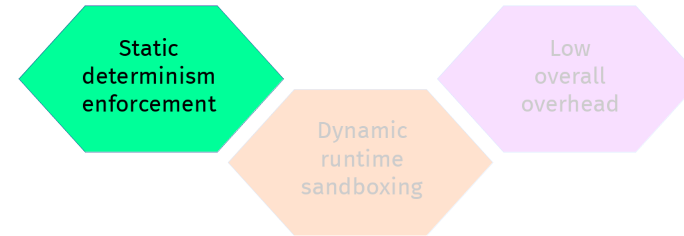
Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```
pgm :: DetIO ()
pgm :: do -- Assume parent starts with R 1.0 on /a
          th1 <- forkWPerms [R “/a”]
                            computation1
          -- Parent has R 0.5 on /a
          th2 <- forkWPerms [R “/a”]
                            computation2
          -- Parent has R 0.25 on /a
          joinThread t1
          -- Parent has R 0.75 on /a
          joinThread t2
          -- Parent has R 1.0 on /a
```

More `detflow`



- Replace nondeterministic IO operations with deterministic alternatives
  - Reading system time
  - `putStrLn`
- Full lattice of permissions, and formalization of permission checkout (see paper)

detflow

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

system calls

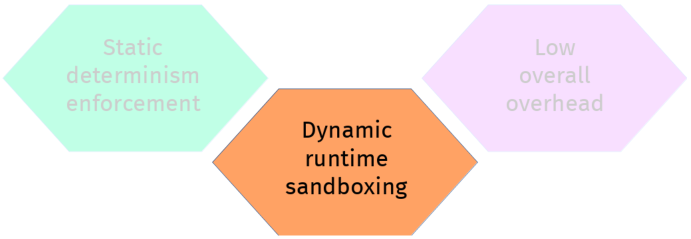
Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

```
system :: String -> DetIO ()

main :: DetIO ()
main = system “gcc foo.c -o foo”
```

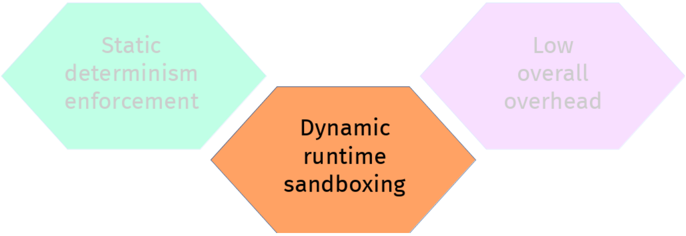
system calls




```
system :: String -> DetIO ()

main :: DetIO ()
main = system “gcc foo.c -o foo”
```

- How can we make shelling out to arbitrary programs (not written in `DetIO`) deterministic?

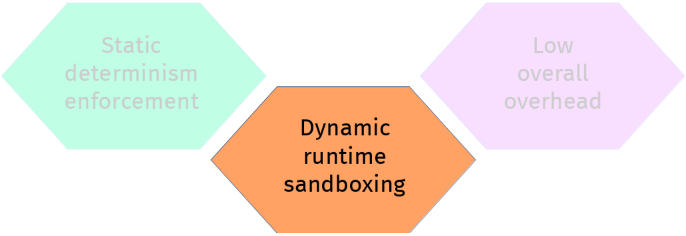
system calls



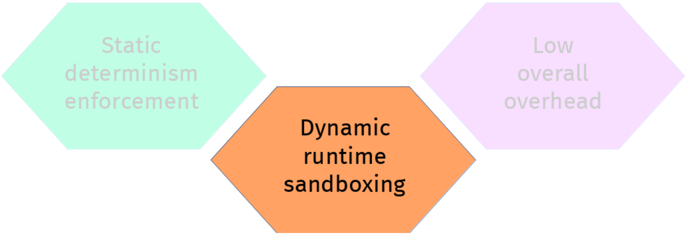

```
system :: String -> DetIO ()

main :: DetIO ()
main = system “gcc foo.c -o foo”
```

- How can we make shelling out to arbitrary programs (not written in `DetIO`) deterministic?
- Answer: run them in a *deterministic runtime*.

libdet

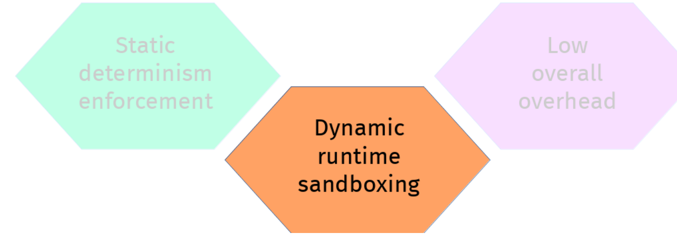

Static determinism enforcement
Dynamic runtime sandboxing
Low overall overhead

libdet



`libdet` must intercept potential sources of nondeterminism at runtime.

`libdet`



`libdet` must intercept potential sources of nondeterminism at runtime.

Reading from “banned” directories

- `/dev/urandom`
- `/proc`

libdet

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead

libdet must intercept potential sources of nondeterminism at runtime.

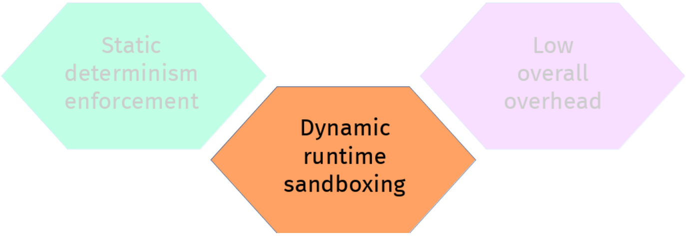
Reading from "banned" directories

- `/dev/urandom`
- `/proc`

Solution

- Intercept calls to `fopen()` (with `LD_PRELOAD`), error if they read anything blacklisted

libdet



libdet must intercept potential sources of nondeterminism at runtime.

Uncontrolled concurrency

- e.g., with pthreads

libdet

libdet must intercept potential sources of nondeterminism at runtime.

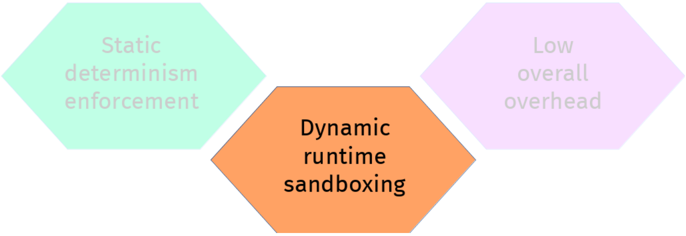
Uncontrolled concurrency

- e.g., with `pthreads`

Solution

- Intercept calls to `pthread_create()` (with `LD_PRELOAD`) to run everything sequentially

libdet

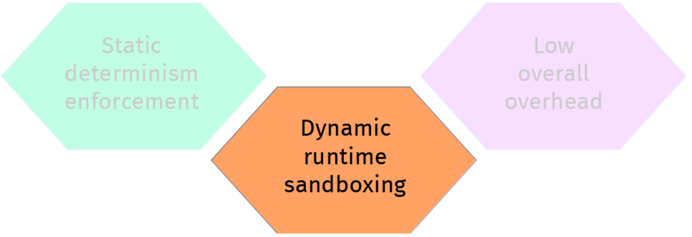


libdet must intercept potential sources of nondeterminism at runtime.

Nondeterministic OS properties

- e.g., reading addresses returned by `mmap()`

libdet



libdet must intercept potential sources of nondeterminism at runtime.

Nondeterministic OS properties

- e.g., reading addresses returned by mmap()

Solution

- Disable address-space layout randomization (ASLR)

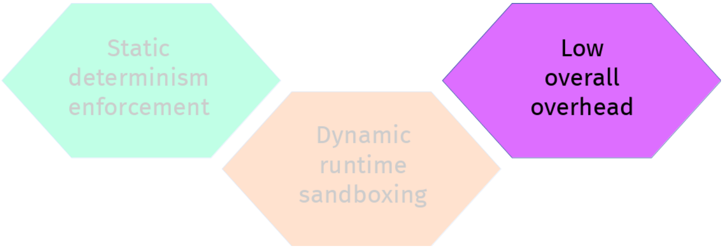
detflow

Static determinism enforcement

Dynamic runtime sandboxing

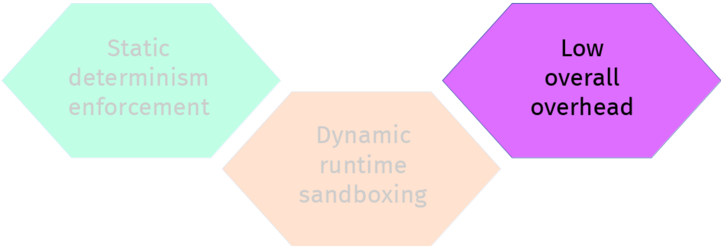
Low overall overhead

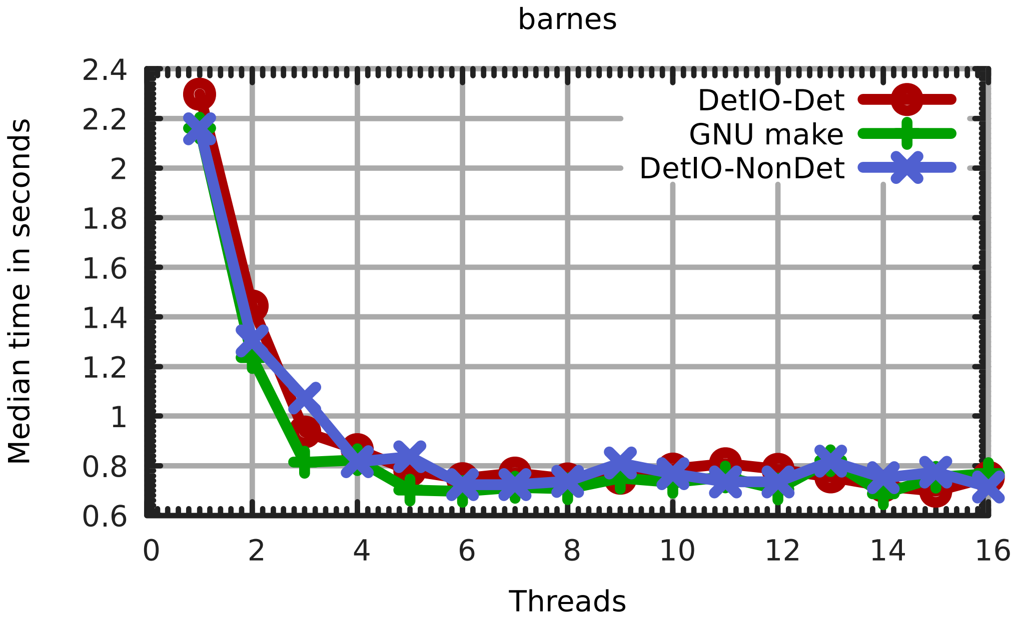
Case studies

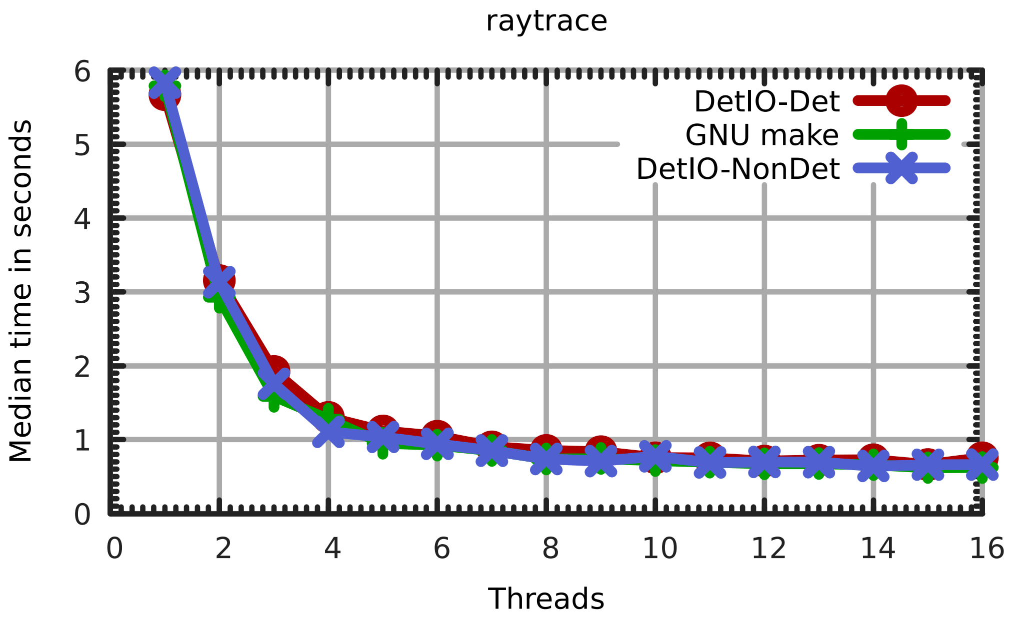


- Ran a deterministic version of `make` against SPLASH2 benchmarks
  - Performance is essentially identical to that of GNU `make`
- Ported various bioinformatics scripts to deflow and measured parallel speedup
  - Overall performance overhead for determinism enforcement is less than 1%

Selected SPLASH2 benchmarks

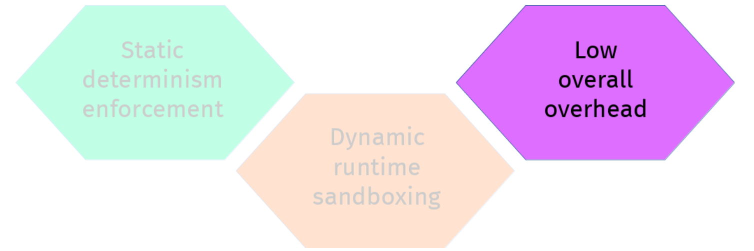


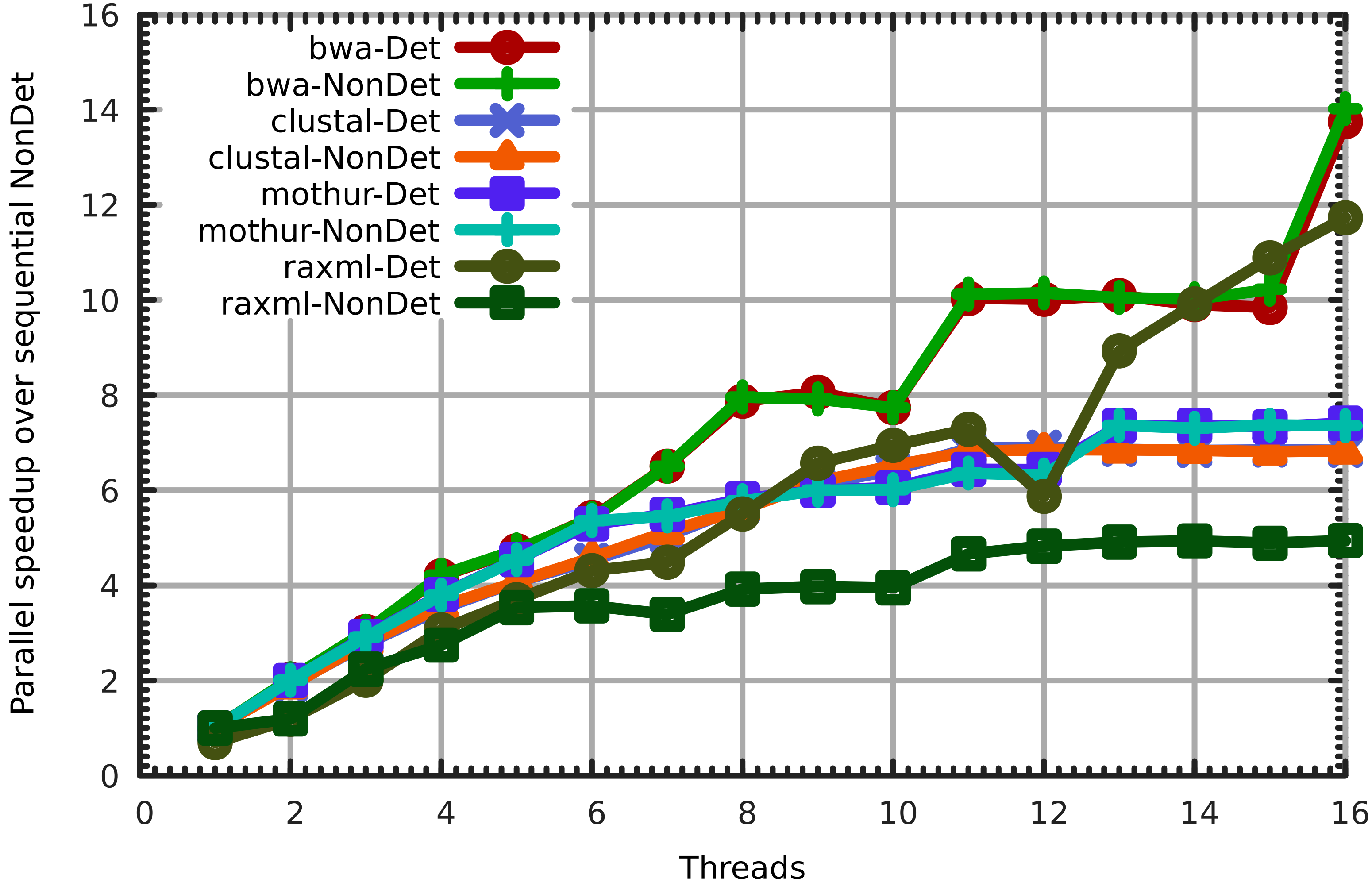




(Lower is better)

Bioinformatics apps, parallel speedup





(Higher is better)

Future work

- Reach closer to catching *all* sources of nondeterminism in runtime
- Dynamic (at-runtime) checkout of permissions
- Make more programs feasible to determinize

`detflow` development:

`https://github.com/iu-parfunc/detmonad`

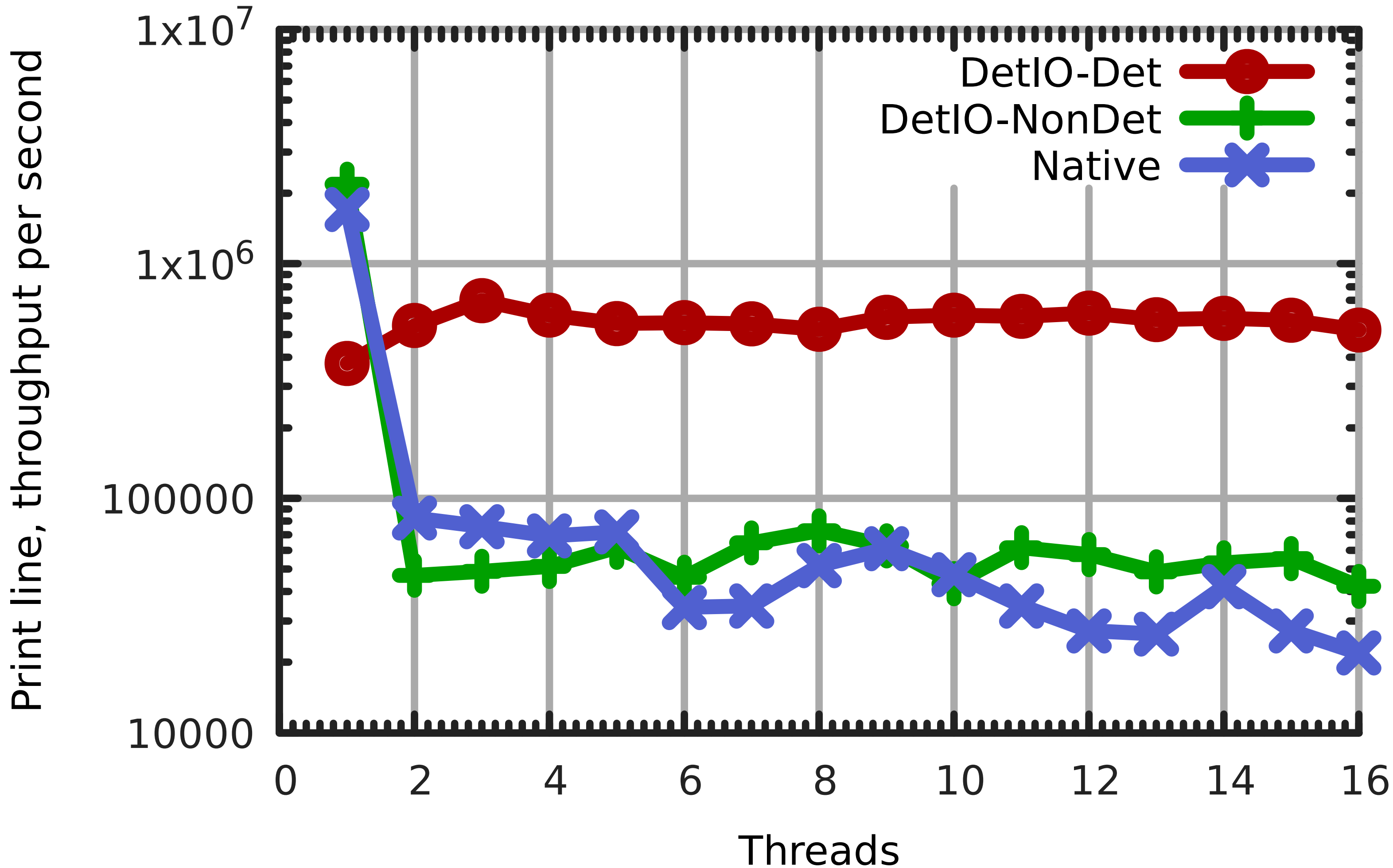
Any questions?

“Hello, World!” throughput

Static determinism enforcement

Dynamic runtime sandboxing

Low overall overhead



(Higher is better)