Understanding the Energy Behavior of Concurrent Haskell Programs

Luís Gabriel Lima

VIRTUS IMPAVIDA

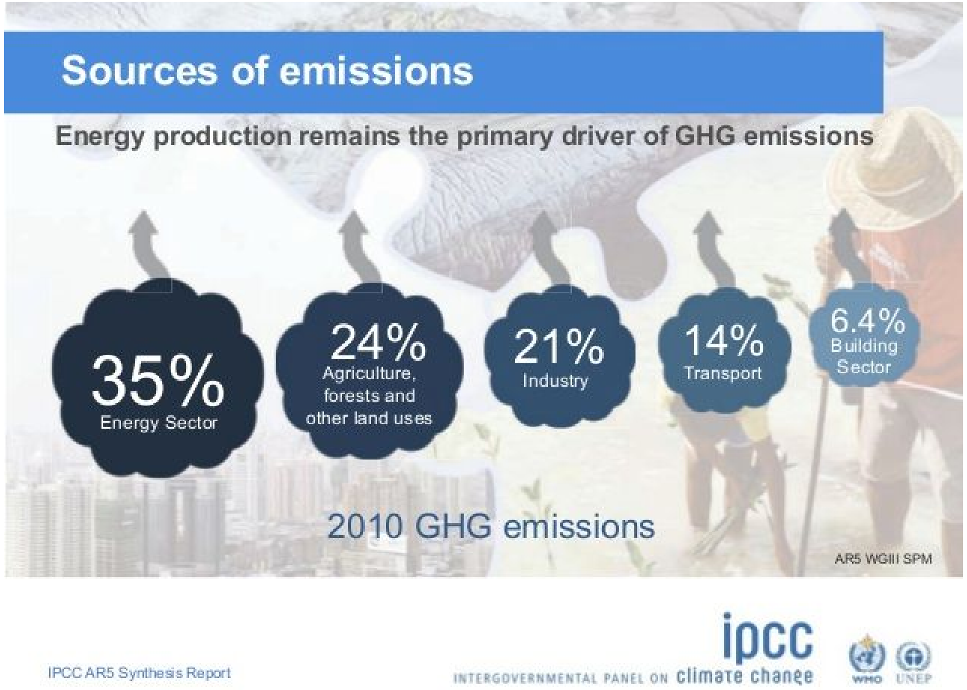
UFPE

M.Sc. Defense
Informatics Center
Federal University of Pernambuco

Recife, August 2016

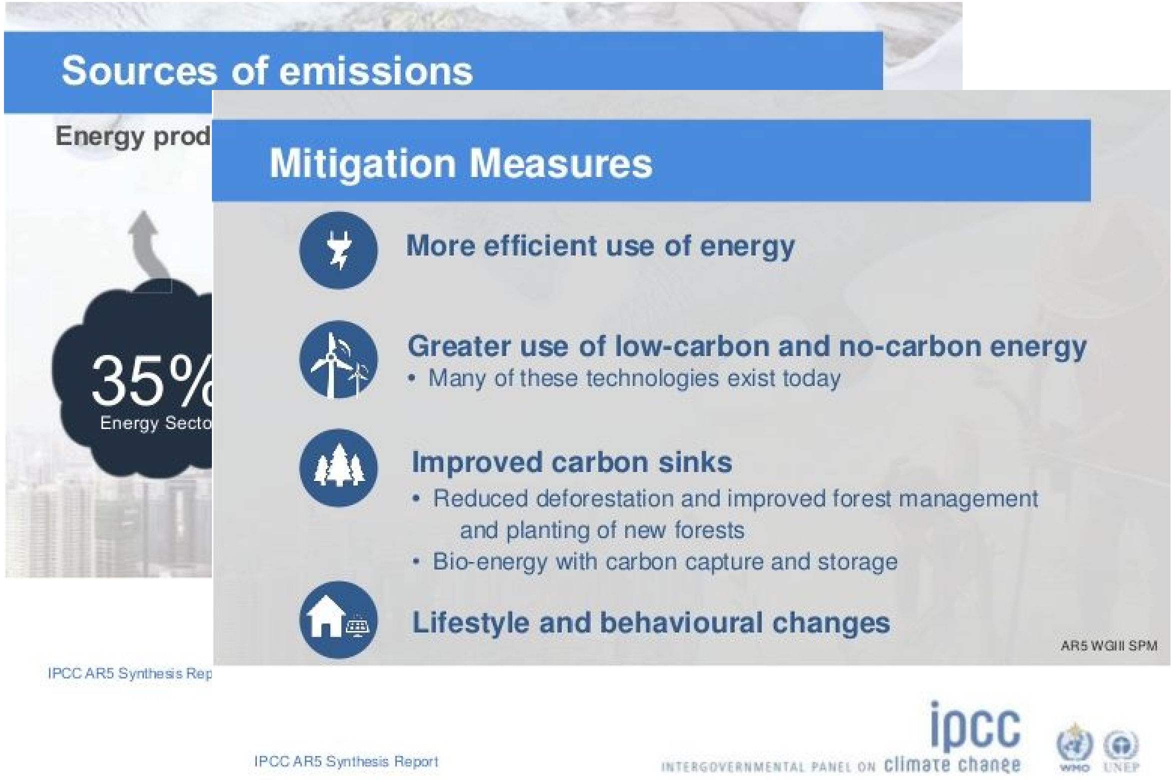
Centro de Informática
UFPE

Why does it matter?



http://pt.slideshare.net/ipcc-media/the-challenges-and-opportunities-of-climate-change-an-overview-based-on-the-ipcc-fifth-assessment-report-ar5

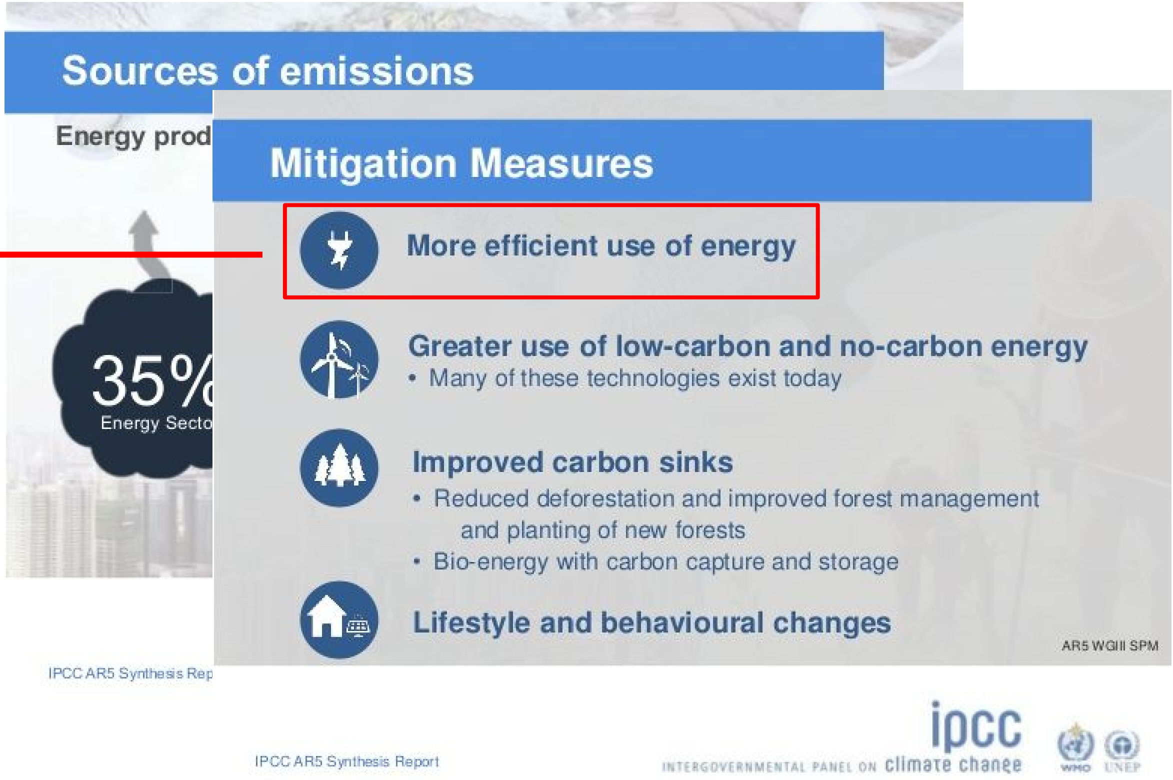
Why does it matter?



http://pt.slideshare.net/ipcc-media/the-challenges-and-opportunities-of-climate-change-an-overview-based-on-the-ipcc-fifth-assessment-report-ar5

Why does it matter?

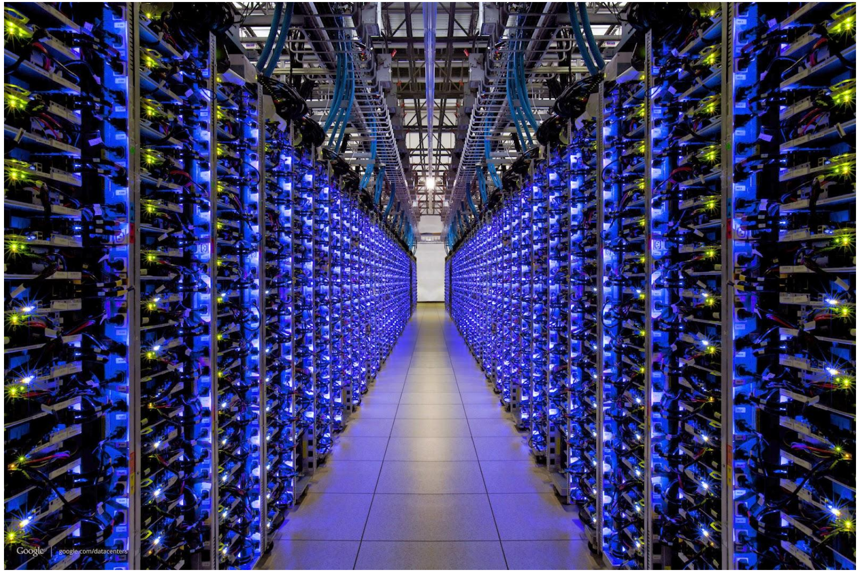
This talk!



http://pt.slideshare.net/ipcc-media/the-challenges-and-opportunities-of-climate-change-an-overview-based-on-the-ipcc-fifth-assessment-report-ar5

It goes beyond saving the planet…

(as if that wasn't important enough)



It goes beyond saving the planet…

(as if that wasn't important enough)

Bloomberg the Company & Its Products | Bloomberg Anywhere Remote Login | Bloomberg Terminal Demo Request

Bloomberg Business | News | Markets | Insights | Video

Technology

Inside the Arctic Circle, Where Your Facebook Data Lives

By Ashlee Vance | October 04, 2013

SEND TO kindle

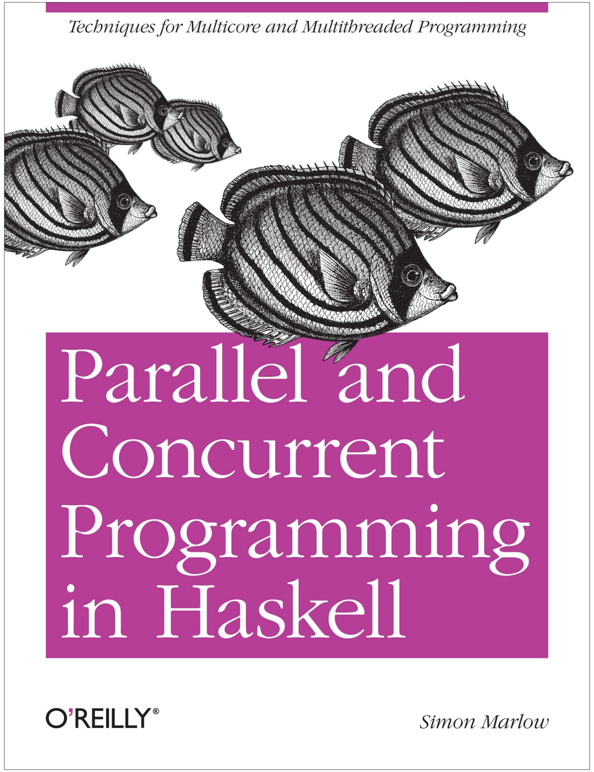
Photograph by Jasper Doest/Foto Natura/Minden Pictures/Corbis

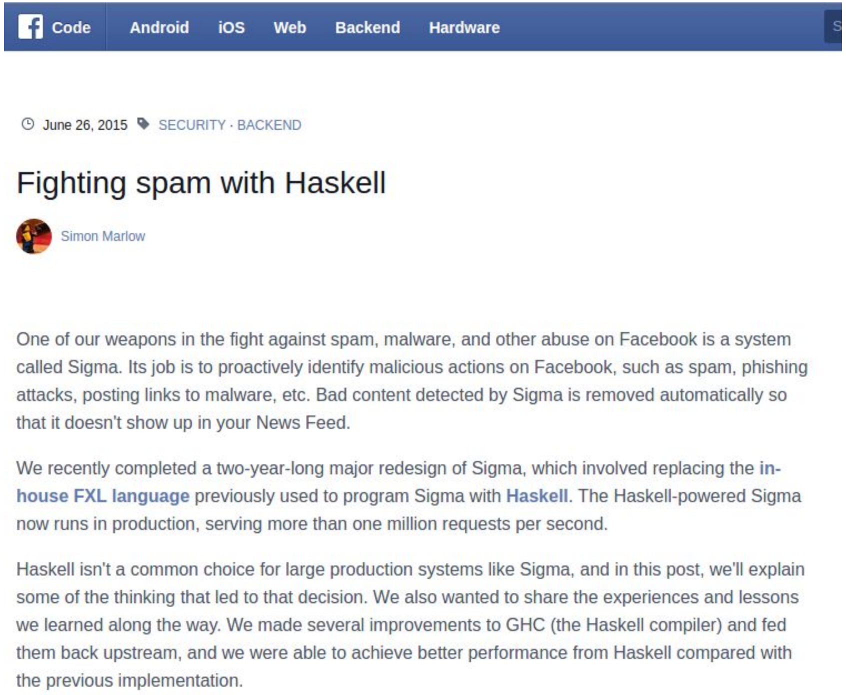
Every year, computing giants including Hewlett-Packard (HPQ), Dell (DELL), and Cisco Systems (CSCO) sell north of $100 billion in hardware. That's the total for the basic iron—

There is No Free Lunch

- **Multicore** processors are ubiquitous;
- Performance of the existing parallel software is reasonably well-understood;
- Little is known about **energy behaviors** of multi-threaded programs on the **application level**.

Haskell in the Concurrency Wilderness





Code Android iOS Web Backend Hardware

June 26, 2015 SECURITY · BACKEND

Fighting spam with Haskell

Simon Marlow

One of our weapons in the fight against spam, malware, and other abuse on Facebook is a system called Sigma. Its job is to proactively identify malicious actions on Facebook, such as spam, phishing attacks, posting links to malware, etc. Bad content detected by Sigma is removed automatically so that it doesn't show up in your News Feed.

We recently completed a two-year-long major redesign of Sigma, which involved replacing the **in-house FXL language** previously used to program Sigma with **Haskell**. The Haskell-powered Sigma now runs in production, serving more than one million requests per second.

Haskell isn't a common choice for large production systems like Sigma, and in this post, we'll explain some of the thinking that led to that decision. We also wanted to share the experiences and lessons we learned along the way. We made several improvements to GHC (the Haskell compiler) and fed them back upstream, and we were able to achieve better performance from Haskell compared with the previous implementation.

The Problem

Lack of Knowledge

Lack of Tools

The Problem

Lack of Knowledge

Lack of Tools

Mining Questions About Software Energy Consumption

Gustavo Pinto
Federal University of Pernambuco
Recife, PE, Brazil
ghlp@cin.ufpe.br

Fernando Castor
Federal University of Pernambuco
Recife, PE, Brazil
castor@cin.ufpe.br

Yu David Liu
State University of New York at Binghamton
Binghamton, NY 13902, USA
davidL@cs.binghamton.edu

ABSTRACT

A growing number of software solutions have been proposed to address application-level energy consumption problems in the last few years. However, little is known about how much software developers are concerned about energy consumption, what aspects of energy consumption they consider important, and what solutions they have in mind for improving energy efficiency. In this paper we present the first empirical study on understanding the views of application pro-

solutions are highly sought after across the compute stack, with more established results through innovations in hardware/architecture [1, 12, 26], operating systems [9, 18, 23], and runtime systems [7, 24, 29]. In recent years, there is a growing interest in studying energy consumption from higher layers of the compute stack and most of these studies focus on application software [13, 22, 25, 33, 27, 17]. These approaches complement prior hardware/OS-centric solutions, so that improvements at the hardware/OS level

The Problem

Lack of Knowledge

Lack of Tools

Oh Boy! I have no idea on how to
to improve the energy efficiency of
my concurrent program...

The Problem

Lack of Knowledge

Lack of Tools

Is there any tool to help me on that?

Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program;

2. Characterize the energy behavior of Haskell's concurrent programming constructs;

3. Provide guidelines for developers on how to write energy-efficient code.

Goals

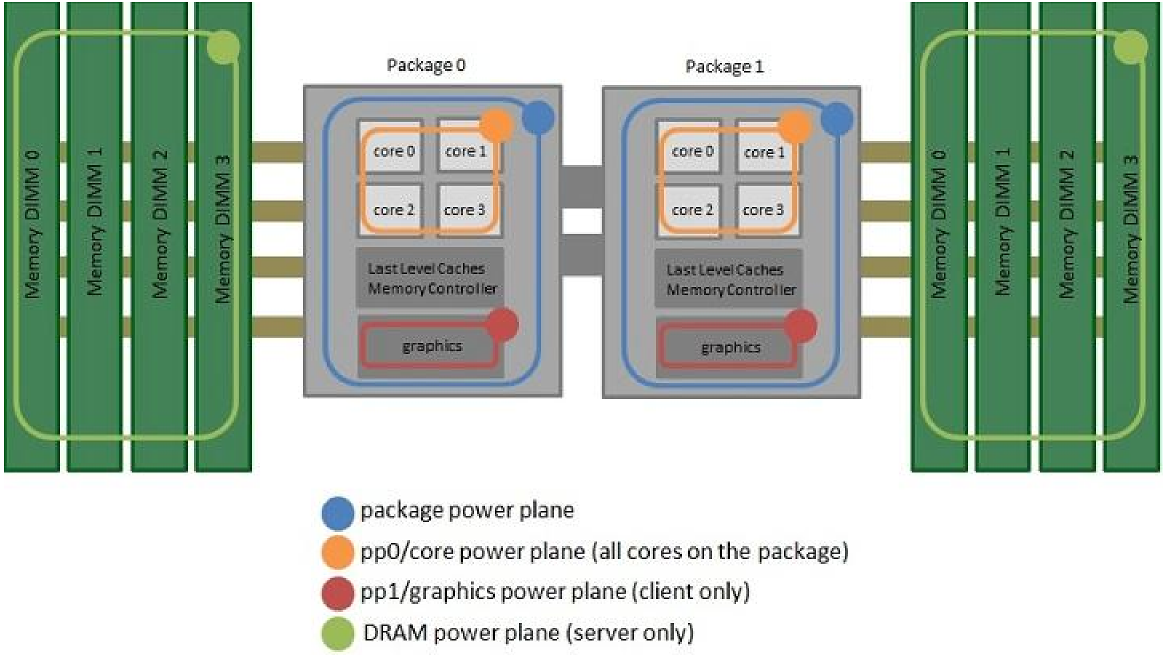
1. Enable developers to effectively measure the energy consumption of a Haskell program;

2. Characterize the energy behavior of Haskell's concurrent programming constructs;

3. Provide guidelines for developers on how to write energy-efficient code.

Measuring Energy Consumption

RAPL



https://software.intel.com/en-us/articles/intel-power-governor

Performance Analysis in Haskell

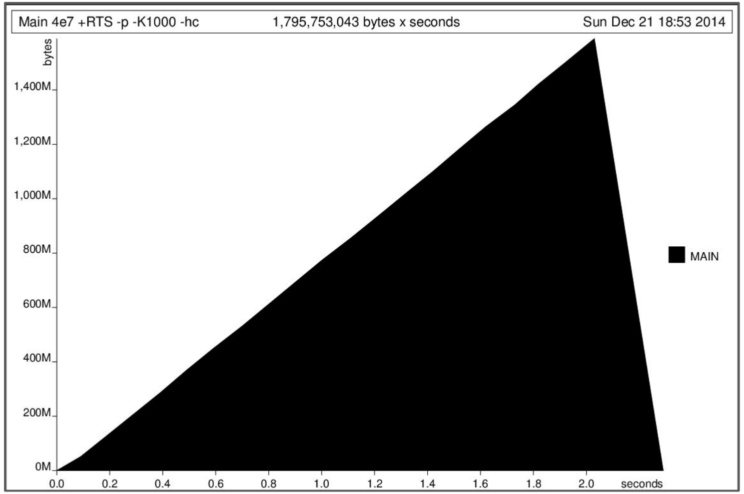
Profiling → GHC Profiler

Benchmarking → Criterion

GHC Profiler

```haskell
import System.Environment
import Text.Printf

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean [1..d])

mean :: [Double] -> Double
mean xs = {-# SCC mean #-} sum xs / fromIntegral (length xs)
```

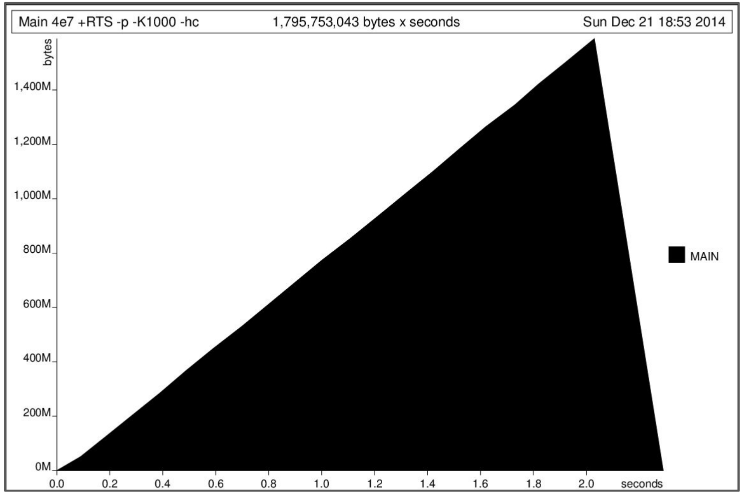
```
        Sun Dec 21 18:53 2014 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -K1000 -hc -RTS 4e7

        total time  =        2.26 secs   (2257 ticks @ 1000 us, 1 processor)
        total alloc = 6,720,116,496 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

MAIN        MAIN     84.7  100.0
mean        Main     15.3    0.0


                                                             individual     inherited
COST CENTRE MODULE                           no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                              55           0   84.7  100.0   100.0  100.0
 mean       Main                             110           1   15.3    0.0    15.3    0.0
 CAF        Main                             109           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal                  102           0    0.0    0.0     0.0    0.0
 CAF        GHC.Float                        101           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Handle.FD                  99           0    0.0    0.0     0.0    0.0
 CAF        Text.Read.Lex                     93           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding                   86           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv             85           0    0.0    0.0     0.0    0.0
 CAF        GHC.Integer.Logarithms.Internals  62           0    0.0    0.0     0.0    0.0
```



GHC Profiler

```haskell
import System.Environment
import Text.Printf

main = do
    [d] <- map read `fmap` getArgs
    printf "%f\n" (mean [1..d])

mean :: [Double] -> Double
mean xs = {-# SCC mean #-} sum xs / fromIntegral (length xs)
```

```
        Sun Dec 21 18:53 2014 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -K1000 -hc -RTS 4e7

        total time  =        2.26 secs   (2257 ticks @ 1000 us, 1 processor)
        total alloc = 6,720,116,496 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

MAIN        MAIN     84.7  100.0
mean        Main     15.3    0.0


                                                          individual     inherited
COST CENTRE MODULE                      no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                         55           0   84.7  100.0   100.0  100.0
 mean       Main                        110           1   15.3    0.0    15.3    0.0
 CAF        Main                        109           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal             102           0    0.0    0.0     0.0    0.0
 CAF        GHC.Float                   101           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Handle.FD             99           0    0.0    0.0     0.0    0.0
 CAF        Text.Read.Lex                93           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding              86           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv        85           0    0.0    0.0     0.0    0.0
 CAF        GHC.Integer.Logarithms.Internals 62       0    0.0    0.0     0.0    0.0
```



Time Profiling

- Uses **frequency counting**;

- At each tick interval (1 ms), the profiler **increments** the **counter** of the currently executing **cost-centre**;

- When the execution finishes, we can estimate the time spent by each **cost-centre**.

Energy Profiling

- Uses **accumulators**;

- At each tick, **adds** the energy consumed since the last tick to the **accumulator** of the currently executing **cost-centre**;

- When the execution finishes, each accumulator holds the energy consumed by its associated **cost-centre**.

Energy Profiling in Action

```
        Sun Feb  8 22:44 2015 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -K100M -Dp -RTS 10e6

        total time   =        1.80 secs   (1796 ticks @ 1000 us, 1 processor)
        total alloc  = 1,680,116,488 bytes  (excludes profiling overheads)
        total energy =       42.81 joules

COST CENTRE MODULE  %time %alloc %energy

MAIN        MAIN     93.3  100.0   92.9
mean        Main      6.7    0.0    7.1


                                                          individual              inherited
COST CENTRE MODULE                   no.     entries  %time %alloc %energy  %time %alloc %energy

MAIN        MAIN                     101           0   93.3  100.0   92.9   100.0  100.0  100.0
 mean       Main                     202           1    6.7    0.0    7.1     6.7    0.0    7.1
 CAF        Main                     201           0    0.0    0.0    0.0     0.0    0.0    0.0
 CAF        GHC.Conc.Signal          195           0    0.0    0.0    0.0     0.0    0.0    0.0
 CAF        GHC.Float                189           0    0.0    0.0    0.0     0.0    0.0    0.0
 CAF        GHC.IO.Encoding          182           0    0.0    0.0    0.0     0.0    0.0    0.0
 CAF        GHC.IO.Encoding.Iconv    180           0    0.0    0.0    0.0     0.0    0.0    0.0
 CAF        GHC.IO.Handle.FD         172           0    0.0    0.0    0.0     0.0    0.0    0.0
 CAF        Text.Read.Lex            144           0    0.0    0.0    0.0     0.0    0.0    0.0
```

Source Code: https://github.com/green-haskell/ghc

Energy Profiling in Action

```
        Sun Feb  8 22:44 2015 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -K100M -Dp -RTS 10e6

        total time   =        1.80 secs   (1796 ticks @ 1000 us, 1 processor)
        total alloc  = 1,680,116,488 bytes  (excludes profiling overheads)
        total energy =       42.81 joules

COST CENTRE MODULE  %time %alloc %energy

MAIN        MAIN     93.3  100.0   92.9
mean        Main      6.7    0.0    7.1


                                                    individual              inherited
COST CENTRE MODULE                  no.     entries  %time %alloc %energy %time %alloc %energy

MAIN        MAIN                    101           0   93.3  100.0   92.9  100.0  100.0  100.0
 mean       Main                    202           1    6.7    0.0    7.1    6.7    0.0    7.1
 CAF        Main                    201           0    0.0    0.0    0.0    0.0    0.0    0.0
 CAF        GHC.Conc.Signal         195           0    0.0    0.0    0.0    0.0    0.0    0.0
 CAF        GHC.Float               189           0    0.0    0.0    0.0    0.0    0.0    0.0
 CAF        GHC.IO.Encoding         182           0    0.0    0.0    0.0    0.0    0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   180           0    0.0    0.0    0.0    0.0    0.0    0.0
 CAF        GHC.IO.Handle.FD        172           0    0.0    0.0    0.0    0.0    0.0    0.0
 CAF        Text.Read.Lex           144           0    0.0    0.0    0.0    0.0    0.0    0.0
```

Source Code: https://github.com/green-haskell/ghc

Criterion Microbenchmarking Library

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0       = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

Criterion in Action

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0       = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

```
benchmarking fib/9
time                 314.4 ns   (312.2 ns .. 318.5 ns)
                     0.999 R²   (0.997 R² .. 1.000 R²)
mean                 315.3 ns   (314.0 ns .. 319.4 ns)
std dev              7.081 ns   (1.625 ns .. 14.63 ns)
variance introduced by outliers: 26% (moderately inflated)
```

Criterion in Action

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0       = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

Time estimate for running fib/9 once

```
benchmarking fib/9
time                 314.4 ns   (312.2 ns .. 318.5 ns)
                     0.999 R²   (0.997 R² .. 1.000 R²)
mean                 315.3 ns   (314.0 ns .. 319.4 ns)
std dev              7.081 ns   (1.625 ns .. 14.63 ns)
variance introduced by outliers: 26% (moderately inflated)
```

Criterion in Action

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0       = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

Time estimate for running fib/9 once

Confidence Interval

```
benchmarking fib/9
time                 314.4 ns   (312.2 ns .. 318.5 ns)
                     0.999 R²   (0.997 R² .. 1.000 R²)
mean                 315.3 ns   (314.0 ns .. 319.4 ns)
std dev              7.081 ns   (1.625 ns .. 14.63 ns)
variance introduced by outliers: 26% (moderately inflated)
```

Criterion in Action

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0      = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

Time estimate for running fib/9 once

Confidence Interval

```
benchmarking fib/9
time                 314.4 ns   (312.2 ns .. 318.5 ns)
                     0.999 R²   (0.997 R² .. 1.000 R²)
mean                 315.3 ns   (314.0 ns .. 319.4 ns)
std dev              7.081 ns   (1.625 ns .. 14.63 ns)
variance introduced by outliers: 26% (moderately inflated)
```

Coefficient of determination (0.999 R^2)

Criterion: Other Performance Metrics

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0      = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

```
benchmarking fib/9
time                 317.2 ns   (314.2 ns .. 319.4 ns)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 314.4 ns   (313.3 ns .. 315.8 ns)
std dev              4.117 ns   (2.682 ns .. 5.398 ns)
cycles:              0.999 R²   (0.999 R² .. 1.000 R²)
  iters              1079.434   (1069.292 .. 1087.144)
  y                  924904.370 (562772.048 .. 1358678.998)
variance introduced by outliers: 13% (moderately inflated)
```

Criterion: Other Performance Metrics

```haskell
import Criterion.Main

fib :: Int -> Int
fib m | m < 0       = error "negative!"
      | otherwise = go m
  where
    go 0 = 0
    go 1 = 1
    go n = go (n-1) + go (n-2)

main :: IO ()
main = defaultMain [
    bench "fib/9" (whnf fib 9)
  ]
```

```
benchmarking fib/9
time                 317.2 ns   (314.2 ns .. 319.4 ns)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 314.4 ns   (313.3 ns .. 315.8 ns)
std dev              4.117 ns   (2.682 ns .. 5.398 ns)
cycles:              0.999 R²   (0.999 R² .. 1.000 R²)
  iters              1079.434   (1069.292 .. 1087.144)
  y                  924904.370 (562772.048 .. 1358678.998)
variance introduced by outliers: 13% (moderately inflated)
```

Estimate of the number of CPU cycles required for running fib/9 once

Criterion + Energy Metrics

```
benchmarking dining-philosophers (forkOS | MVar)
time                  2.183 s    (1.915 s .. 2.510 s)
                      0.997 R²   (0.991 R² .. 1.000 R²)
mean                  2.179 s    (2.113 s .. 2.212 s)
std dev               57.17 ms   (0.0 s .. 57.19 ms)
energy:               0.999 R²   (0.997 R² .. 1.000 R²)
  iters               180.947    (164.629 .. 200.826)
  y                   0.937      (-71.359 .. 37.230)
variance introduced by outliers: 19% (moderately inflated)
```

Source Code: https://github.com/green-haskell/criterion

Criterion + Energy Metrics

```
benchmarking dining-philosophers (forkOS | MVar)
time                 2.183 s    (1.915 s .. 2.510 s)
                     0.997 R²   (0.991 R² .. 1.000 R²)
mean                 2.179 s    (2.113 s .. 2.212 s)
std dev              57.17 ms   (0.0 s .. 57.19 ms)
energy:              0.999 R²   (0.997 R² .. 1.000 R²)
  iters              180.947    (164.629 .. 200.826)
  y                  0.937      (-71.359 .. 37.230)
variance introduced by outliers: 19% (moderately inflated)
```

Estimate of the **energy** in Joules required for running dining-philosophers once

Source Code: https://github.com/green-haskell/criterion

Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program;

2. Characterize the energy behavior of Haskell's concurrent programming constructs;

3. Provide guidelines for developers on how to write energy-efficient code.

Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program;

2. Characterize the energy behavior of Haskell's concurrent programming constructs;

3. Provide guidelines for developers on how to write energy-efficient code.

Concurrency in GHC

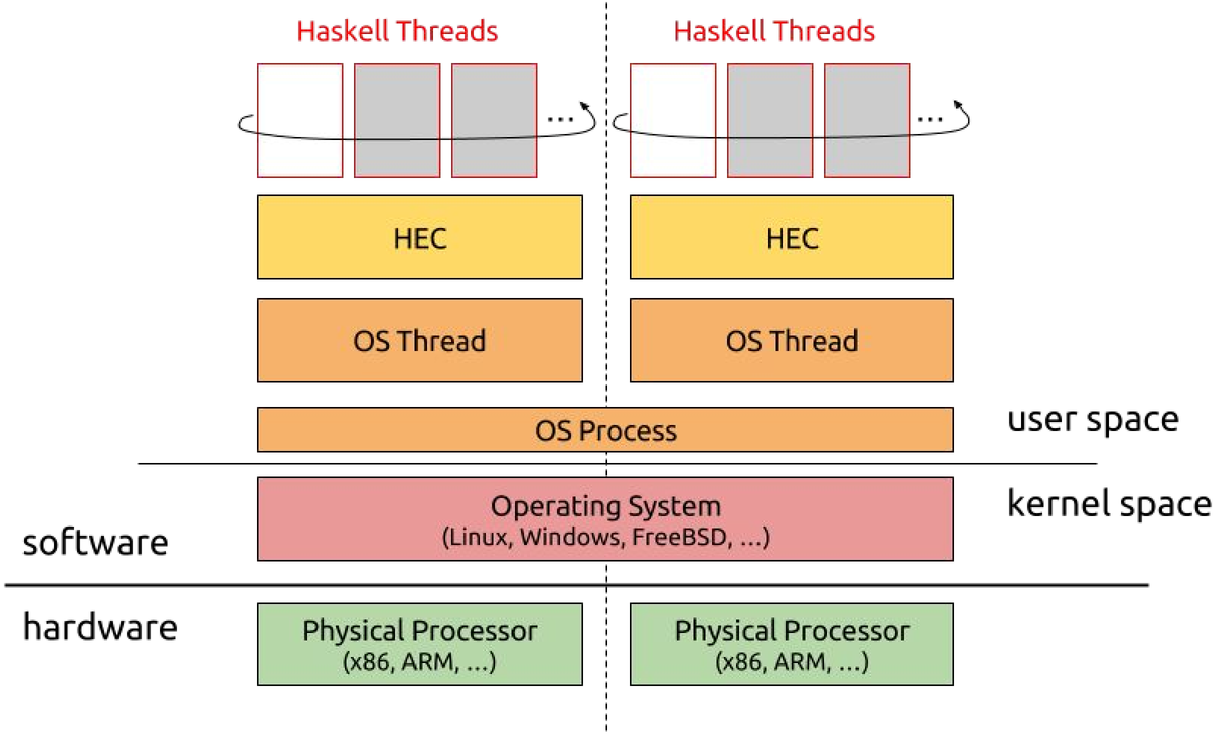
- Main abstraction: **Haskell threads**;
- Haskell threads are executed on **capabilities** (or **H**askell **E**xecution **C**ontext);
- The number of **capabilities** can be defined at runtime;
- The runtime system has its own scheduler;
- **Haskell threads** can be migrated among **capabilities**.

Concurrency in GHC

- Main abstraction: **Haskell threads**;
- Haskell threads are executed on **capabilities** (or **H**askell **E**xecution **C**ontext);
- The number of **capabilities** can be defined at runtime;
- The runtime system has its own scheduler;
- **Haskell threads** can be migrated among **capabilities**.

load balancing

Concurrency Layers



https://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

Concurrent Programming Constructs

Threading Strategies

`forkIO`

`forkOn`

`forkOS`

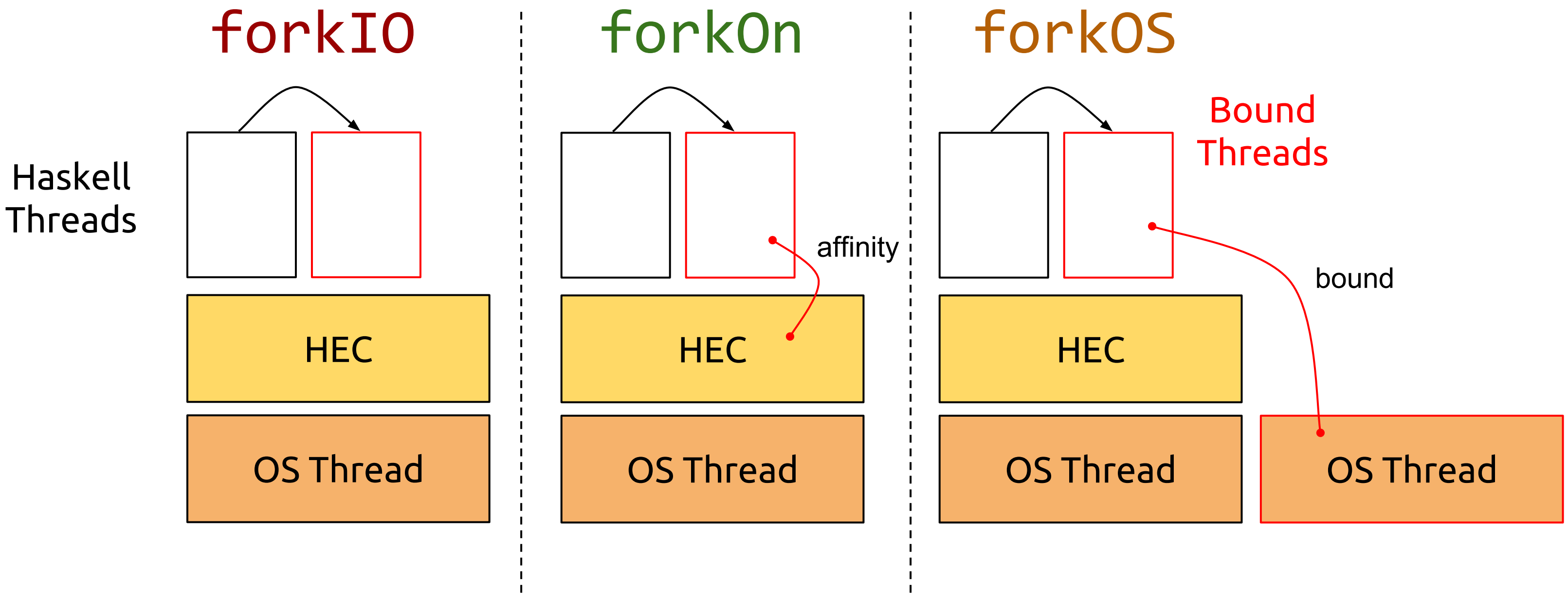
Primitives for Sharing Data

`MVar`

`TVar`

`TMVar`

Threading Strategies



https://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf

Primitives for Sharing Data

`MVar`

- Holds a single value
- Full or empty
- Blocking

`TVar`

- Holds a single value
- Never empty
- Non-blocking

`TMVar`

- Mimics an MVar
- Uses a TVar internally

Primitives for Sharing Data

`MVar`

- Holds a single value
- Full or empty
- Blocking

`TVar`

- Holds a single value
- Never empty
- Non-blocking

`TMVar`

- Mimics an MVar
- Uses a TVar internally

Can only be used inside a transaction!

Benchmarks

- **CPU-intensive**: `mandelbrot`, `spectral-norm`
- **Memory-intensive**: `k-nucleotide`, `regex-dna`
- **I/O-intensive**: `warp`
- **Synchronization-intensive**: `chameneos-redux`, `dining-philosophers`
- **Mixed**: `fasta`, `tsearch`

Benchmarks

- **CPU-intensive:** `mandelbrot`, `spectral-norm`
- **Memory-intensive:** `k-nucleotide`, `regex-dna`
- **I/O-intensive:** `warp`
- **Synchronization-intensive:** `chameneos-redux`, `dining-philosophers`
- **Mixed:** `fasta`, `tsearch`

Computer Language Benchmarks Game　Rosetta Code　Created by us

Methodology

spectral-norm

- forkIO-MVar
- forkIO-TVar
- forkIO-TMVar
- forkOn-MVar
- forkOn-TVar
- forkOn-TMVar
- forkOS-MVar
- forkOS-TVar
- forkOS-TMVar

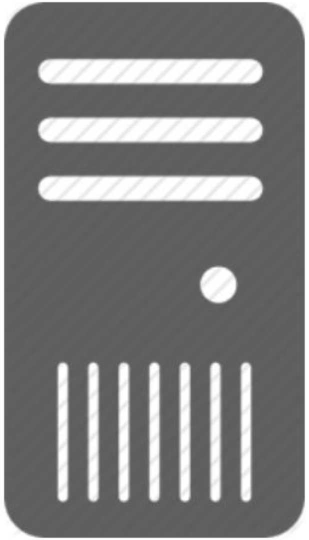
Methodology

spectral-norm

- forkIO-MVar
- forkIO-TVar
- forkIO-TMVar
- forkOn-MVar
- forkOn-TVar
- forkOn-TMVar
- forkOS-MVar
- forkOS-TVar
- forkOS-TMVar

- Each benchmark has up to 9 variants;
- Each variant is a Criterion microbenchmark;
- Each variant is executed with N = {1, 2, 4, 8, 16, 20, 32, 40, 64}

Experimental Environment



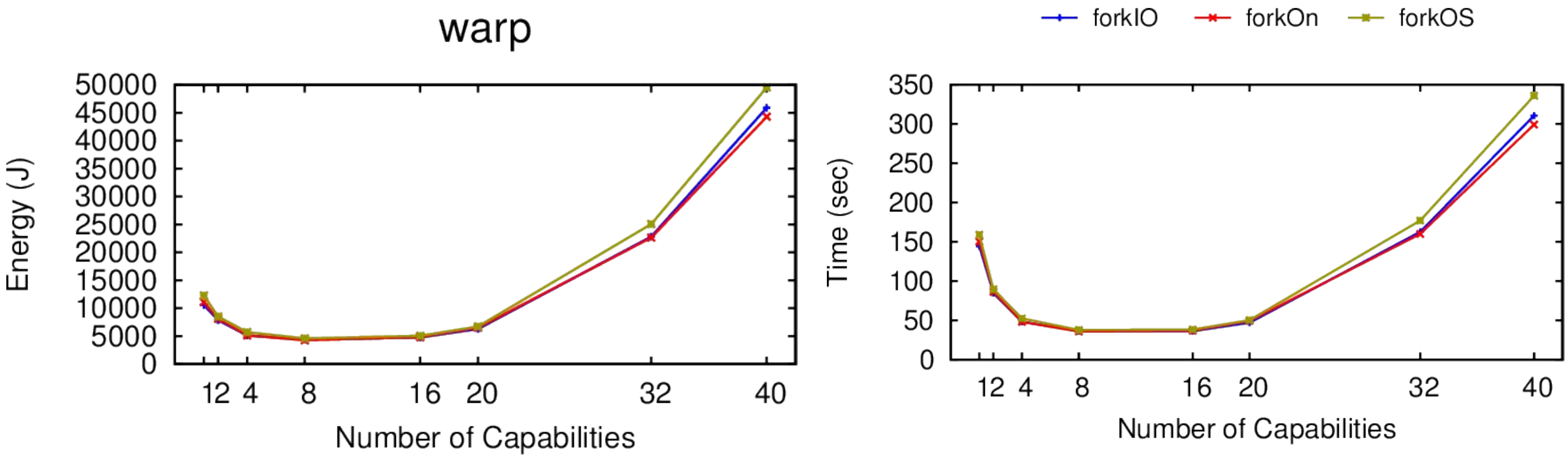
2x10-core Intel Xeon E5-2660 v2 processors + 256GB DDR3

Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25)

Criterion 1.1.0 with energy extension

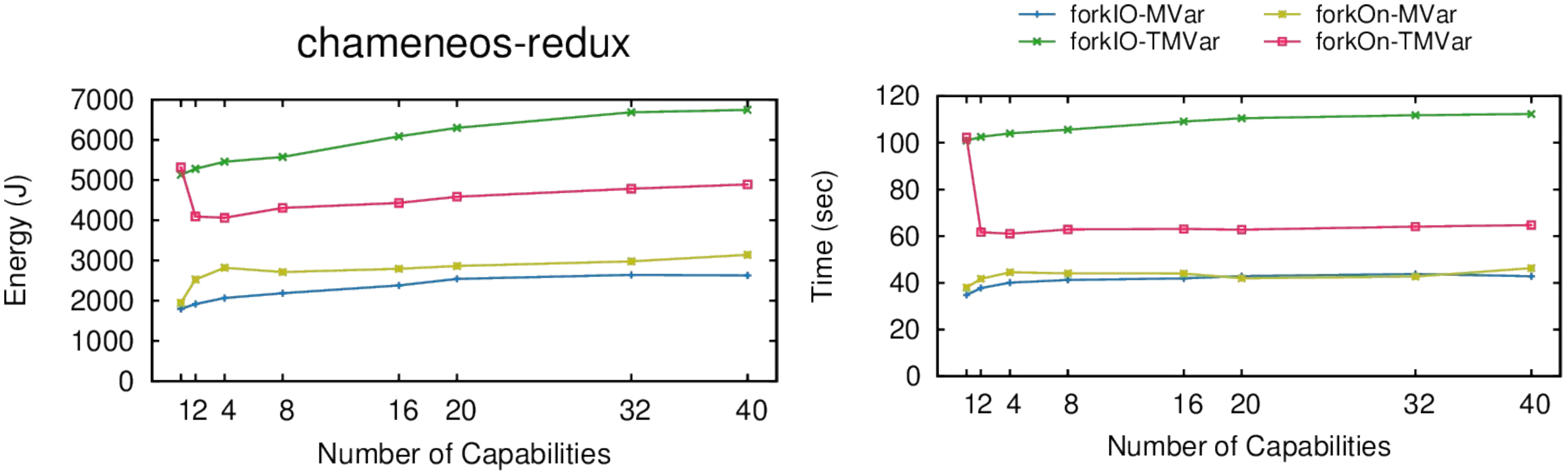
GHC 7.10.2

Results

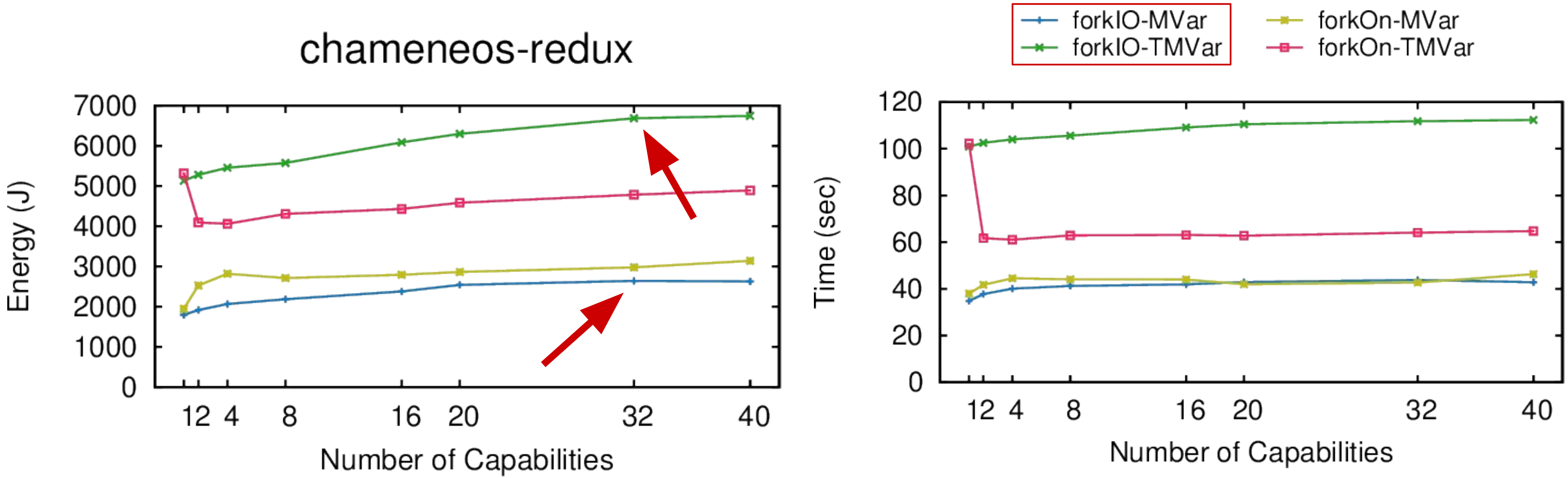


warp
forkIO
forkOn
forkOS
Energy (J)
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
12 4 8 16 20 32 40
Number of Capabilities
Time (sec)
350
300
250
200
150
100
50
0
12 4 8 16 20 32 40
Number of Capabilities

Small Changes Can Produce Big Savings

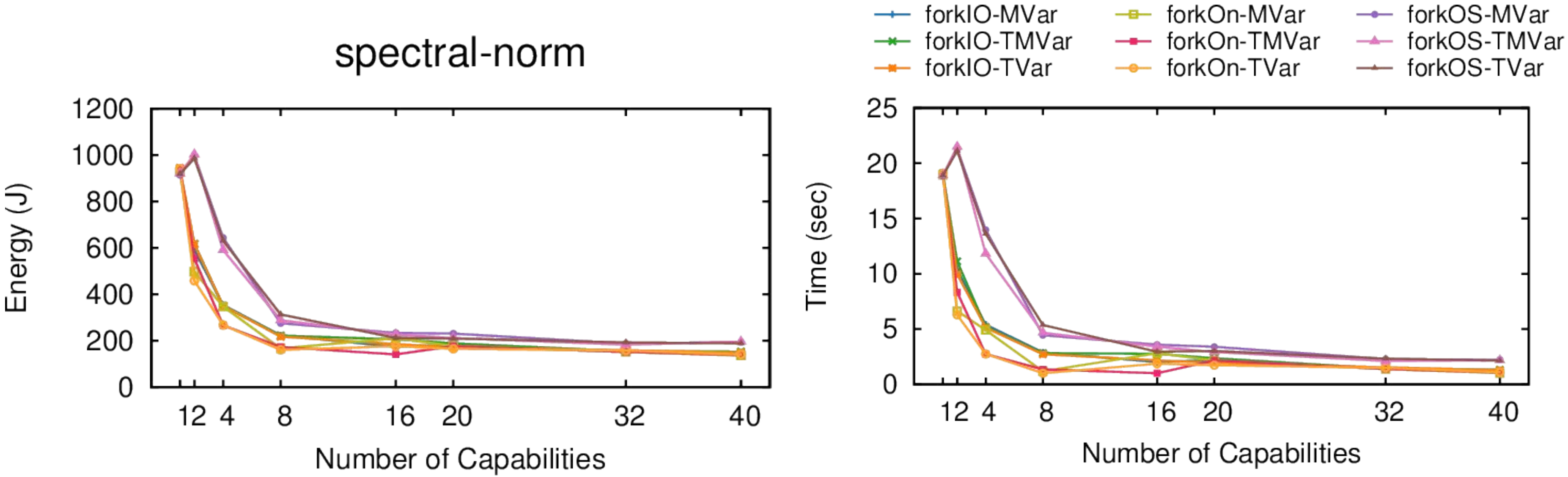


Small Changes Can Produce Big Savings

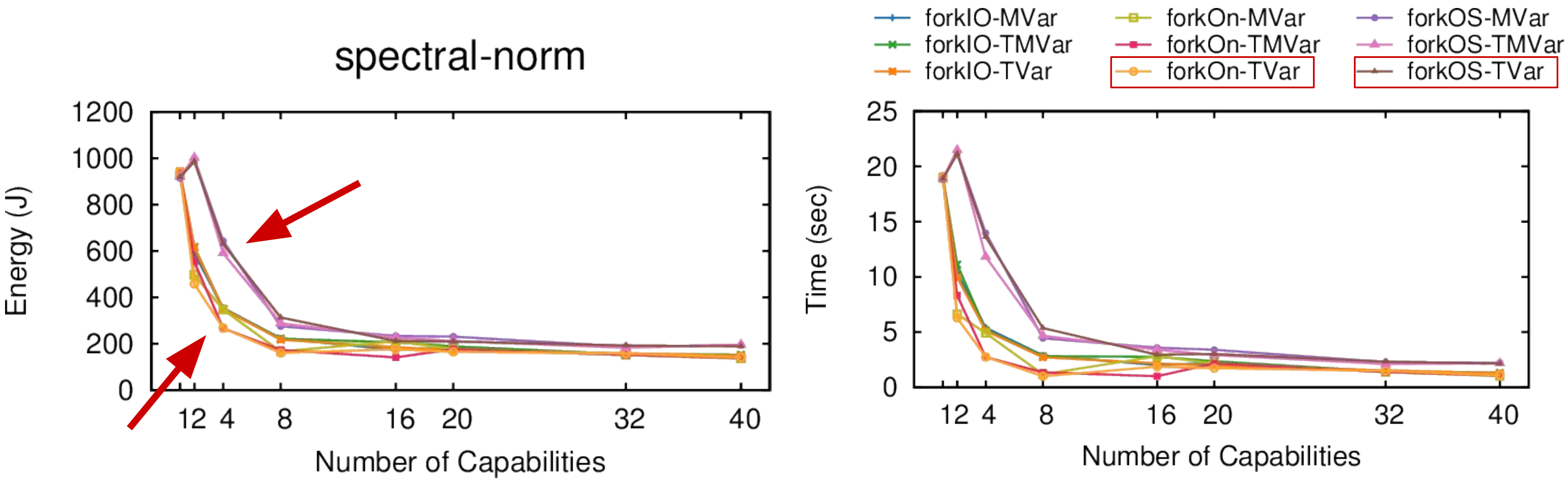


`TMVar` is 2.5x worse than `MVar`

Small Changes Can Produce Big Savings



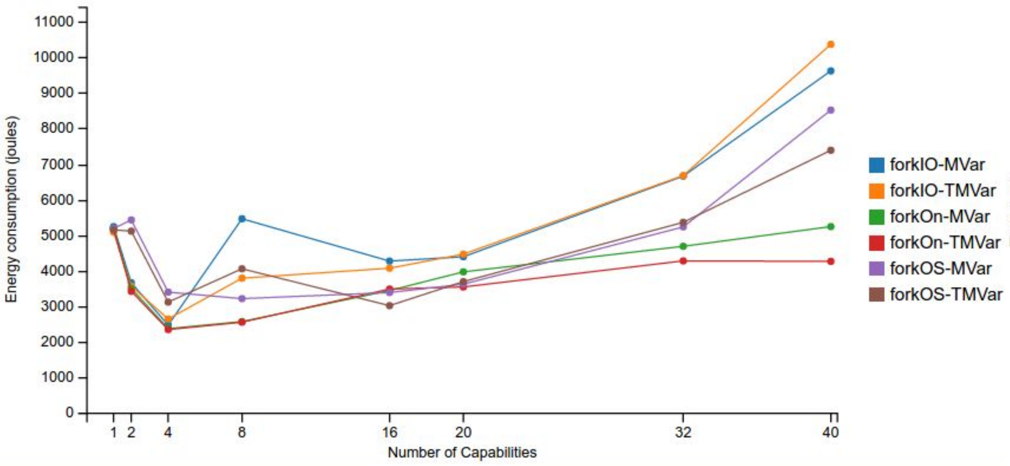
Small Changes Can Produce Big Savings

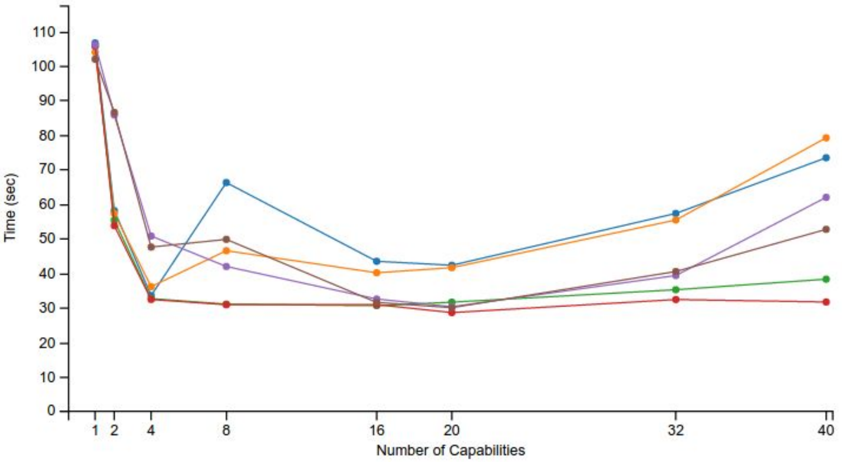


forkOS is 2.3x worse than forkOn

Faster is Not Always Greener

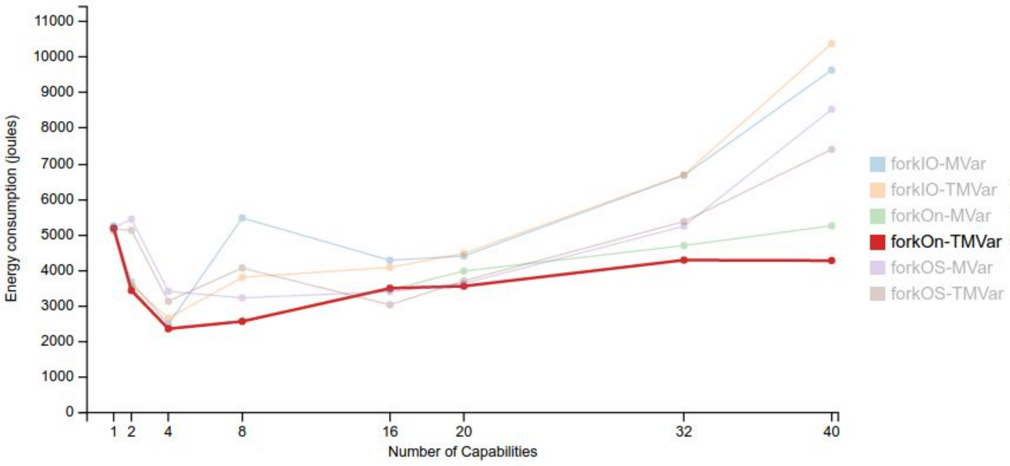
regex-dna

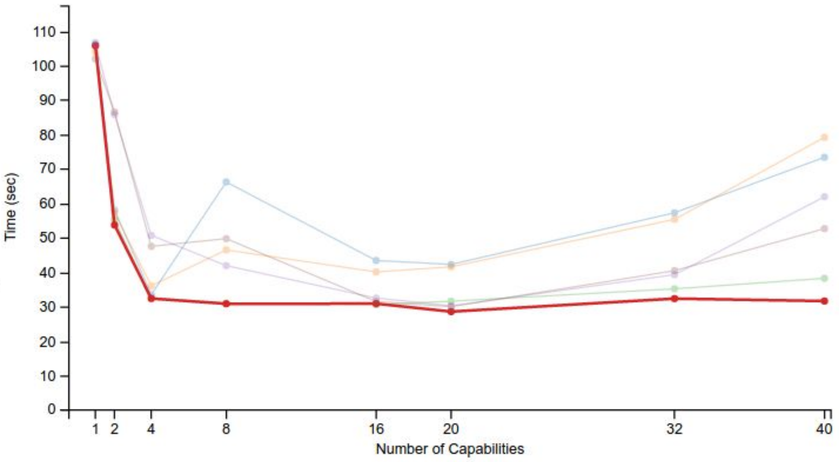




Faster is Not Always Greener

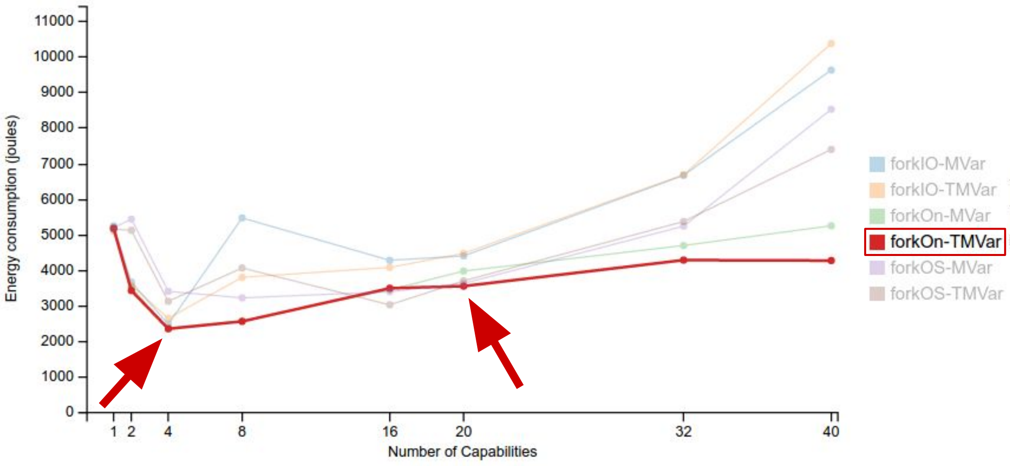
regex-dna

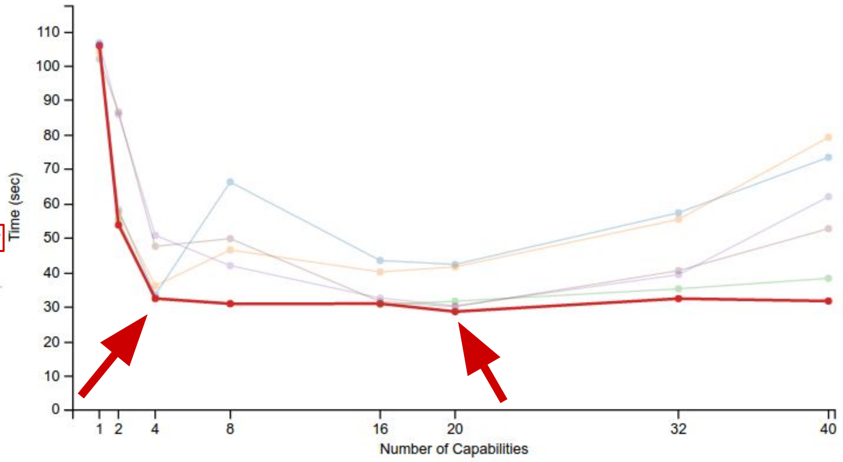




Faster is Not Always Greener

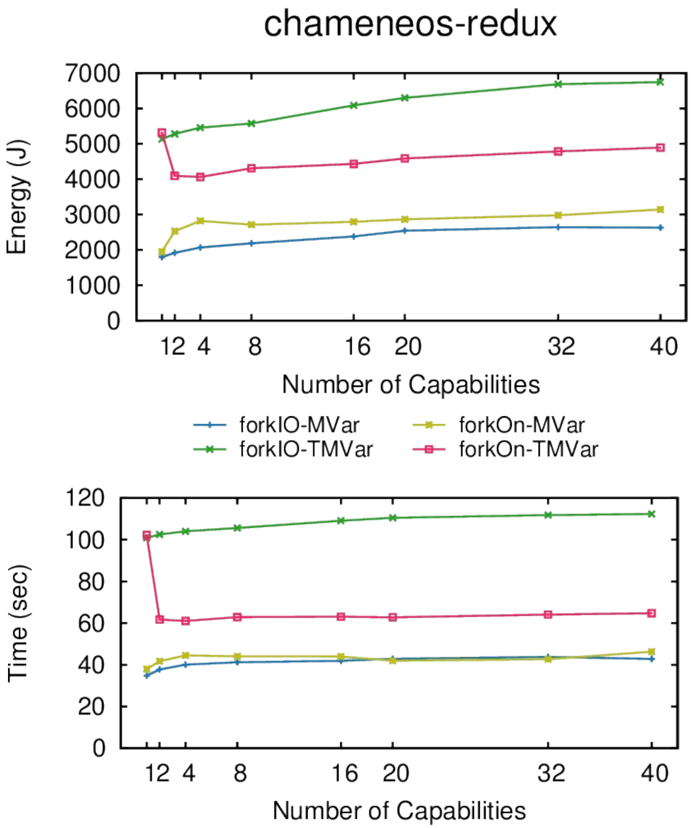
regex-dna

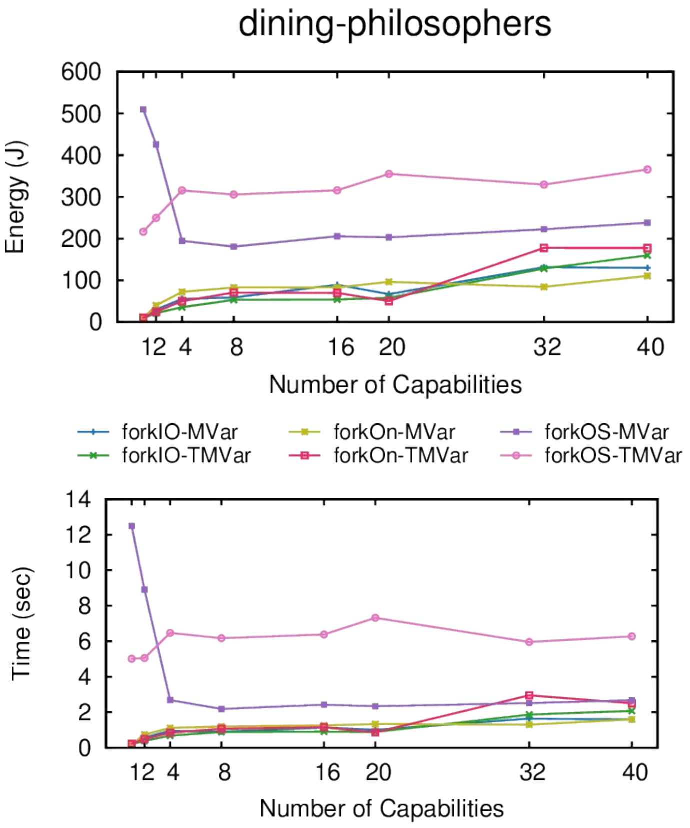




12% faster and **51% less energy-efficient**

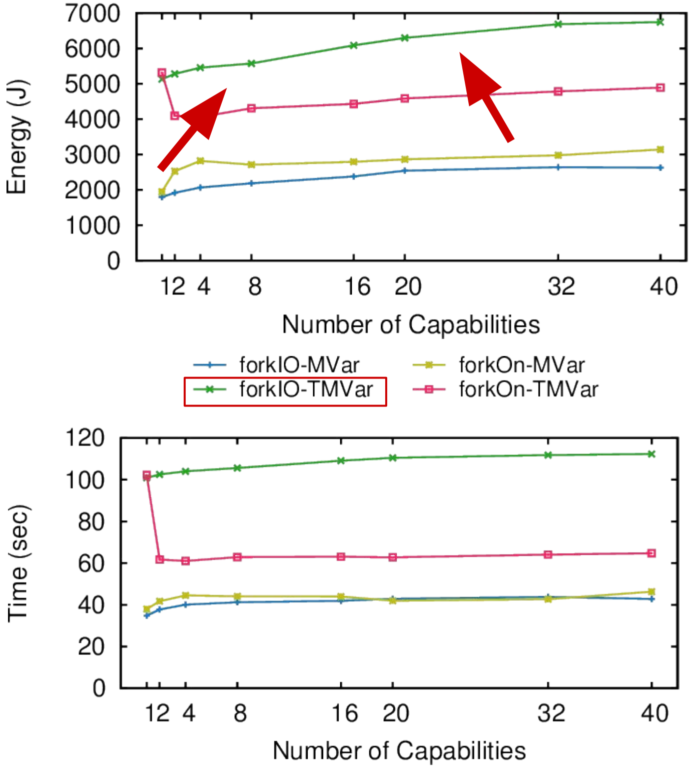
There is No Overall Winner



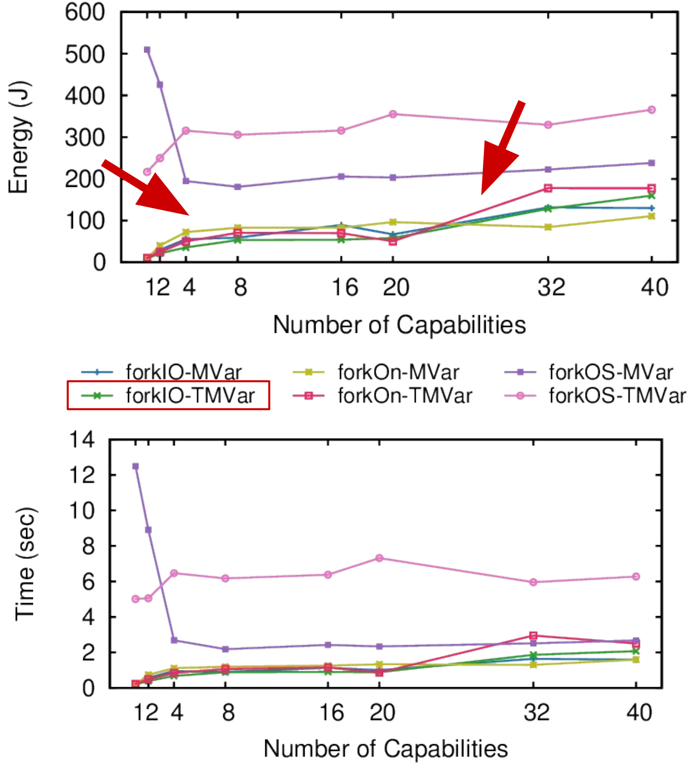


There is No Overall Winner

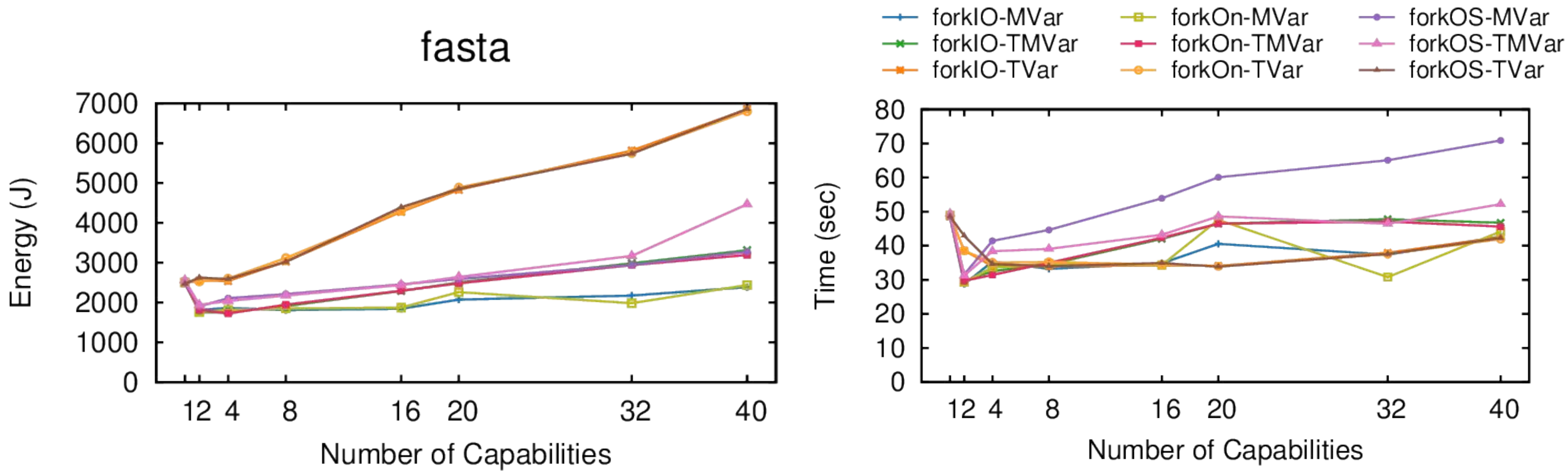
chameneos-redux



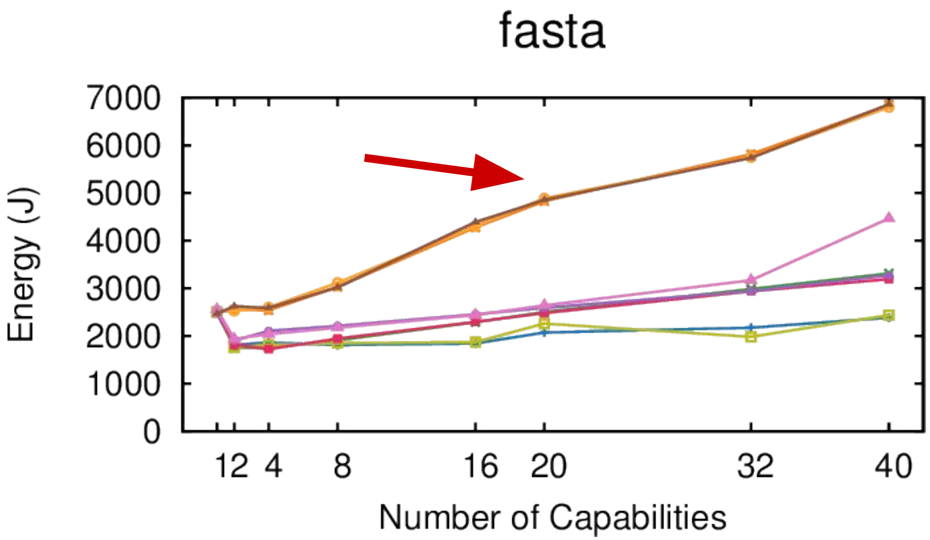
dining-philosophers

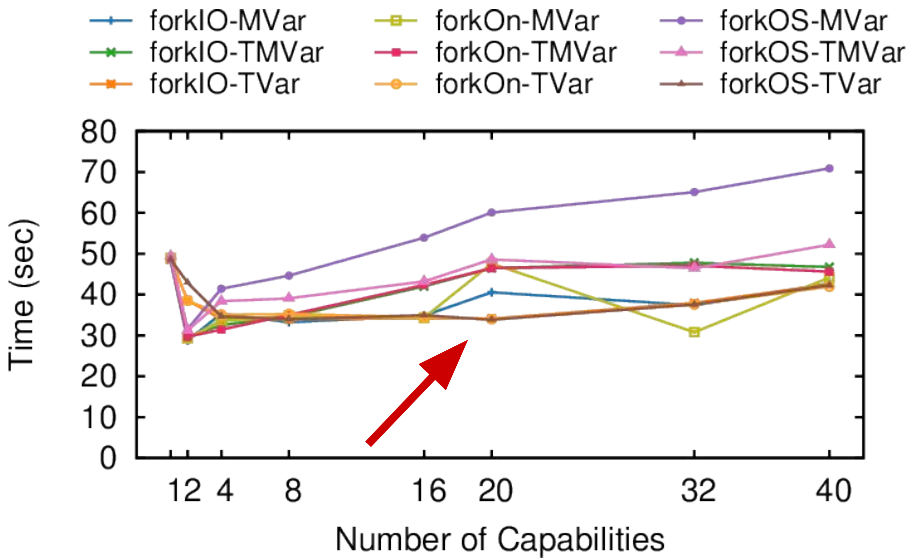


Case Study: `fasta`

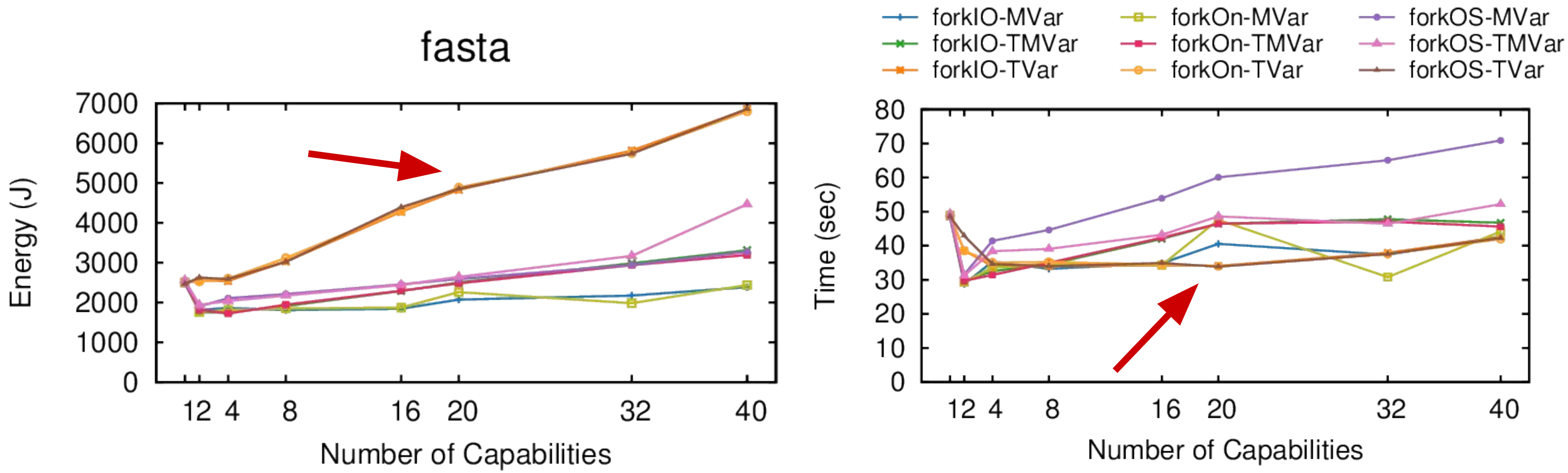


Case Study: `fasta`



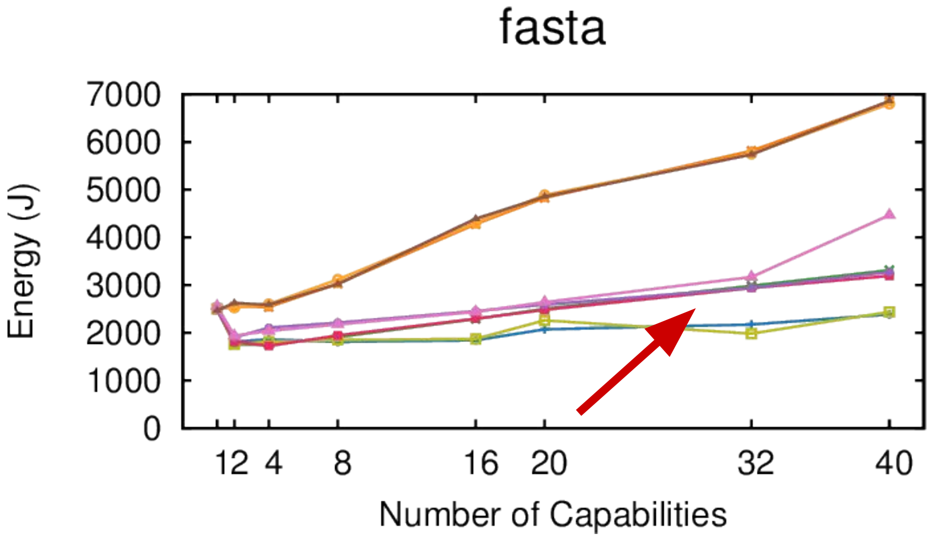


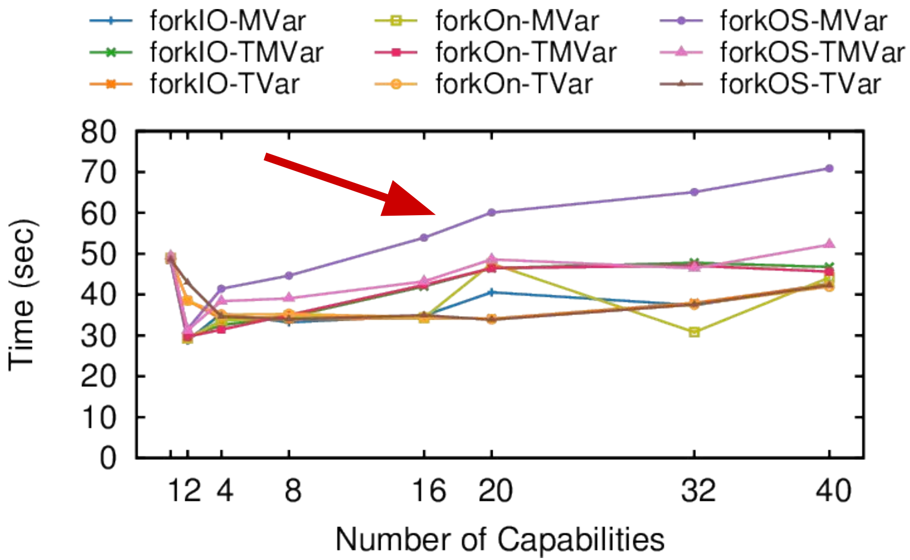
Case Study: `fasta`



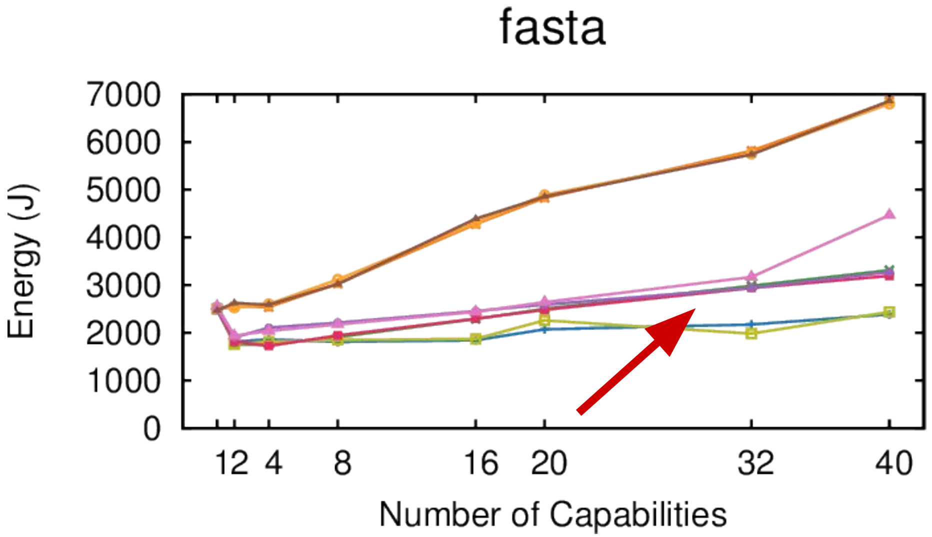
Best performance and worst energy consumption!

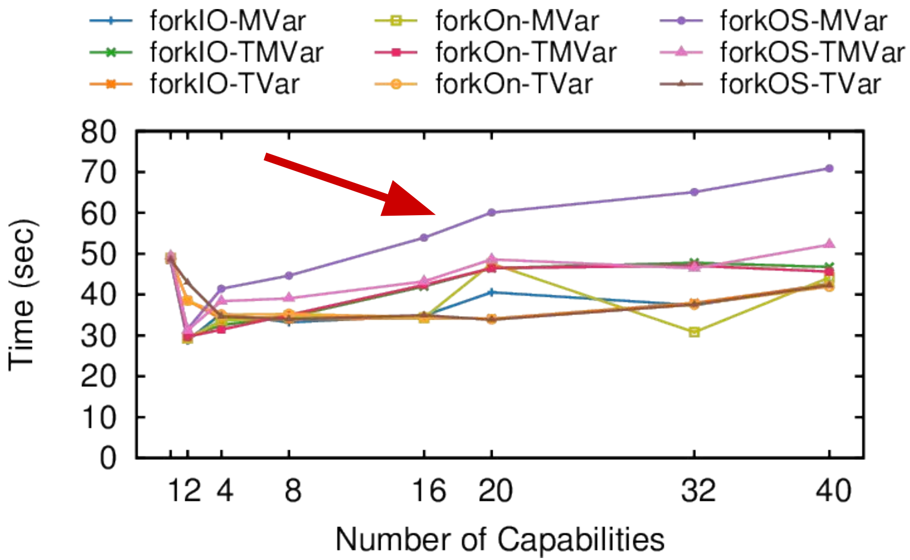
Case Study: `fasta`





Case Study: `fasta`



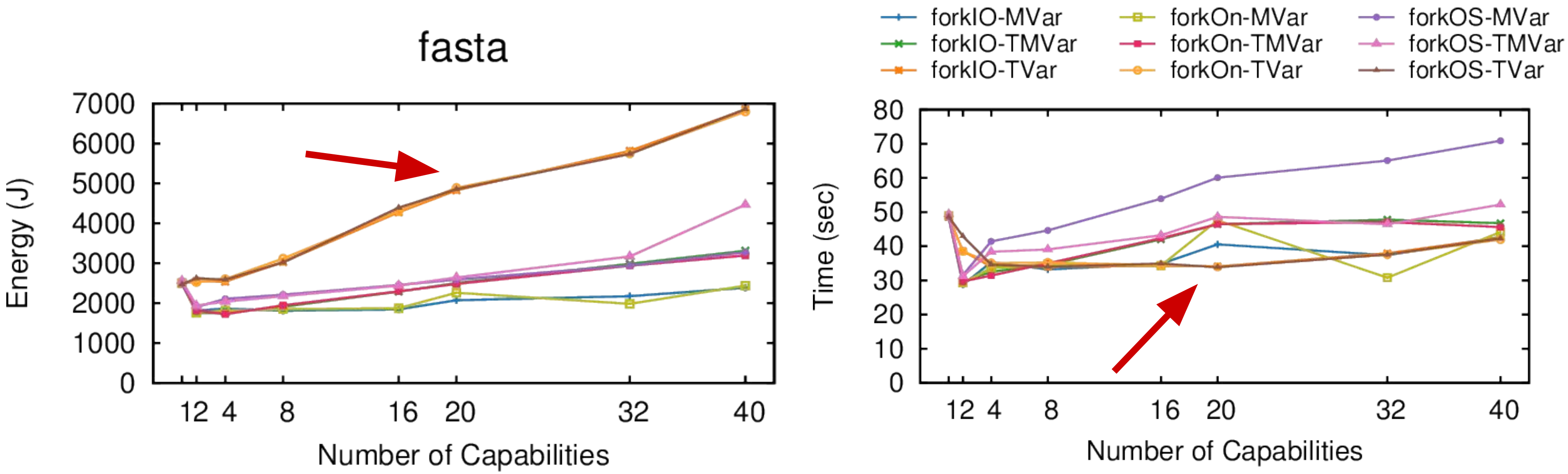


Worst performance and average energy consumption

How It Works

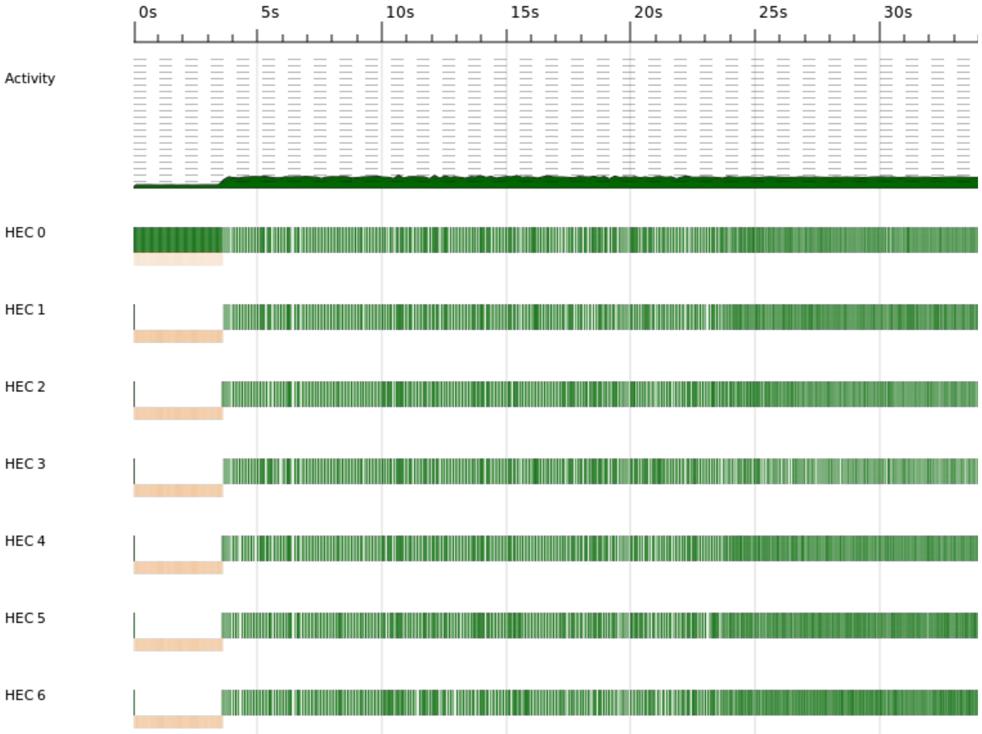
1. Take `seed0` from the shared variable
2. Generate `random_numbers` and `seed1`
3. Put `seed1` on the shared variable
4. Compute the DNA sequence based on `random_numbers`
5. Wait until the predecessor DNA sequence is written to output
6. Write DNA sequence to output

The Fastests Consume More Energy (1/4)

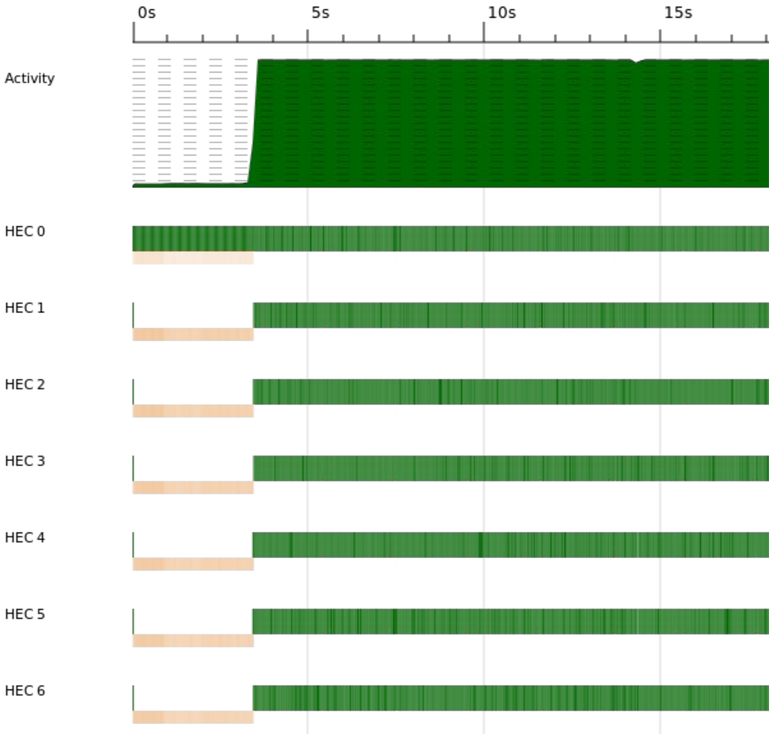


Best performance and worst energy consumption!

The Fastests Consume More Energy (2/4)



forkIO-MVar



forkIO-TVar

The Fastests Consume More Energy (3/4)

1. Take `seed0` from the shared variable
2. Generate `random_numbers` and `seed1`
3. Put `seed1` on the shared variable

The Fastests Consume More Energy (3/4)

1. Take `seed0` from the shared variable
2. Generate `random_numbers` and `seed1`
3. Put `seed1` on the shared variable

→ **MVar** vs **TVar**

The Fastests Consume More Energy (3/4)

1. Take `seed0` from the shared variable
2. Generate `random_numbers` and `seed1`
3. Put `seed1` on the shared variable

- Using **MVar** makes the program almost sequential;
- With **TVar**, all threads are competing to generate the same number;
- Multiple transaction abortions cause high CPU activity.

The Fastests Consume More Energy (4/4)

STM transaction statistics (2016-07-20 19:16:02.445387 UTC):

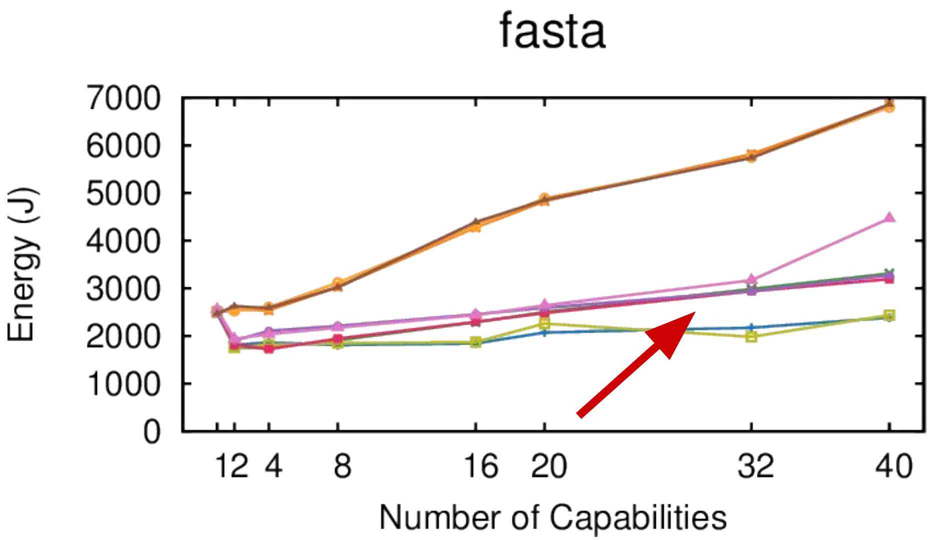
Transaction	Commits	Retries	Ratio
generate-numbers	299	4138	13.84
output-sync	261	33	0.13
wait-semaphore	2	2	1.00

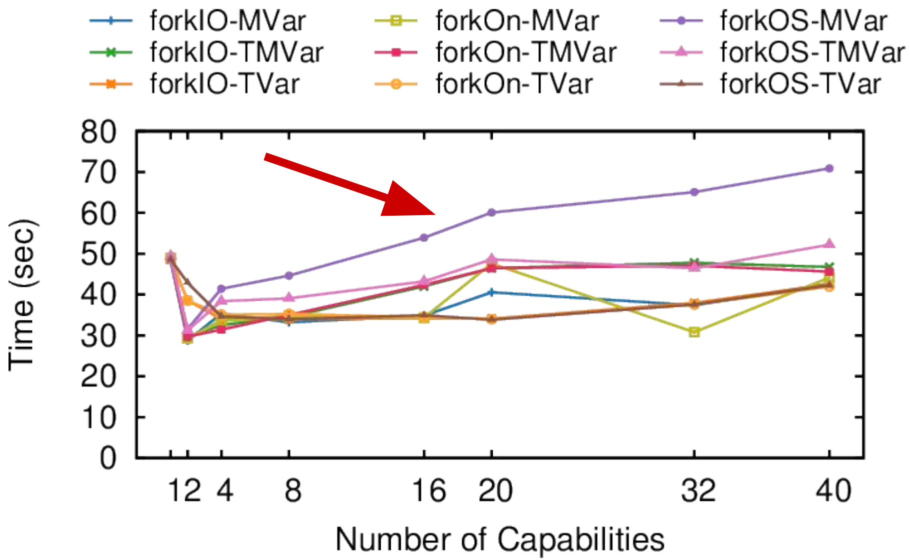
The Fastests Consume More Energy (4/4)

STM transaction statistics (2016-07-20 19:16:02.445387 UTC):

Transaction	**Commits**	**Retries**	**Ratio**
generate-numbers	299	4138	13.84
output-sync	261	33	0.13
wait-semaphore	2	2	1.00

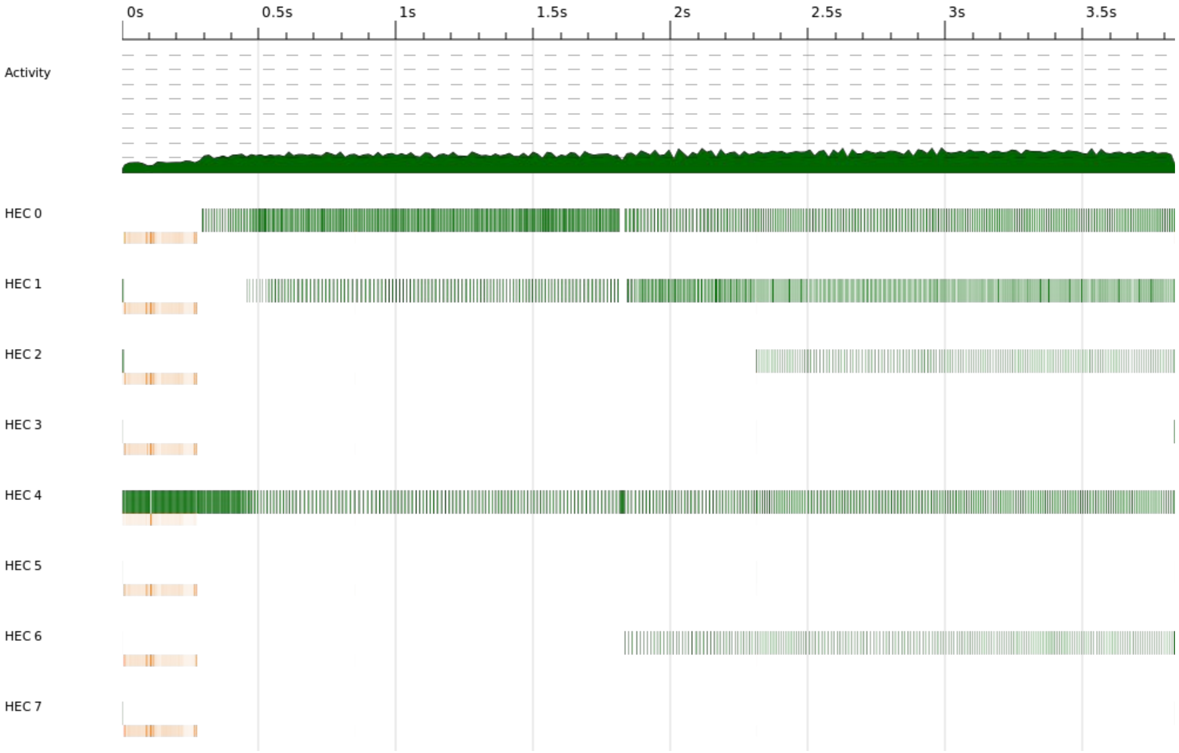
The Slowest Consumes Less Energy (1/2)





Worst performance and average energy consumption

The Slowest Consumes Less Energy (2/2)



forkOS-MVar

A Bug in the Scheduler

#12419 merge bug

Aberto 3 semanas atrás atrás
Modificado por último 8 dias atrás atrás

Scheduling bug with forkOS + MVar

Relatado por:	luisgabriel	Responsável:	
Prioridade:	normal	Marco:	8.0.2
Componente:	Runtime System	Versão:	8.0.1
Palavras-Chave:	forkOS; scheduler	Cc:	simonmar
Operating System:	Linux	Architecture:	x86_64 (amd64)
Type of failure:	None/Unknown	Test Case:	
Blocked By:		Blocking:	
Related Tickets:		Differential Rev(s):	Phab:D2430, Phab:D2441
Wiki Page:			

Descrição

I have noticed a weird scheduling behavior when performing some experiments with the **fasta** benchmark [1] from The Computer Language Benchmarks Game. When I switch `forkIO` by `forkOS` the scheduler stops to assign work for some capabilities, and they stay idle for the whole execution of the program.

ThreadScope view using forkIO: https://s31.postimg.org/r3mclspe3/fork_IO_N8_ghc8.png

ThreadScope view using forkOS: https://s31.postimg.org/p9n265fff/fork_OS_N8_ghc8.png

I was able to reproduce this behavior in both **GHC 7.10.2** and **GHC 8.0.2**. I was also able to reproduce it on two different machines running Ubuntu Server 14.04.3 LTS (kernel 3.19.0-25):

- 2x10-core Intel Xeon E5-2660 v2 processors (Ivy Bridge), 2.20 GHz, with 256GB of DDR 1600MHz
- 4-core Intel i7-3770 (IvyBridge) with 8 GB of DDR 1600MHz

Source code + .eventlog files: https://dl.dropboxusercontent.com/u/5798150/fasta-bug.zip

[1] http://benchmarksgame.alioth.debian.org/u64q/program.php?test=fasta&lang=ghc&id=7

Track the lengths of the thread queues Browse files

```
Summary:
Knowing the length of the run queue in O(1) time is useful: for example
we don't have to traverse the run queue to know how many threads we have
to migrate in schedulePushWork().

Test Plan: validate

Reviewers: e
Subscribers:
Differential
```

master

simonmar

Showing 5 c

Fix to thread migration Browse files

```
Summary:
If we had 2 threads on the run queue, say [A,B], and B is bound to the
current Task, then we would fail to migrate any threads.  This fixes it
so that we would migrate A in that case.

This will help parallelism a bit in programs that have lots of bound
threads.

Test Plan:
Test program
does behave

Reviewers: e
Subscribers:
Differential
GHC Trac Iss
```

master

simonmar

Showing 1 c

Another try to get thread migration right Browse files

```
Summary:
This is surprisingly tricky.  There were linked list bugs in the
previous version (D2430) that showed up as a test failure in
setnumcapabilities001 (that's a great stress test!).

This new version uses a different strategy that doesn't suffer from
the problem that @ezyang pointed out in D2430.  We now pre-calculate
how many threads to keep for this capability, and then migrate any
surplus threads off the front of the queue, taking care to account for
threads that can't be migrated.

Test Plan:
1. setnumcapabilities001 stress test with sanity checking (+RTS -DS) turned on:

...
cd testsuite/tests/concurrent/should_run
make TEST=setnumcapabilities001 WAY=threaded1 EXTRA_HC_OPTS=-with-rtsopts=-DS CLEANUP=0
while true; do ./setnumcapabilities001.run/setnumcapabilities001 4 9 2000 || break; done
...

2. The test case from #12419

Reviewers: niteria, ezyang, rwbarton, austin, bgamari, erikd

Subscribers: thomie, ezyang

Differential Revision: https://phabricator.haskell.org/D2441

GHC Trac Issues: #12419
```

master

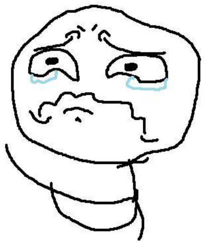
simonmar committed 9 days ago — 1 parent ce13a9a — commit 89fa4e968f47cfb42d0dc33fc3bfffdce31d850e

Showing 1 changed file with **62 additions** and **99 deletions**. Unified Split

Discussion

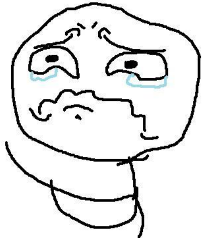
Bad news:

- The relationship between performance and energy is not obvious;



Discussion

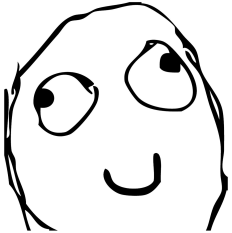
Bad news:



- The relationship between performance and energy is not obvious;

Good news:

- In most cases, switching between concurrency primitives is very simple;
- For most benchmarks, there is a configuration that most of the time beats the others;
- It's easy (and cheap) to experiment with different settings.



Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program;

2. Characterize the energy behavior of Haskell's concurrent programming constructs;

3. Provide guidelines for developers on how to write energy-efficient code.

Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program;

2. Characterize the energy behavior of Haskell's concurrent programming constructs;

3. <u>Provide guidelines for developers on how to write energy-efficient code</u>.

Use forkOn for embarrassingly parallel problems

Scenario:

- Your program creates multiple threads;
- There is **little or no dependency** among these threads;
- They perform **almost the same** amount of work.

Use forkOn for embarrassingly parallel problems

Scenario:

- Your program creates multiple threads;
- There is **little or no dependency** among these threads;
- They perform **almost the same** amount of work.

Solution:

- Use `forkOn` to spawn the threads;
- Distribute the threads **evenly** among the capabilities.

Use forkOn for embarrassingly parallel problems

Scenario:

- Your program creates multiple threads;
- There is **little or no dependency** among these threads;
- They perform **almost the same** amount of work.

Solution:

- Use `forkOn` to spawn the threads; → Reduces the scheduling overhead
- Distribute the threads **evenly** among the capabilities.

Use forkOn for embarrassingly parallel problems

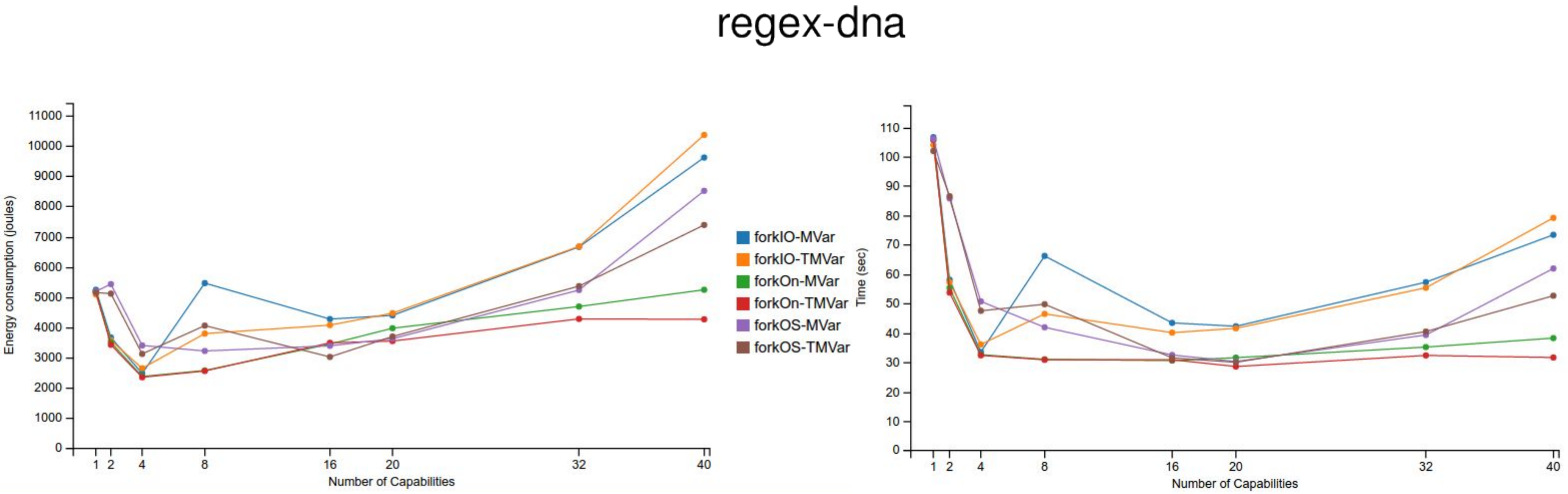
Scenario:

- Your program creates multiple threads;
- There is **little or no dependency** among these threads;
- They perform **almost the same** amount of work.

Solution:

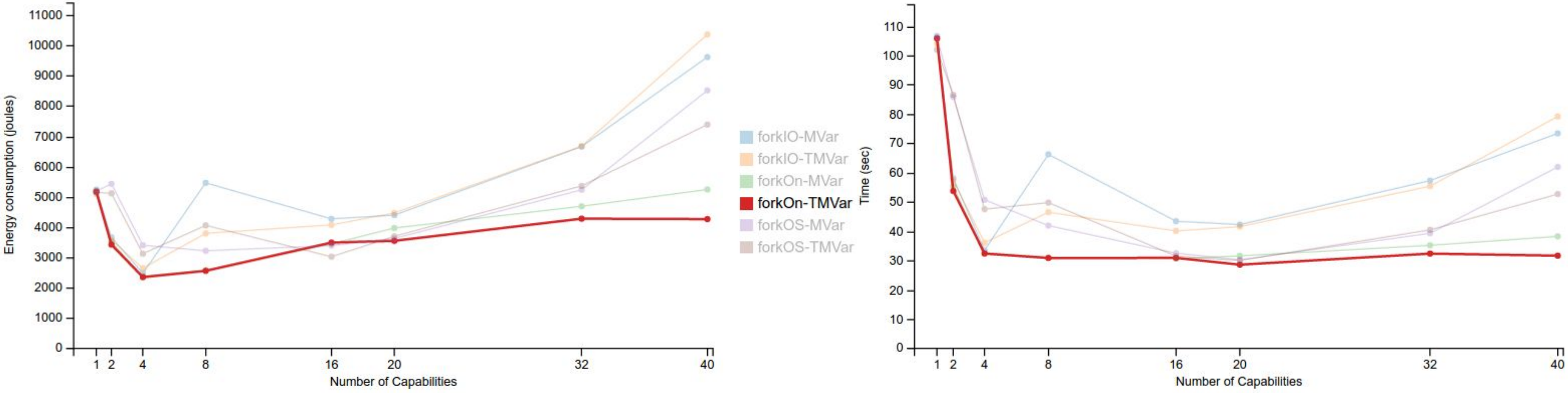
- Use `forkOn` to spawn the threads; → Reduces the scheduling overhead
- Distribute the threads **evenly** among the capabilities. → Improves performance

Use forkOn for embarrassingly parallel problems



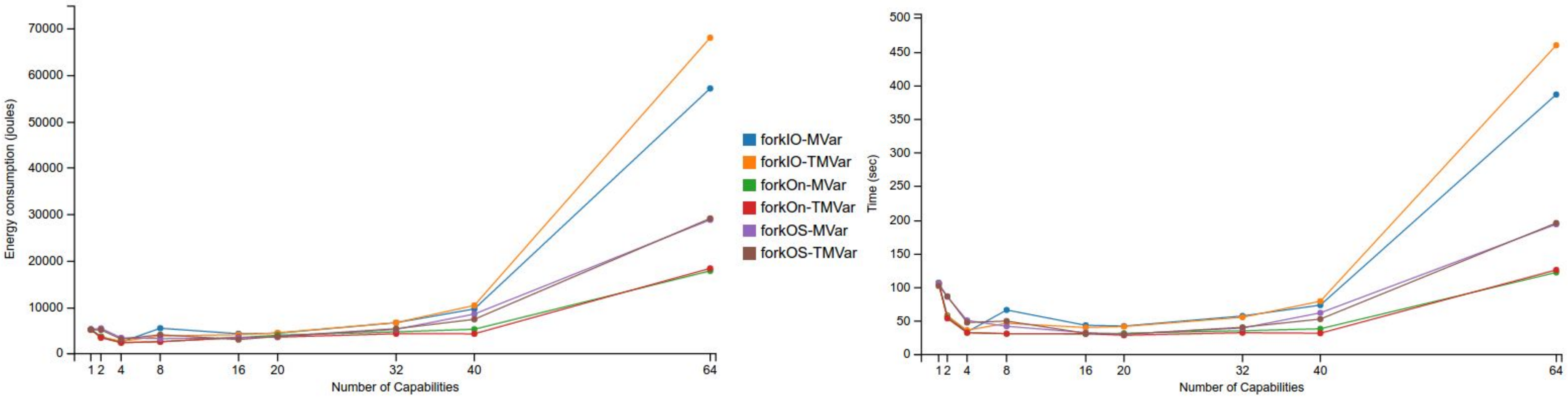
Use forkOn for embarrassingly parallel problems

regex-dna

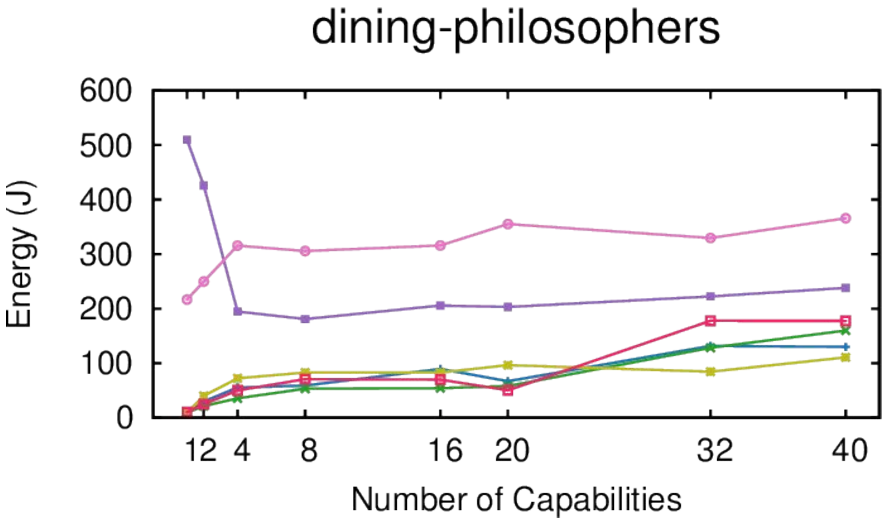


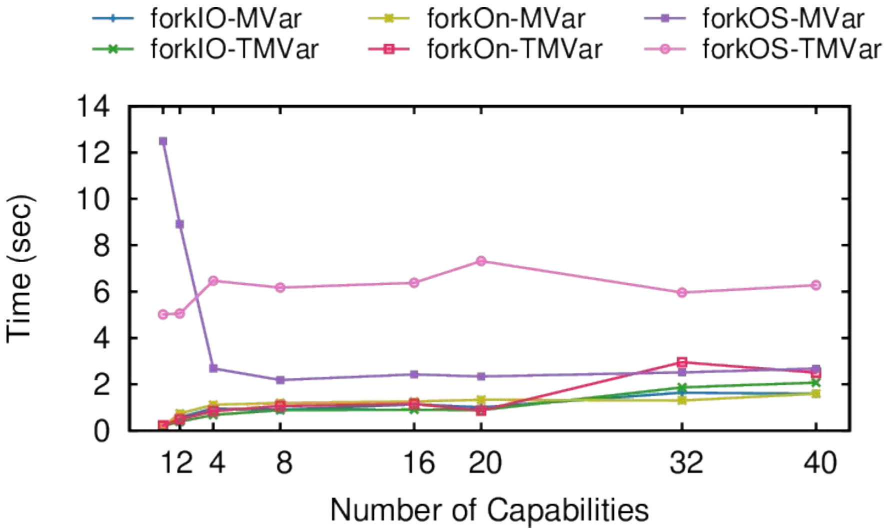
Avoid setting more capabilities than available CPUs

regex-dna

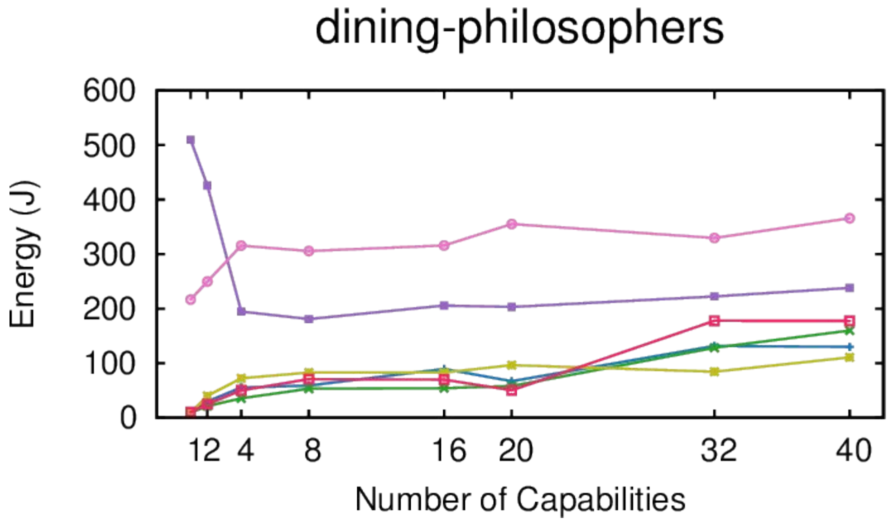


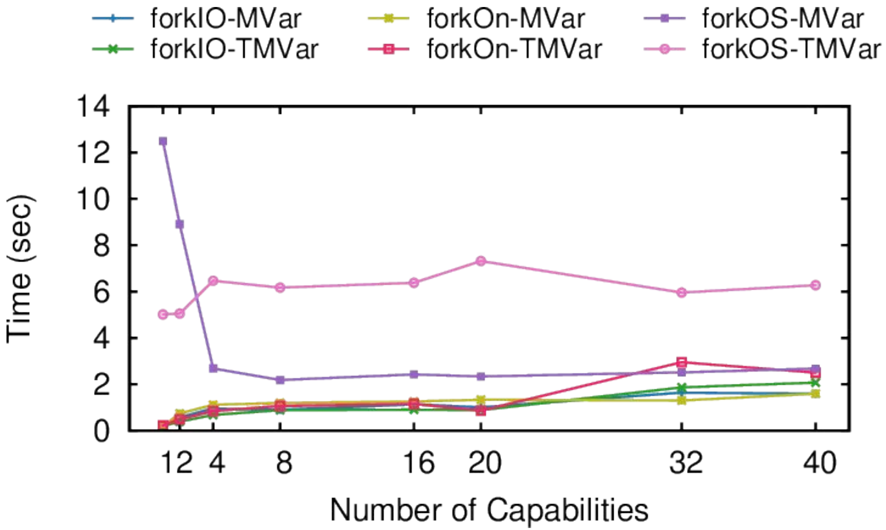
Avoid using forkOS, except when you can't





Avoid using forkOS, <u>except when you can’t</u>





Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program; ✓

2. Characterize the energy behavior of Haskell's concurrent programming constructs; ✓

3. Provide guidelines for developers on how to write energy-efficient code. ✓

Contributions

- A tool for fine-grained energy analysis;
- A tool for coarse-grained energy analysis;
- An understanding of the energy behavior of concurrent Haskell programs;
- A list of guidelines on how to write energy-efficient software;

Contributions

- A tool for fine-grained energy analysis;
- A tool for coarse-grained energy analysis;
- An understanding of the energy behavior of concurrent Haskell programs;
- A list of guidelines on how to write energy-efficient software;
- A paper published at the main research track of SANER'16.

Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language

Future Work

- Develop a software model for estimating the energy consumed by core;
- Adapt the GHC energy profiler to handle parallel execution;
- Extend ThreadScope to support energy consumption;
- Replicate our study on different hardware (Haswell and Broadwell);
- Study how the various GHC options impact energy consumption;
- In-depth analysis of each benchmark of our suite;
- Analyse other concurrent programming models (e.g. Actor Model).

Understanding the Energy Behavior of Concurrent Haskell Programs

Goals

1. Enable developers to effectively measure the energy consumption of a Haskell program; ✓
2. Characterize the energy behavior of Haskell's concurrent programming constructs; ✓
3. Provide guidelines for developers on how to write energy-efficient code. ✓

70

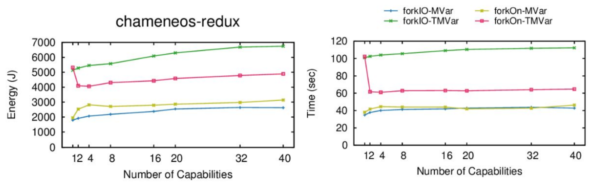
Performance Analysis in Haskell

Profiling → GHC Profiler

Benchmarking → Criterion

17

Small Changes Can Produce Big Savings



40

Use forkOn for embarrassingly parallel problems

Scenario:

- Your program creates multiple threads;
- There is little or no dependency among these threads;
- They perform almost the same amount of work.

Solution:

- Use `forkOn` to spawn the threads; → Reduces the scheduling overhead
- Distribute the threads evenly among the capabilities. → Improves performance

65

Luís Gabriel Lima lgnfl@cin.ufpe.br

Fernando Castor castor@cin.ufpe.br
(advisor)

João Paulo Fernandes jpf@di.ubi.pt
(co-advisor)

http://green-haskell.github.io/

Thank you!

VIRTUS IMPAVIDA UFPE

Centro de Informática UFPE