Programación Paralela en Haskell

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

1. Introducción al Paralelismo
2. Paralelismo básico en Haskell
3. Ejemplo map: sudoku
4. Ejemplo reduce
5. Otras formas de paralelismo
6. Bibliografía

Recursos de computación

Actualmente se puede encontrar recursos paralelos en:

- **Procesadores actuales** incluyen múltiples núcleos (del orden de decenas) interconectados con una memoria principal (compartida).
- **Tarjetas gráficas** se pueden emplear como procesadores masivamente paralelos con miles de núcleos, compartiendo una memoria gráfica (separada de la del sistema).
- **Clústers** de ordenadores, interconectados por una red rápida con una memoria distribuida. **Supercomputadores**.
- **Chips** dedicados (FPGAs).

La tendencia es aprovechar estos recursos en los proyectos software y científicos de computación.

Paralelismo vs concurrencia

- **Concurrencia**: uso de *múltiples hilos* para modularidad de interacción. Puede realizarse sin ejecución simultánea de código.

Paralelismo vs concurrencia

- **Concurrencia**: uso de *múltiples hilos* para modularidad de interacción. Puede realizarse sin ejecución simultánea de código.

- **Paralelismo**: uso de *múltiples procesadores* para mayor rendimiento. Necesita ejecución simultánea de código.

Paralelismo vs concurrencia

- **Concurrencia**: uso de *múltiples hilos* para modularidad de interacción. Puede realizarse sin ejecución simultánea de código.

- **Paralelismo**: uso de *múltiples procesadores* para mayor rendimiento. Necesita ejecución simultánea de código.

  Tipos de paralelismo:
  - Paralelismo de tareas
  - Paralelismo de datos

Paralelismo de Tareas

Paralelismo de Datos

Paralelismo programación funcional

- Paralelismo en **programación funcional**:
  - *Determinista*: el programa siempre produce la misma respuesta, pero puede ejecutarse más rápido conforme hayan más núcleos.
  - Sin condiciones de carrera *ni interbloqueo*.
  - Añadir paralelismo sin sacrificar *corrección*.
  - Paralelismo para acelerar código *Haskell puro* (no IO).

Patrones de paralelismo

- Veremos dos patrones básicos de paralelismo y como se pueden implementar en Haskell mediante la mónada eval: map y reduce.
- **Map**:
  - Aplicar una misma función (`f :: a -> b`) a los elementos de una *colección* (por ejemplo, una lista (`xs :: [a]`)), obteniendo una colección actualizada (`(xs :: [b])`). Por ejemplo:

```
xs = [1..10^6]
cuadrados = map (^2) xs
```

  - Lanzar un hilo para evaluar la función a cada elemento en paralelo, y sincronizar los hilos al final.

Patrones de paralelismo

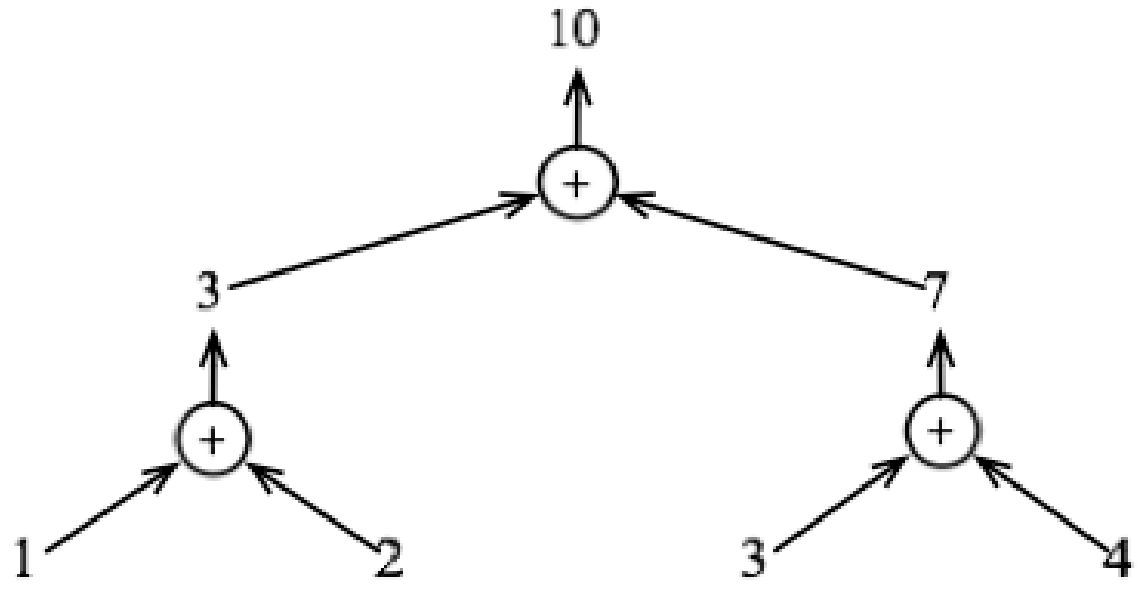
- **Reduce**:
  - Aplicar una función `f :: a -> a -> a` a los elementos de una *colección* (por ejemplo, una lista `xs :: [a]`) para obtener un valor único valor final resumido (`y :: a`). Por ejemplo:

```
xs = [1..10^6]
factorialMillon = product xs
```

  - ¿Cómo podemos hacer esto en paralelo?

Patrones de paralelismo

- **Reduce**:
  - ¿Cómo podemos hacer esto en paralelo?
  - Lanzar un hilo para aplicar la función a cada par de elementos y sincronizar. Repetir el proceso con los resultados parciales hasta que solo quede un elemento, que será el resultado.




```
foldr   (fn(x,y)=>x+y)   0   [1, 2, 3, 4]
```

1 Introducción al Paralelismo

2 Paralelismo básico en Haskell

3 Ejemplo map: sudoku

4 Ejemplo reduce

5 Otras formas de paralelismo

6 Bibliografía

La mónada Eval

```haskell
data Eval a
instance Monad Eval

runEval :: Eval a -> a

rpar :: a -> Eval a
rseq :: a -> Eval a
```

Evaluación de funciones:

- **rpar**: evalúa su argumento en paralelo (sin esperar al resultado).
- **rseq**: evalúa su argumento en secuencial y espera al resultado.
- **runEval**: realiza la computación Eval y devuelve su resultado.

Instalación: `cabal install parallel`

La mónada Eval

Ejemplo 1

```
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a,b)
```

- Evalúa la función `(f x)` y `(f y)` en paralelo, y devuelve sus resultados en un par.
- **Problema**: `return (a,b)` se ejecutaría inmediatamente después de comenzar la evaluación, sin esperar a obtener `a` ni `b`.

- ¿Qué pasaría aquí?

```
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  return (a,b)
```

La mónada Eval

Ejemplo 2

```
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  rseq a
  rseq b
  return (a,b)
```

- Evalúa la función `(f x)` y `(f y)` en paralelo, y devuelve sus resultados en un par.
- Evaluamos `a` y `b` y esperamos su resultado, por lo que ahora sí, `return` se ejecutaría después de evaluar por completo `(f x)` y `(f y)`.

La mónada Eval

Evaluación profunda

```
runEval $ do
  a <- rpar (f xs)
  b <- rpar (f ys)
  rseq (a,b)
  return (a,b)
```

- La evaluación de `f` sobre `xs` e `ys` se realiza en forma normal de cabeza débil (WHNF) (es decir, hasta el primer ':' o cabeza de la lista).
- Necesitamos evaluar f sobre toda la lista (forma normal, NF). Para ello forzar evaluación de toda la lista con `force`, del módulo `Control.DeepSeq`.

```
runEval $ do
  a <- rpar (force (f xs))
  b <- rpar (force (f ys))
  rseq (a,b)
  return (a,b)
```

La mónada Eval

Evaluación profunda

```
force :: NFData a => a -> a
```

- La evaluación `force` evalúa todo su argumento y lo retorna. Su comportamiento se define para cada tipo de datos mediante la clase NFData. Habrá que crear instancia para un nuevo tipo de dato.

```
class NFData a where
  rnf :: a -> ()
  rnf a = a `seq` ()
```

- `rnf` (reduce to NF) evalua su argumento y devuelve return. `seq` se emplea para secuenciar evaluaciones. `deepseq` es `seq` en NF.

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b

force :: NFData a => a -> a
force x = x `deepseq` x
```

La mónada Eval

Sparks

- El argumento de rpar se denomina *spark*.
- El sistema en tiempo de ejecución colecciona todos los sparks y los ejecuta (o no) cuando hayan procesadores disponibles, mediante "work stealing".
- La **creación** de un spark no es costosa, simplemente añade una referencia a un vector (*pool*).
- Los sparks pueden ser *convertidos (converted)* cuando se ejecutan en paralelo, o *podados (pruned)* cuando se determinan que ya se han evaluado por otro spark, o no se llegan nunca a referenciar en el código. En concreto:
  - *converted*: convertido en paralelismo real.
  - *overflowed*: el vector (*pool*) de sparks está completo.
  - *dud*: rpar es aplicado a una expresión ya evaluada.
  - *GC'd*: la expresión no se utiliza en el programa, por lo que se elimina.
  - *fizzled*: la expresión se evaluó después de crearse su spark.

1 Introducción al Paralelismo

2 Paralelismo básico en Haskell

3 Ejemplo map: sudoku

4 Ejemplo reduce

5 Otras formas de paralelismo

6 Bibliografía

Ejemplo Sudoku

- Fichero donde se representa un tablero de sudoku por línea.
- Resolvedor de Sudoku con una poda inteligente.
- **Objetivo:** Resolver tableros en paralelo (con **map**).
- El código fuente se puede acceder en este repositorio git.
- Usaremos el compilador ghc y parámetros del sistema:
  - Compilación: *ghc -O2 fichero.hs -threaded -rtsopts*
    - -**O2**: optimización del código a nivel 2
    - -**threaded**: emplear hilos del sistema
    - -**rtsopts**: permitir que se muestre información de la ejecución.
  - Ejecución: *./fichero +RTS -NX -s*
    - +**RTS** -**NX**: hacer uso de X procesadores del sistema.
    - -**s**: mostrar información de la ejecución

Ejemplo Sudoku

Solución secuencial

Fichero *sudoku1.hs*:

```
import Sudoku
import Control.Exception
import System.Environment
import Data.Maybe

main :: IO ()
main = do
  [f] <- getArgs                        -- Ruta del fichero
  file <- readFile f                    -- Lectura de fichero

  let puzzles   = lines file            -- Tableros por línea
      solutions = map solve puzzles     -- Resolver cada tablero
  print (length (filter isJust solutions))  -- Cuantos con solución
```

Ejemplo Sudoku

Solución secuencial

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
$ ./sudoku1 sudoku17.1000.txt +RTS -s
2,362,862,096 bytes allocated in the heap
38,751,680 bytes copied during GC
214,112 bytes maximum residency (14 sample(s))
71,360 bytes maximum slop
2 MB total memory in use (0 MB lost due to fragmentation)

Tot time (elapsed)  Avg pause  Max pause
Gen  0      4574 colls,     0 par    0.12s    0.12s     0.0000s
     0.0003s
Gen  1        14 colls,     0 par    0.00s    0.00s     0.0003s
     0.0004s

TASKS: 3 (1 bound, 2 peak workers (2 total), using -N1)

SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    2.62s  (  2.62s elapsed)
GC      time    0.13s  (  0.13s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    2.75s  (  2.75s elapsed)

Alloc rate    902,441,432 bytes per MUT second
```

Ejemplo Sudoku

Solución paralela con partición estática

Partición de la lista de tableros en dos (estática). Fichero *sudoku2.hs*:

```
main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles = lines file
      (as,bs) = splitAt (length puzzles `div` 2) puzzles --

      solutions = runEval $ do
                    as' <- rpar (force (map solve as))
                    bs' <- rpar (force (map solve bs))
                    rseq as'
                    rseq bs'
                    return (as' ++ bs')

  print (length (filter isJust solutions))
```

Ejemplo Sudoku

Solución paralela con partición estática

```
$ ghc -O2 sudoku2.hs -rtsopts -threaded
...
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -s
2,370,828,176 bytes allocated in the heap
48,592,432 bytes copied during GC
2,613,568 bytes maximum residency (8 sample(s))
317,952 bytes maximum slop
9 MB total memory in use (0 MB lost due to fragmentation)

Tot time (elapsed)  Avg pause  Max pause
Gen  0      3008 colls,  3008 par    0.25s    0.13s     0.0000s
     0.0009s
Gen  1         8 colls,     7 par    0.04s    0.02s     0.0023s
     0.0052s

Parallel GC work balance: 49.47% (serial 0%, perfect 100%)

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

SPARKS: 2 (1 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    2.86s  (  1.74s elapsed)
GC      time    0.29s  (  0.14s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    3.16s  (  1.89s elapsed)

Alloc rate    827,641,496 bytes per MUT second
```

Ejemplo Sudoku

Solución paralela con partición estática

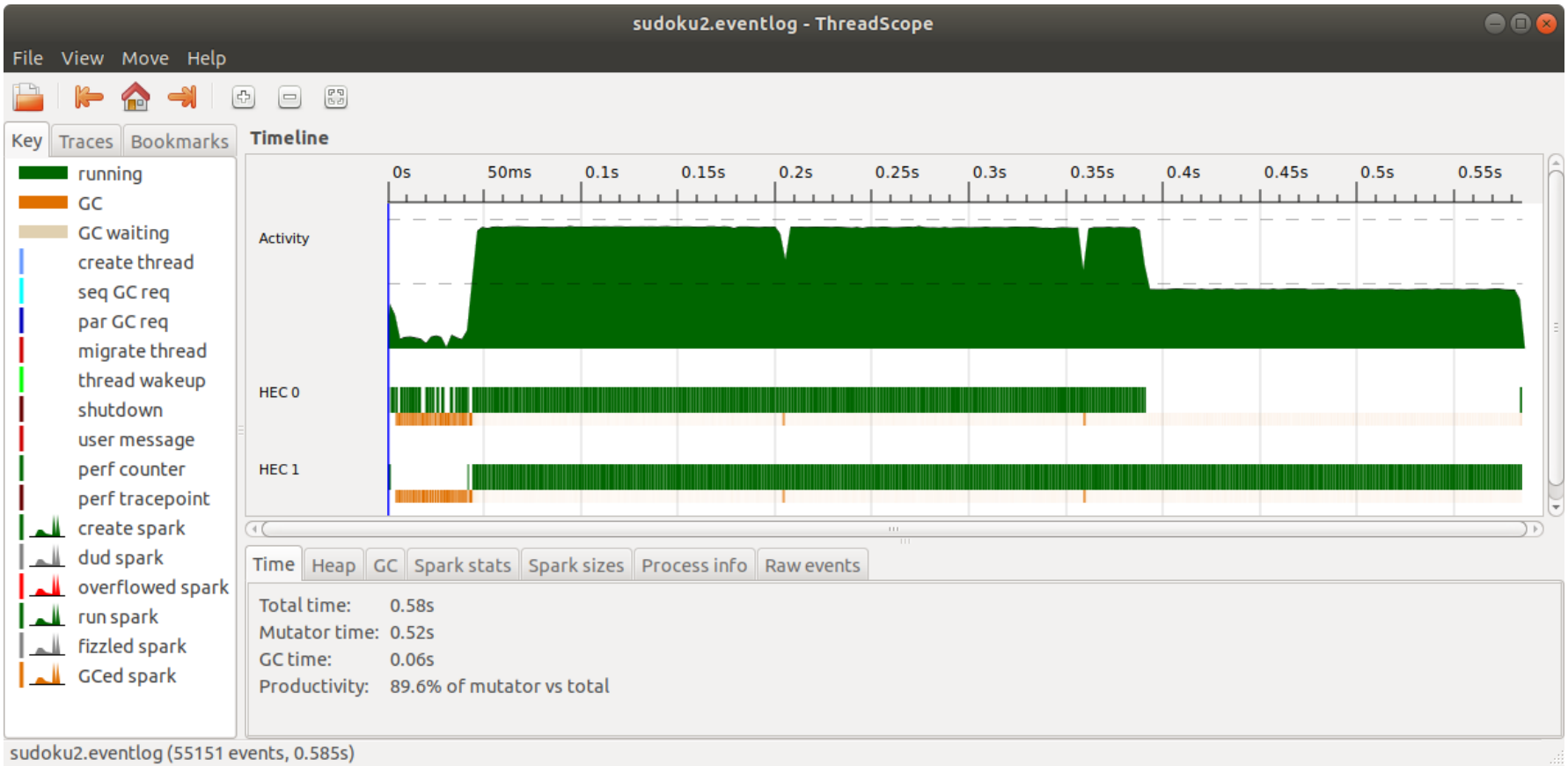
- Usando dos cores no alcanzamos doble de eficiencia (1,89s vs 2,75s).
- Veamos qué ocurre con la herramienta threadscope (siguiente diapositiva):

```
$ rm sudoku2; ghc -O2 sudoku2.hs -threaded -rtsopts -eventlog
[2 of 2] Compiling Main             ( sudoku2.hs, sudoku2.o )
Linking sudoku2 ...
$ ./sudoku2 sudoku17.1000.txt +RTS -N2 -l
$ threadscope sudoku2.eventlog
```

- Partición en dos trozos no es suficiente (cantidad de trabajo descompensado).

Ejemplo Sudoku

Solución paralela con partición estática



Una línea HEC por núcleo. En verde y más grueso, el tiempo ejecutando código. En naranja y más fino, ejecutando garbage collector (GC).

Ejemplo Sudoku

Solución paralela con partición dinámica (map paralelo)

Un spark por elemento de la lista de tableros (dinámica) con **map paralelo**.
Fichero *sudoku3.hs*:

```
parMap :: (a -> b) -> [a] -> Eval [b]
parMap f [] = return []
parMap f (a:as) = do b <- rpar (f a)
                     bs <- parMap f as
                     return (b:bs)

main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f
  let puzzles   = lines file
      solutions = runEval (parMap solve puzzles)
  print (length (filter isJust solutions))
```

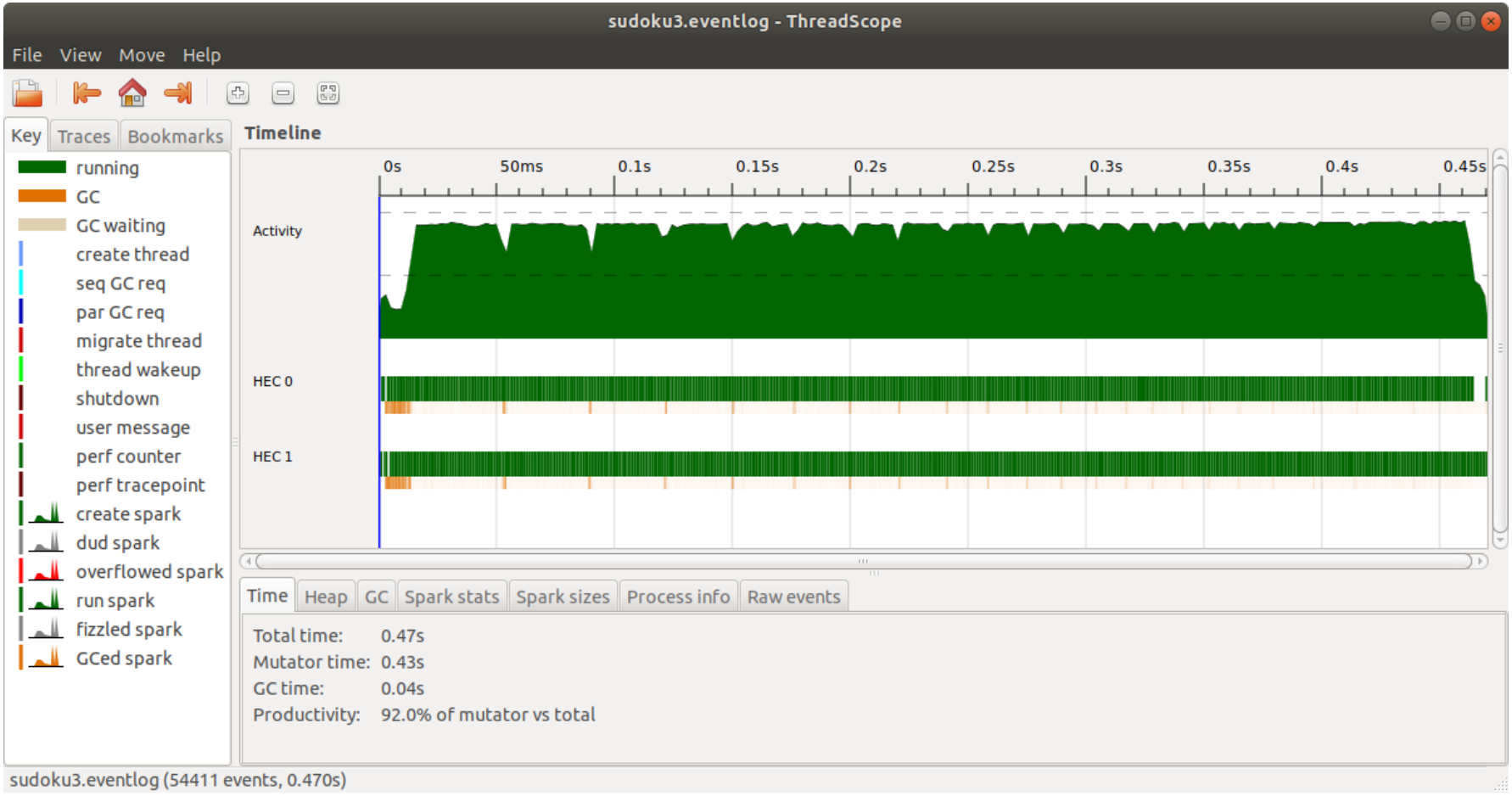
Resultado:

```
SPARKS: 1000 (1000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

Total   time    3.21s  (  1.61s elapsed)
```

Ejemplo Sudoku

Solución paralela con partición dinámica (map paralelo)



1 Introducción al Paralelismo

2 Paralelismo básico en Haskell

3 Ejemplo map: sudoku

4 Ejemplo reduce

5 Otras formas de paralelismo

6 Bibliografía

Ejemplo reduce

- Una forma de hacer reduce es con dos funciones:
  - Una función que aplique solo un paso de la reducción aplicando la operación cada par de elementos.

```
parReduce' :: Num a => [a] -> Eval [a]
parReduce' [] = return []
parReduce' [x] = return [x]
parReduce' (x:y:xs) = do b <- rpar (x + y)
                         bs <- parReduce' xs
                         rseq (b:bs)
```

Ejemplo reduce

- Una forma de hacer reduce es con dos funciones:
  - Una función que aplique solo un paso de la reducción aplicando la operación cada par de elementos.
  - Una función que llame a la anterior en cada iteración hasta conseguir un solo elemento.

```
parReduce :: Num a => [a] -> a
parReduce [] = error "Debe contener al menos un elemento"
parReduce [x] = x
parReduce xs = runEval $ do ms <- parReduce' xs
                            return (parReduce ms)
```

- Se puede mejorar imponiendo un corte en el tamaño de la lista.
  - Si es menor a ese corte, no vale la pena aplicar paralelismo (se debe ajustar de forma experimental).

1 Introducción al Paralelismo

2 Paralelismo básico en Haskell

3 Ejemplo map: sudoku

4 Ejemplo reduce

5 Otras formas de paralelismo

6 Bibliografía

Otros módulos de paralelismo

- **Estrategias de evaluación**: construido sobre la mónada Eval, provee otra capa de abstracción, consiguiendo separar control de paralelismo.

```
let solutions = map solve puzzles `using` parList rseq
```

- **La mónada par** (paralelismo de flujo de datos): Basado en llamadas `fork` y variables `IVar` para comunicación entre procesos.
- **Accelerate** (computación GPU): colección de funciones paralelizadas en GPU basados en Arrays. Disponible en Data.Array.Accelerate.
- **Repa** (paralelismo de datos): basado en arrays, posibilidad de generar código.

1. Introducción al Paralelismo
2. Paralelismo básico en Haskell
3. Ejemplo map: sudoku
4. Ejemplo reduce
5. Otras formas de paralelismo
6. Bibliografía

Bibliografía


S. Marlow. Parallel and Concurrent Programming in Haskell, O'Reilly. Capítulo 2..


S. Marlow. Parallel and Concurrent Programming in Haskell. CEFP 2011, LNCS 7241 (2012), 339-401.