Paralelismo de dados com futures

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

31 de julho de 2018

Crédito de parte das imagens, a menos se especificado: Wikipedia

Objetivos

- Familiarizar-se com o uso de futures
- Praticar exemplos de paralelismo de dados com futures

Revisão: modelos de programação concorrente em Racket

- Dois modelos de programação concorrente (i.e. monoprocessado):
- **Threads e locks**:
  - Utiliza threads lógicas
  - Procedimentos para comunicação de threads
  - Travas (e.g. semáforos e variáveis de condição)
- **Passagem de mensagens**:
  - Complementar ao modelo de threads e locks
- Dois modelos de programação concorrente e paralela (i.e. multiprocessado):
  - **Futures**:
    - Paralelismo restrito a tarefas “seguras” (e.g. não bloqueantes)
  - **Places**:
    - Combina troca de mensagens com futures
    - Mais geral e mais flexivel (porém com maior overhead)
    - Atinge paralelismo distribuído

Futures and promises

Conceitos básicos

Futures e promises

Modelo de programação concorrente que consiste em definir promessas que gerarão, concorrentemente ou paralelamente, futuros.

- Conceitualmente baseado no modelo de avaliação preguiçosa
- Surge como recurso da programação funcional concorrente
- Limitações:
  - Operações não-seguras bloqueiam as threads
  - Operações não-seguras são numerosas (e.g. alocação e desalocação de memória, E/S, etc.)
  - Operações não-seguras, se numerosas e frequentes, tentem a degradar a execução para modo sequencial
- Na prática, é útil apenas para algoritmos de espaço constante, em memória principal, com alocação na pilha
- Ainda assim, múltiplos problemas podem ser resolvidos dentro dessa restrição

Paralelismo de dados com futures

Paralelismo de dados

Processo de paralelização que consiste em distribuir dados em diferentes nós de processamento.

- Princípio: a mesma operação é executada em diferentes subconjuntos dos dados
- Futures podem simplificar paralelismo de dados
- Com futures, podemos efetuar paralelismo efetivamente, desde que eliminamos operações não-seguras
- Estratégias:
  - Carregar dados com antecedência na memória principal
  - Utilizar estruturas imutáveis de acesso aleatório, i.e. vetores[1]
  - Escolher algoritmos que não necessitam alocar memória na heap, i.e. utilizam apenas a memória da stack[2]

[1] Se permitíssemos estruturas de acesso aleatório mutáveis, nosso espaço de problemas possíveis aumentaria bastante, mas precisamos manter restrição de imutabilidade para garantir a aderência ao paradigma funcional

[2] No modelo de futures da Racket, cada thread possui uma pilha, mas a heap é mantida pela thread principal, logo alterações na heap bloqueiam todas as threads

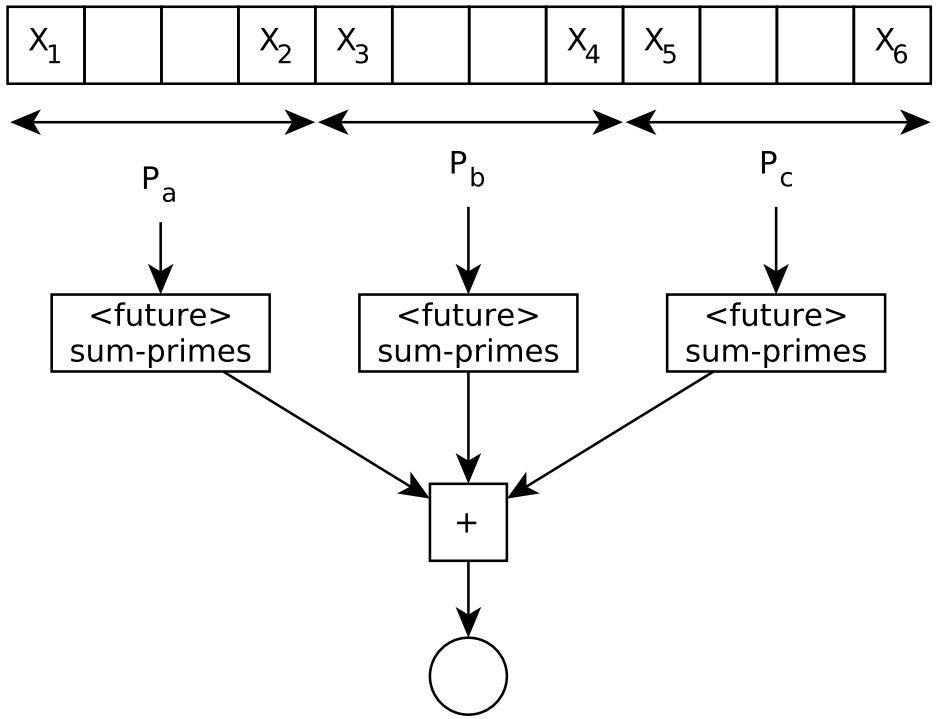
Exercício 1

1. Construa o procedimento `(sum-primes x y)` que some todos os números primos do intervalo $[x, y]$. Não é permitido usar listas, i.e., o algoritmo deve ser $O(1)$ em espaço[3].
2. Defina o procedimento `(par-sum-primes x y)` o qual utiliza três futures para calcular a soma de primos. Para tal, o intervalo [x,y] deve ser dividido em três particões, e cada future efetua a operação em uma das particões. Ao final, os resultados dos três futures devem ser somados pela thread primordial.
    1. Teste com cada partição tendo 1/3 do tamanho da particão $[x, y]$
    2. Teste com o seguinte layout de partições, começando em x e terminando em y: a partição 1 tem 50% dos elementos, a partição 2 tem 35% dos elementos e a partição 3 tem 15% dos elementos. Por exemplo, se $[x, y] = [1, 100]$, as partições são $[1, 50]$, $[51, 85]$, $[86, 100]$.
    3. Compare o tempo de execução dos três modos de processamento, e detemine qual apresentou a melhor eficiência.

[3] Essa restrição visa minimizar a atuação do garbage collector, o que pode prejudicar o uso da futures

Exercício 1

Arquitetura da versão paralela com três partições



- **Particionamento**: Dividimos os dados em três partições (P_a, P_b, P_c)
- **Mapeamento**: Cada partição é processada em uma instância da mesma tarefa, paralelamente
- **Redução**: Os resultados parciais das tarefas são combinados em um resultado final

Exercício 1

Solução: versão iterativa e versão paralela

```
;;;;;; iterative version
(define (sum-primes-a x y)
  (let loop ([i x] [s 0])
    (cond [(> i y) s]
          [(prime? i) (loop (add1 i) (+ s i))]
          [else (loop (add1 i) s)])))
```

```
; parallel with 1/3 1/3 1/3 partitions
(define (par-sum-primes-a x y)
  (define part-size (/ (- y x) 3))
  (define x1 x)
  (define x2 (floor (+ x part-size)))
  (define x3 (add1 x2))
  (define x4 (floor (+ x2 part-size)))
  (define x5 (add1 x4))
  (define x6 y)
  (let ([p-a (future (lambda () (sum-primes-a x1 x2)))]
        [p-b (future (lambda () (sum-primes-a x3 x4)))]
        [p-c (future (lambda () (sum-primes-a x5 x6)))])
    (+ (touch p-a) (touch p-b) (touch p-c))))
```

Exercício 1

Testes de execução: versão iterativa e versão paralela

```
1  #lang racket
2
3  (require "sum-primes-futures.rkt")
4
5  (define (main args)
6    (cond [(< (vector-length args) 1)
7           (displayln "Usage: sum-primes-futures-timing <n-primes>"
8                      (current-error-port))
9           (exit 1)])
10   (let ([n-primes (string->number (vector-ref args 0))])
11     (displayln (time (sum-primes-a 2 n-primes)))
12     (displayln (time (par-sum-primes-a 2 n-primes)))))
13
14 (main (current-command-line-arguments))
```

```
$ racket sum-primes-futures-timing.rkt 10000000
cpu time: 11984 real time: 11970 gc time: 0
3203324994356
cpu time: 15604 real time: 6346 gc time: 0
3203324994356
```

- Mesmo usando três threads, o tempo caiu pela metade
- Vamos analisar os futures

Exercício 1

Análise dos futures: versão paralela

```
1  #lang racket
2
3  (require "sum-primes-futures.rkt")
4  (require future-visualizer)
5
6  (define (main args)
7    (cond [(< (vector-length args) 1)
8           (displayln "Usage: sum-primes-future-vis <n-primes>"
9                      (current-error-port))
10          (exit 1)])
11   (let ([n-primes (string->number (vector-ref args 0))])
12     (visualize-futures (par-sum-primes-a 2 n-primes))))
13
14 (main (current-command-line-arguments))
```

Exercício 1

Análise dos futures: versão paralela

- No exemplo anterior, as threads perdem um tempo inicial com jit-on-demand (não há muito o que possamos fazer)
- A thread 1 termina antes das outras
  - Embora cada partição contenha a mesma quantidade de números, a partição 1 contém os números menores (cujo teste de primalidade tem execução mais curta)
- Vamos tentar balancear a carga

Exercício 1

Análise dos futures: versão com 3 partições não-uniformes

```
1  ; parallel with 0.5 0.35 0.15 partitions
2  (define (par-sum-primes-b x y)
3    (define interval-size (- y x))
4    (define part-a-size (* (/ 1 2) interval-size))
5    (define part-b-size (* (/ 35 100) interval-size))
6    (define x1 x)
7    (define x2 (floor (+ x1 part-a-size)))
8    (define x3 (add1 x2))
9    (define x4 (floor (+ x2 part-b-size)))
10   (define x5 (add1 x4))
11   (define x6 y)
12   (let ([p-a (future (lambda () (sum-primes-a x1 x2)))]
13         [p-b (future (lambda () (sum-primes-a x3 x4)))]
14         [p-c (future (lambda () (sum-primes-a x5 x6)))])
15     (+ (touch p-a) (touch p-b) (touch p-c))))
```

```
$ racket sum-primes-futures-timing.rkt 10000000
cpu time: 11980 real time: 11961 gc time: 0
3203324994356
cpu time: 15992 real time: 6904 gc time: 0
3203324994356
cpu time: 14868 real time: 6102 gc time: 0
3203324994356
```

- Houve uma pequena melhora

Exercício 1

Aumentando para 4 futures

- Vamos tentar alocar mais threads para trabalhar

```
1  ; parallel with 0.25 0.25 0.25 0.25 partitions
2  (define (par-sum-primes-d x y)
3    (define part-size (/ (- y x) 4))
4    (define x1 x)
5    (define x2 (floor (+ x part-size)))
6    (define x3 (add1 x2))
7    (define x4 (floor (+ x2 part-size)))
8    (define x5 (add1 x4))
9    (define x6 (floor (+ x4 part-size)))
10   (define x7 (add1 x6))
11   (define x8 y)
12   (let ([p-a (future (lambda () (sum-primes-a x1 x2)))]
13         [p-b (future (lambda () (sum-primes-a x3 x4)))]
14         [p-c (future (lambda () (sum-primes-a x5 x6)))]
15         [p-d (future (lambda () (sum-primes-a x7 x8)))])
16     (+ (touch p-a) (touch p-b) (touch p-c) (touch p-d))))
```

```
$ racket sum-primes-futures-timing.rkt 10000000
cpu time: 11980 real time: 11963 gc time: 0
3203324994356
cpu time: 15056 real time: 6445 gc time: 0
3203324994356
cpu time: 15096 real time: 5878 gc time: 0
3203324994356
cpu time: 17616 real time: 5542 gc time: 0
3203324994356
```

- Mais uma pequena melhora

Exercício 1

Aumentando as threads: parte 2

- Agora o programa está usando todos os cores do processador:

```
> (processor-count)
4
```

- Mas as threads dos intervalos iniciais ainda não estão sendo bem aproveitadas

Exercício 1

Usando espaço adicional: parte 2

- **Exercício 2.** experimente decompor os dados em $n \geq 4$ partições, variando o valor de n, e verifique experimentalmente, via tempo de execução, qual é a quantidade de threads ótima para esse problema. Verifique se, com mais threads concorrendo, o uso dos processadores é sempre otimizado.

Exercício 3

A série de Leibniz estima o valor de π:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

1. Implemente um procedimento iterativo que aproxime o valor de π considerando n termos da série
2. Proponha uma outra versão que use paralelismo de dados para computar a série, usando todos os processadores disponíveis no computador
3. Compare os desempenhos da versão iterativa e da versão paralela

Exercício 3

Solução: parte 1

- Já vimos essa implementação antes:

```
1  ; iterative sequential pi approximation
2  (define (pi-seq n)
3    (let loop ([p 0] [d 1] [i 0])
4      (if (= i n)
5          (* p 4)
6          (if (even? i)
7              (loop (+ p (/ 1.0 d)) (+ d 2) (+ i 1))
8              (loop (- p (/ 1.0 d)) (+ d 2) (+ i 1))))))
```

```
> (pi-seq 1000)
3.140592653839794
```

Exercício 3

Solução: parte 2

- Para usar todos os processadores disponíveis, vamos criar algumas operações genéricas e colocá-las numa biblioteca:

```
1  #lang racket
2
3  (provide in-partition-range)
4  (provide partition-thunks)
5
6  ; creates a stream range in [start end+1] by chunk steps
7  (define (in-partition-range start end chunk)
8    (if (> start end)
9        (stream-cons (add1 end) (stream))
10       (stream-cons start (in-partition-range (+ chunk start) end chunk))))
11
12 ; creates one thunk for each processor
13 ; every thunk process a data partition,
14 ;   defined as the half-open interval [x,y)
15 ; assumes a higher order proc that receives the partition as args
16 (define (partition-thunks start end proc)
17   (define chunk (ceiling (/ (- end start) (processor-count))))
18   (define (make-thunk i j)
19     (let () (lambda () (proc i j))))
20   (let ([pr (in-partition-range start end chunk)])
21     (for/list ([i pr]
22                [j (stream-rest pr)])
23       (make-thunk i j))))
```

Exercício 3

Solução: parte 2

- Para resolver o problema por partes, vamos adaptar a parametrização do algoritmo iterativo:

```
1  ; compute the series in a range
2  ; (future unsafe, triggers garbage collection)
3  (define (pi-iter-a i n)
4    (let loop ([p 0] [d (+ (* 2 i) 1)] [i i])
5      (if (= i n)
6          p
7          (if (even? i)
8              (loop (+ p (/ 1.0 d)) (+ d 2) (+ i 1))
9              (loop (- p (/ 1.0 d)) (+ d 2) (+ i 1))))))
```

- E orquestrar a execução paralela:

```
1  ; approximate pi using all available processors
2  (define (pi-par n)
3    (let* ([thunks (partition-thunks 0 (sub1 n) pi-iter-a)]
4           [futures (map future thunks)])
5      (* 4 (foldl + 0 (map touch futures)))))
```

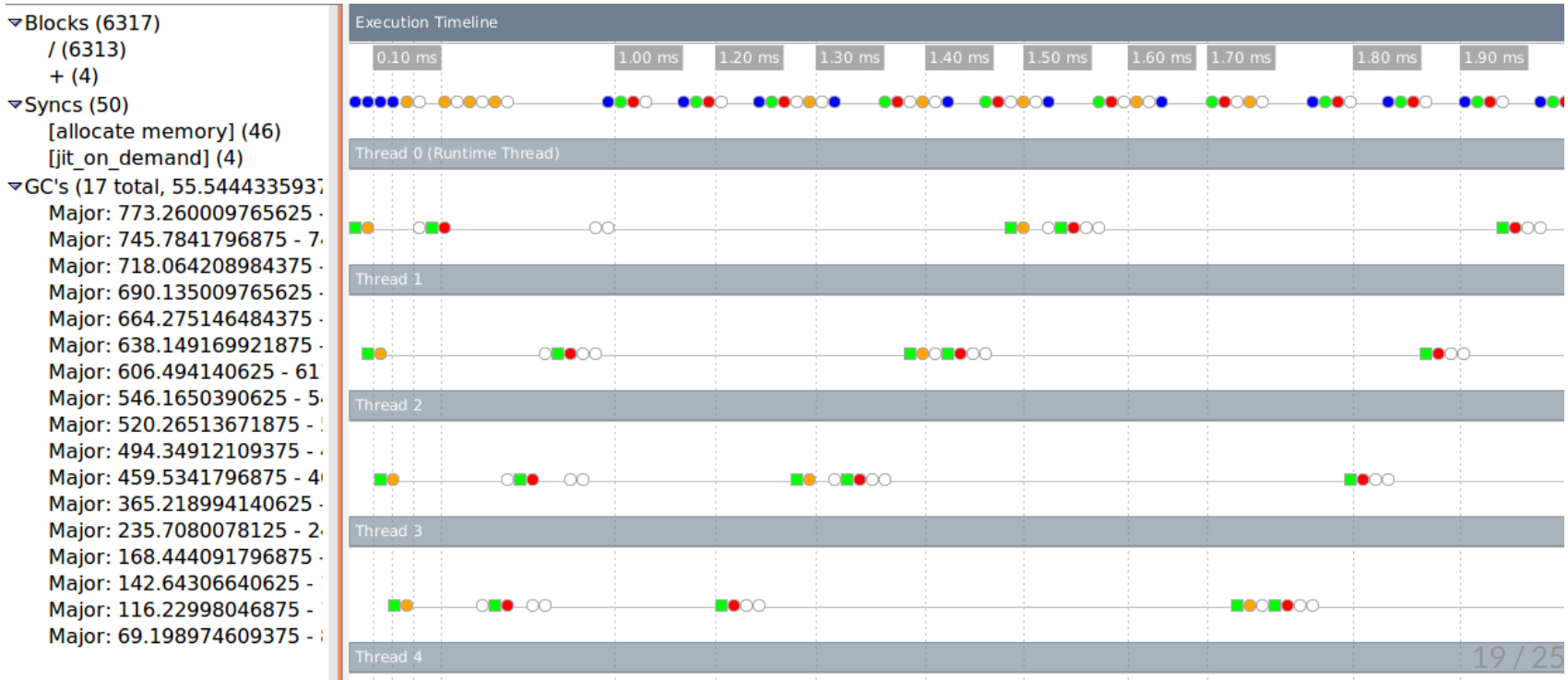
Exercício 3

Solução: parte 2

- Praticamente não houve melhora:

```
$ racket pi-futures-timing.rkt 10000000
cpu time: 388 real time: 387 gc time: 20
3.1415925535897915
cpu time: 380 real time: 379 gc time: 4
3.1415925535897427
```

- Vamos analisar os futures:



Exercício 3

Solução: parte 2

- A visualização nos mostra que a execução degradou para sequencial
- A maioria da operações bloqueantes são aritméticas sobre números de precisão arbitrária
- Essas operações demandam alocação de memória na heap
- Vamos usar `flonum` (reais de precisão fixa[4]):

```
1  (require racket/flonum)
2
3  ; compute pi series in a range
4  ; (future safe, uses flonum)
5  (define (pi-iter-b i n)
6    (let loop ([p 0.0] [d (fl+ (fl* 2.0 (->fl i)) 1.0)] [i i])
7      (if (= i n)
8          p
9          (if (even? i)
10             (loop (fl+ p (fl/ 1.0 d)) (fl+ d 2.0) (+ i 1))
11             (loop (fl- p (fl/ 1.0 d)) (fl+ d 2.0) (+ i 1))))))
```

[4] Precisão de 32 bits ou de 64 bits, sempre o máximo disponível no processador

Exercício 3

Solução: parte 2

- Testando o desempenho com a modificação:

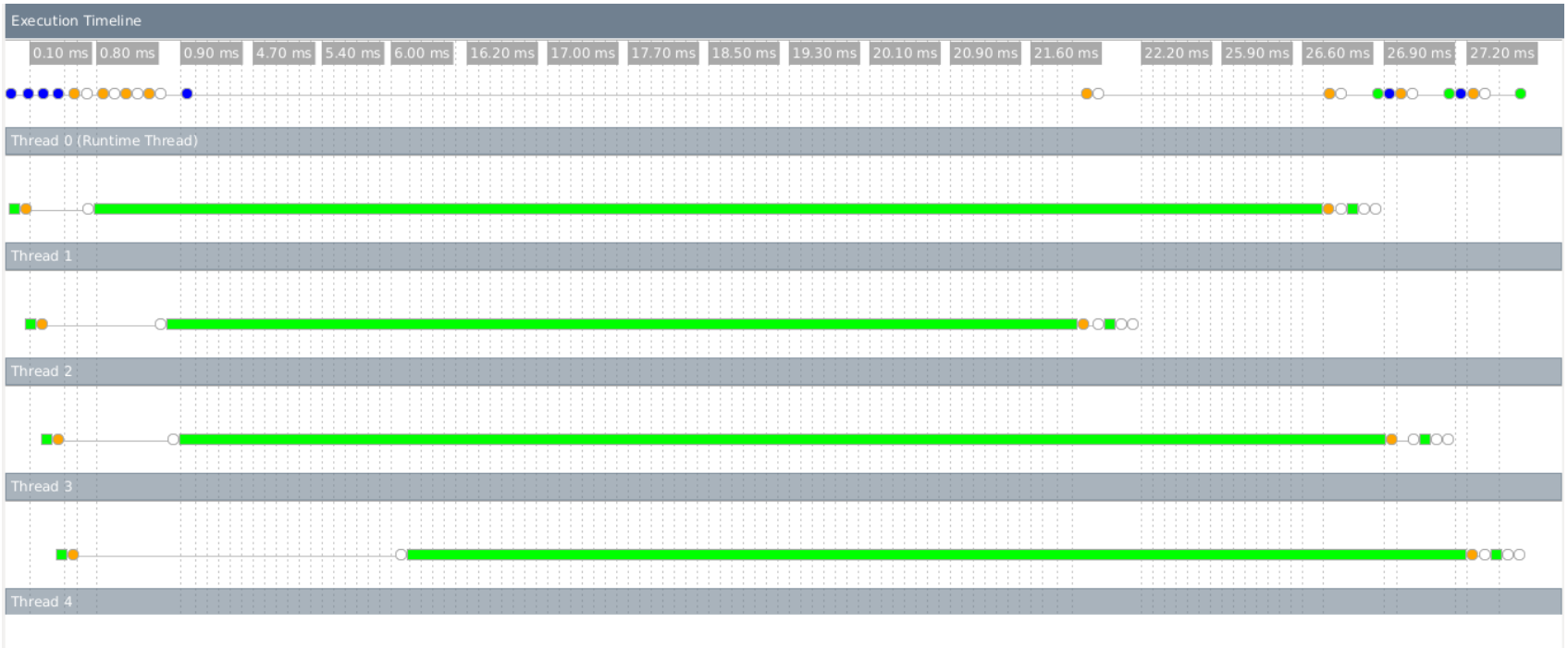
```
$ racket pi-futures-timing.rkt 10000000
cpu time: 388 real time: 387 gc time: 20
3.1415925535897915
cpu time: 160 real time: 49 gc time: 0
3.1415925535897427
```

- O tempo melhorou consideravelmente
- Os ganhos combinaram:
  - Maior eficiência das operações aritméticas (números como tipos “primitivos” vs. “objetos”), i.e. se tivéssemos utilizado flonum na versão sequencial espera-se que os ganhos sejam de aprox. 4:1 ao invés de 8:1
  - Menos bloqueios devido à alocação de memória

Exercício 3

Solução: parte 2

- Vamos analisar o reflexo nos futures:



Exercícios

Exercício 4

Construa variações do procedimento `(count-occur x v)` que determina a quantidade de ocorrências do elemento `x` no vetor[5] `v`:

1. Defina a versão `(count-occur-seq x v)` que resolve o problema sequencialmente
2. Defina a versão `(count-occur-par x v)` que resolve o problema paralelamente, utilizando todos os processadores disponíveis (i.e. utilizar paralelismo de dados sobre partições do vetor). O algoritmo não pode alocar ou desalocar memória na heap.
3. Avalie o tempo de execução das duas soluções

[5] Usar um vector imútável da Racket

Exercício 5

O predicado `(string-contains? s contained)` testa se a string `s` contém a string `contained`:

```
> (string-contains? "hello world" "hell")
#t
> (string-contains? "hello world" "world!")
#f
```

Implemente a seguinte estratégia para paralelizar a busca de substring:

- Criar uma thread para cada sufixo de `s` que seja maior ou igual a `contained`
- Verificar se `contained` está contida no sufixo
- Combinar os resultados das threads

Por exemplo, se `s = "aviao"` e `contained = "vi"`, as threads procuram por `"vi"` em `"aviao"`, `"viao"`, `"iao"` e `"ao"`. Implemente o procedimento `(par-string-contains? s contained)` que implemente essa estratégia usando futures. Não é preciso gerar os sufixos como novas strings, i.e. use apenas índices da string para delimitar partições.

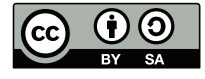
Paralelismo de dados com futures

Profs. Diogo S. Martins e Emilio Francesquini

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

31 de julho de 2018



Crédito de parte das imagens, a menos se especificado: Wikipedia