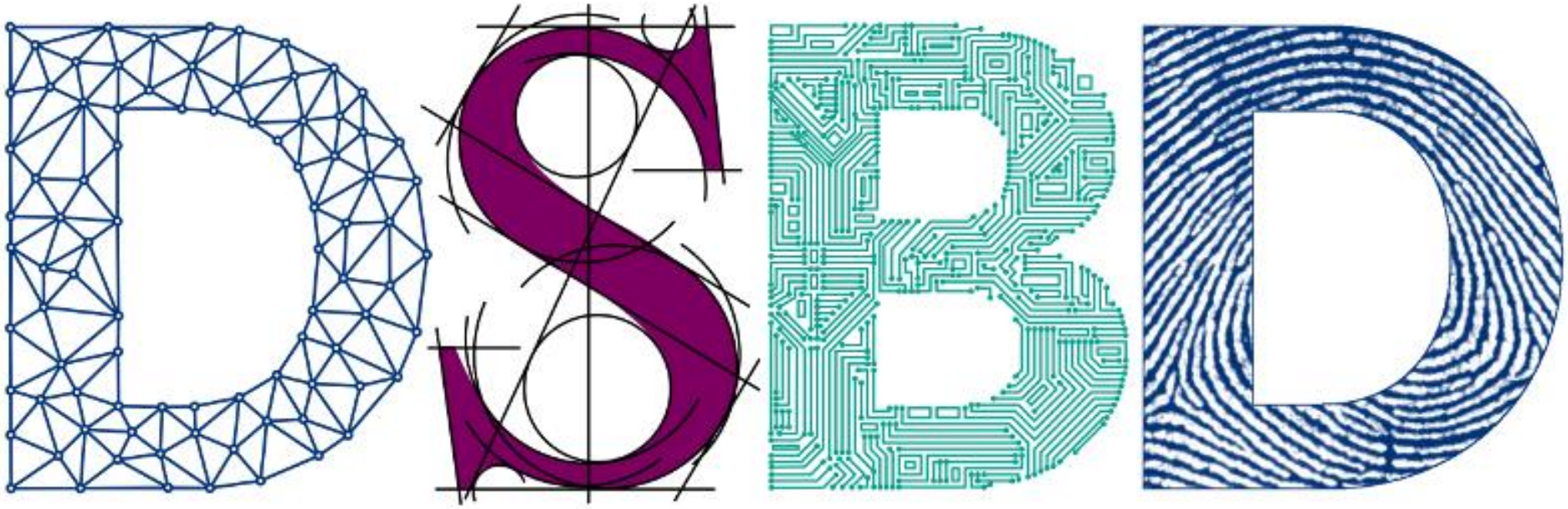


ESCREVENDO CÓDIGOS PARALELOS COM PYTHON

Infraesrutura Computacional Pt.3

Marco A. Z. Alves

[Bastian Venthur - Berlin Institute of Technology – 2011]

POR QUE CONCORRÊNCIA?

1970-2005

- CPUs tornam-se cada vez mais rápidas a cada ano
- Lei de Moore: O número de transistores [...] dobra aproximadamente a cada dois anos.

Mas!

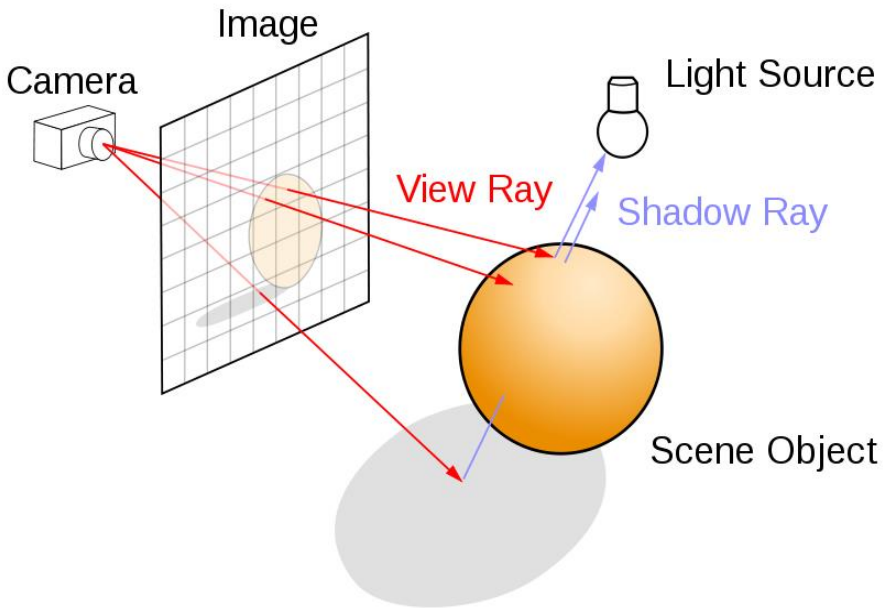
- Limites físicos: Miniaturização em níveis atômicos, consumo de energia, calor produzido por CPUs, etc.
- Estagnação nas taxas de clock da CPU desde 2005

Desde 2005

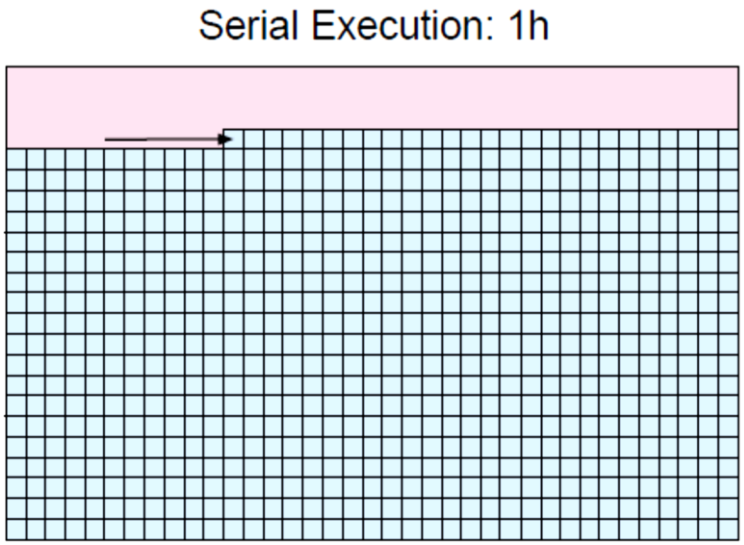
- Os produtores de chips buscavam mais núcleos em vez de taxas de clock mais altas.

APLICAÇÕES ÚTEIS PARA CONCORRÊNCIA RAY TRACING

Traça o caminho de um olho imaginário (câmera) através de cada pixel em uma tela e calcule a cor do (s) objeto (s) visível (s) através dele.

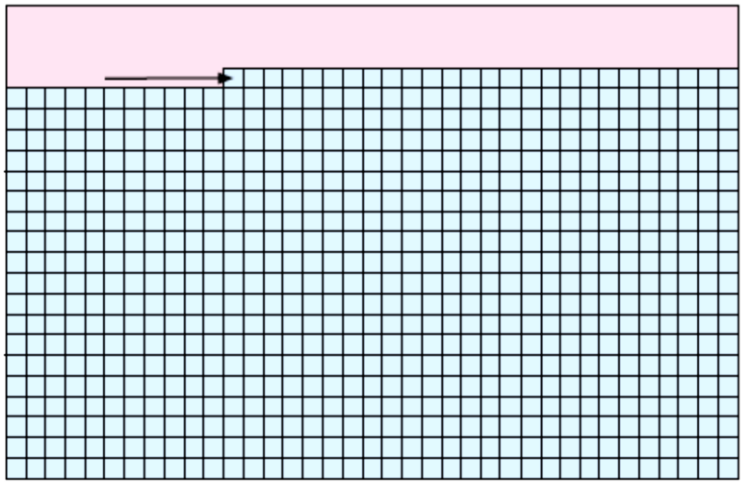


APLICAÇÕES ÚTEIS PARA CONCORRÊNCIA RAY TRACING

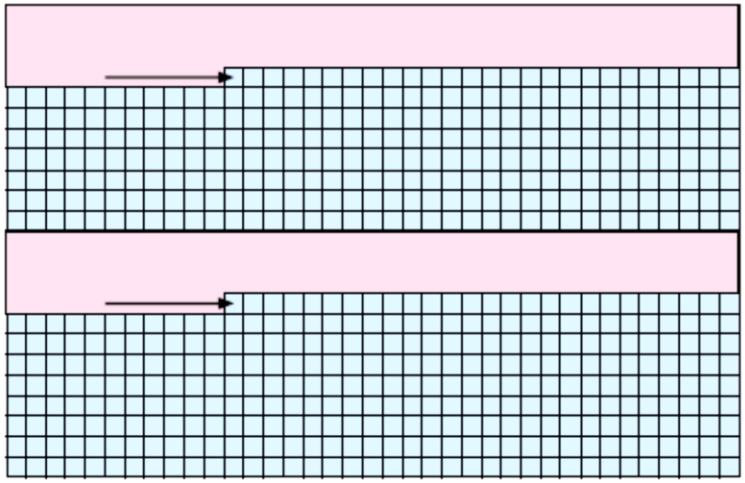


APLICAÇÕES ÚTEIS PARA CONCORRÊNCIA RAY TRACING

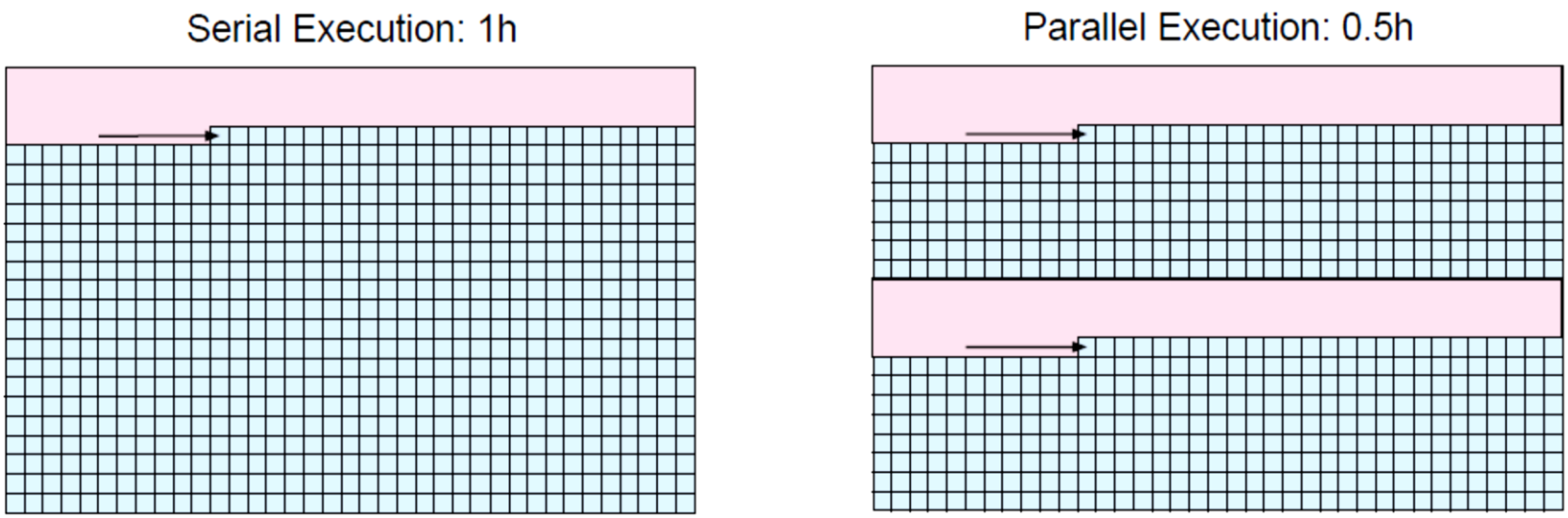
Serial Execution: 1h



Parallel Execution: 0.5h



APLICAÇÕES ÚTEIS PARA CONCORRÊNCIA RAY TRACING



Ray Tracing é embaraçosamente paralelo:

- Pouco ou nenhum esforço para separar o problema em tarefas paralelas
- Nenhuma dependência ou comunicação entre as tarefas

OUTRO EXEMPLO
ALGUM CÁLCULO ALEATÓRIO

```
L1: a = 2
L2: b = 3
L3: p = a + b
L4: q = a * b
L5: r = q - p
```

Alguma sincronização ou comunicação entre as tarefas é necessária para resolver este cálculo corretamente.

OUTRO EXEMPLO
ALGUM CÁLCULO ALEATÓRIO

```
L1: a = 2

L2: b = 3

L3: p = a + b

L4: q = a * b

L5: r = q - p
```

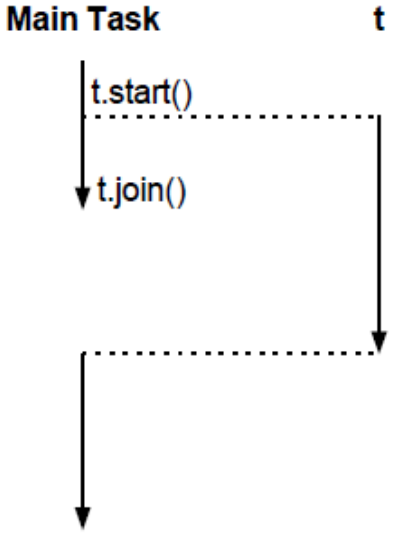
L1|| L2; L3 || L4; L5
L3 e L4 precisam esperar por L1 and L2
L5 precisa esperar por L3 e L4

Alguma sincronização ou comunicação entre as tarefas é necessária para resolver este cálculo corretamente.

INICIANDO E TERMINANDO UMA TAREFA

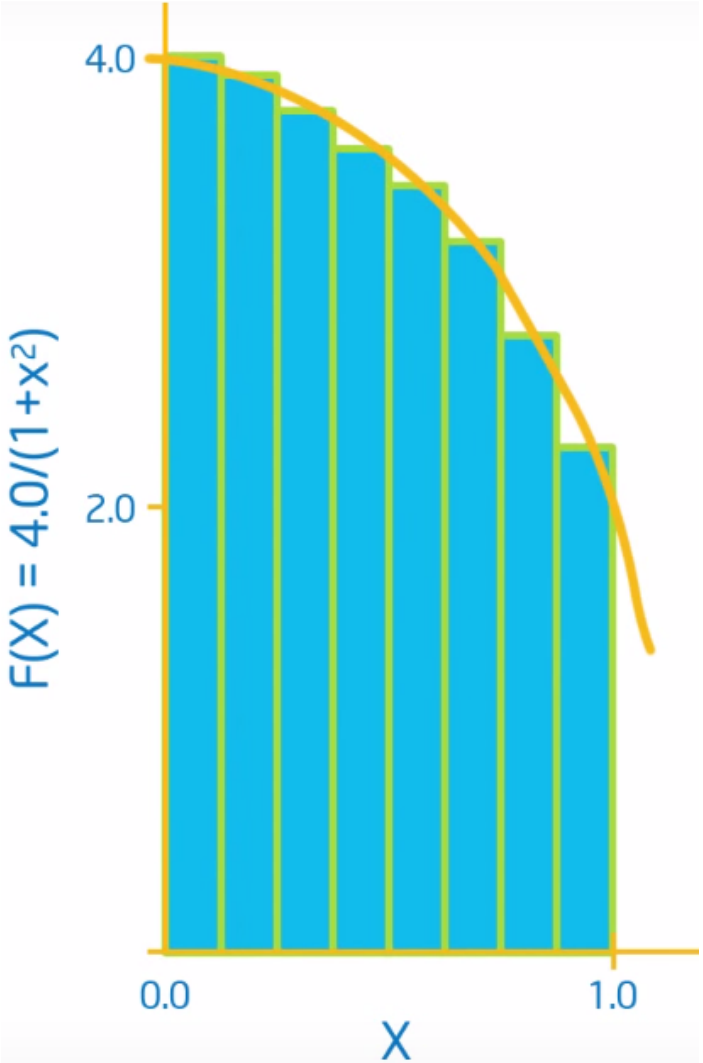
Uma tarefa é um programa ou método que é executado simultaneamente.

```
# Inicia a tarefa T
# T irá executar concorrentemente e
# (i.e. esse) programa irá continuar
T = Task()
T.start()
...
# wait for t to finish
T.join()
```



Join sincroniza a tarefa pai com a tarefa filho, aguardando que a tarefa filho termine.

EXERCÍCIO INTEGRAÇÃO NUMÉRICA



Matematicamente, sabemos que:

$$\int_0^1 \frac{4.0}{1+x^2}\,dx = \pi$$

Podemos aproximar essa integral como a soma de retângulos:

$$\sum_{i=0}^{n} F(x_i)\Delta x \cong \pi$$

Onde cada retângulo tem largura Δx e altura $F(x_i)$ no meio do intervalo i.

PI SEQUENCIAL

```
import time
def pi(start, end, step):
    print "Start: " + str(start)
    print "End: " + str(end)
    sum = 0.0
    for i in range(start, end):
        x = (i+0.5) * step
        sum = sum + 4.0/(1.0+x*x)
    return sum
if __name__ == "__main__":
    num_steps = 100000000 #100.000.000
    sums = 0.0
    step = 1.0/num_steps

    tic = time.time()
    sums = pi(0, num_steps, step)
    toc = time.time()

    pi = step * sums
    print pi
    print "Pi: %.8f s" %(toc-tic)
```

DOIS TIPOS DE TAREFAS: THREADS E PROCESSOS

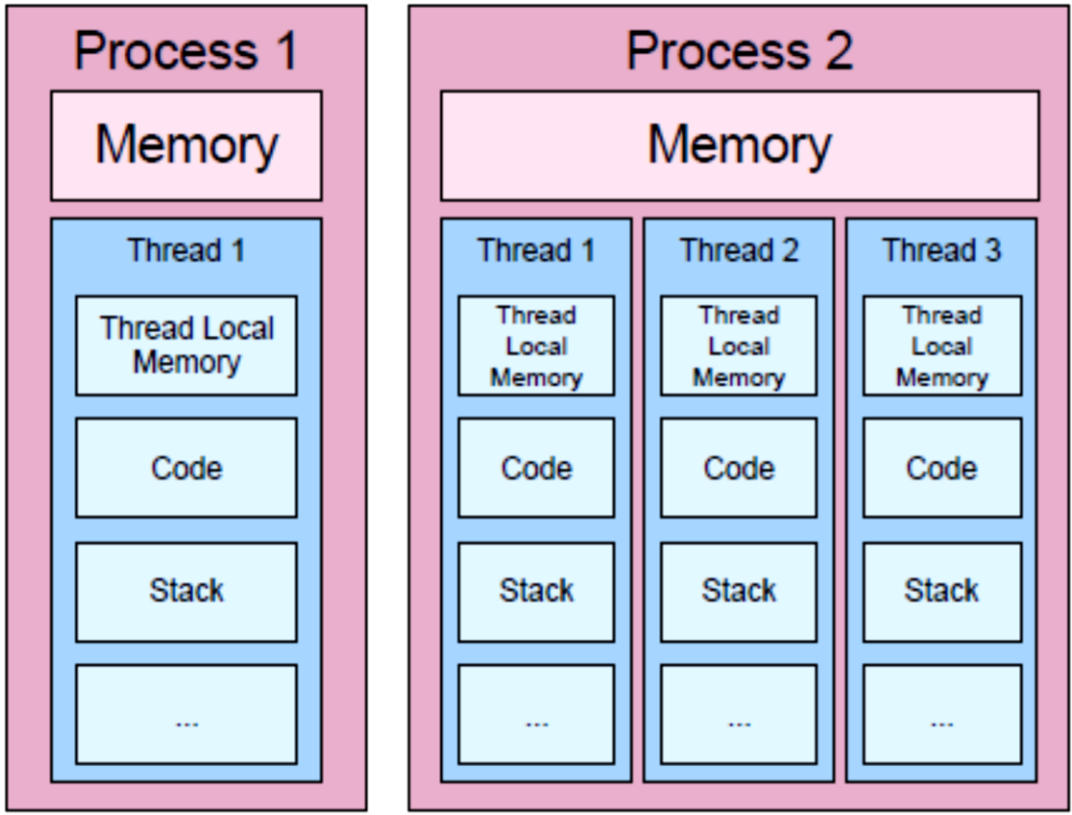
Um processo tem uma ou mais threads

Os processos possuem sua própria memória (variáveis, etc.)

Threads compartilham a memória do processo ao qual pertencem

Threads também são chamados de processos leves:

- Elas são geradas mais rápido que processos
- Trocas de contexto (se necessário) são mais rápidas



COMUNICAÇÃO ENTRE TAREFAS MEMÓRIA COMPARTILHADA E PASSAGEM DE MENSAGENS

Basicamente você tem dois paradigmas:

1. Memória Compartilhada
   - Taks A e B compartilham alguma memória
   - Sempre que uma tarefa modifica uma variável na memória compartilhada, a(s) outra(s) tarefa(s) vê essa mudança imediatamente

2. Passagem de Mensagem
   - A tarefa A envia uma mensagem para a tarefa B
   - A tarefa B recebe a mensagem e faz algo com ela

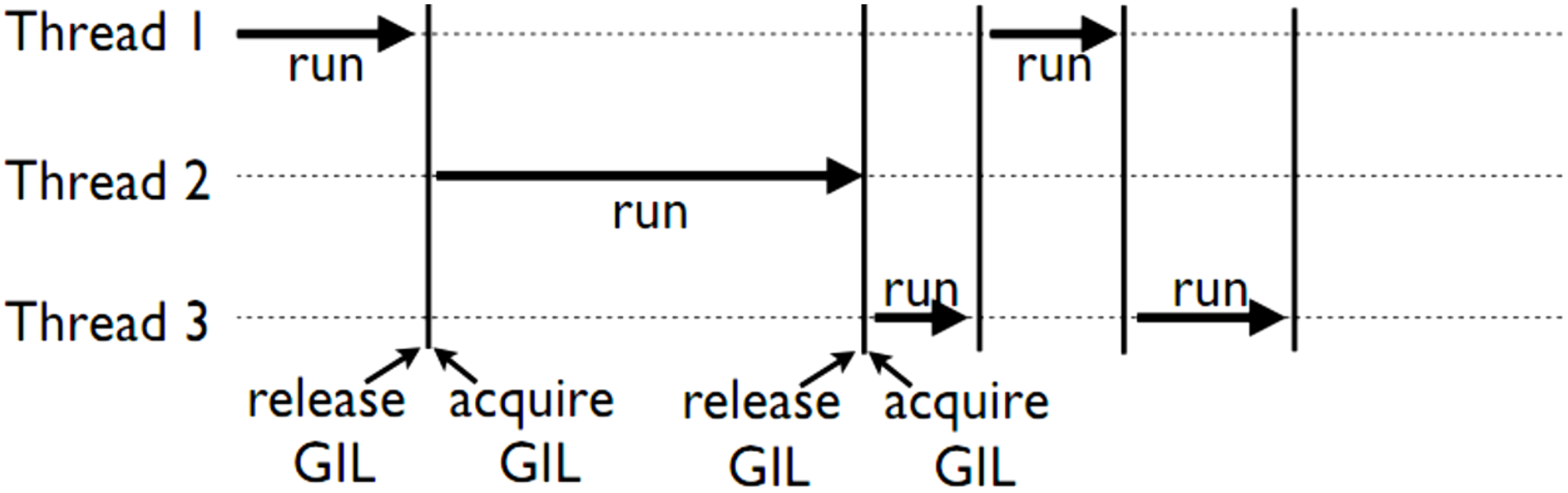
O primeiro paradigma é geralmente usado com threads e o último com processos.

PARALELISMO DE THREAD

No entanto, o interpretador padrão do Python foi projetado com a simplicidade em mente e possui um mecanismo seguro para thread, o chamado “GIL” (Global Interpreter Lock).

Para evitar conflitos entre threads, ele executa apenas uma instrução por vez (o chamado processamento serial ou single-threading).

COMO FUNCIONA A GIL



Na verdade, impede que os threads sejam executados em paralelo no Python.

COMO FUNCIONA A GIL

Na verdade, não é tão ruim assim:

Os desenvolvedores de módulos não precisam se preocupar com efeitos de encadeamento maléficos.

Não é problemático assim ao executar vários threads de E/S (e.g. disco) simultaneamente.

Geralmente, as extensões C (Cython) liberam o GIL durante a operação, permitindo usar várias threads simultaneamente.

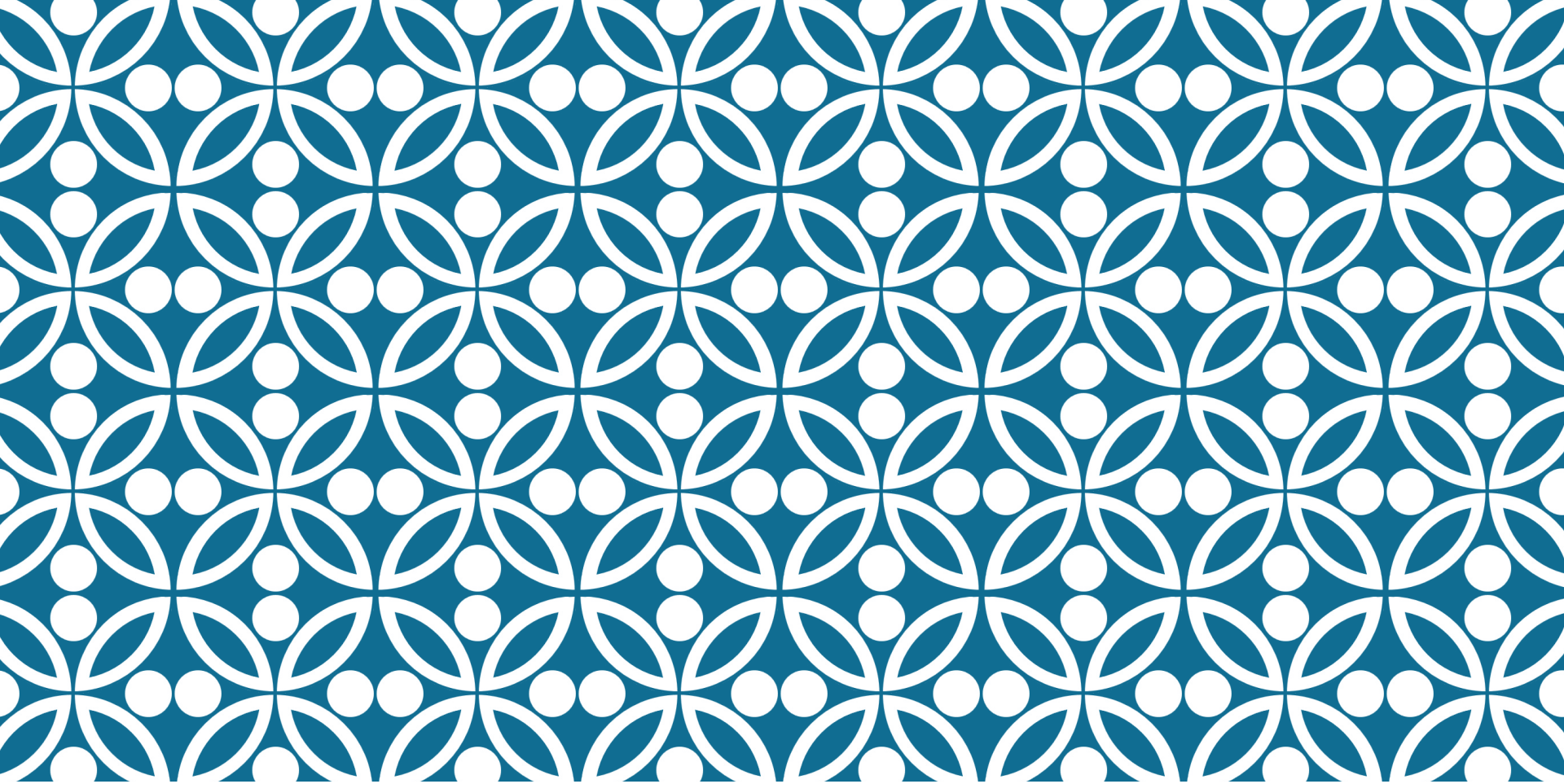
COMO FUNCIONA A GIL

Uma maneira de superar as limitações do GIL discutidas acima é usar vários processos completos em vez de threads. Cada processo tem seu próprio GIL, então eles não bloqueiam um ao outro da mesma maneira que os threads fazem.

A partir do python 2.6, a biblioteca padrão inclui um módulo de multiprocessamento, com a mesma interface do módulo de threading.

É possível compartilhar memória entre processos, incluindo matrizes numpy.

Isso permite a maioria dos benefícios das threads sem os problemas do GIL. Ele também fornece uma interface simples Pool(), que apresenta comandos map e apply.



O MÓDULO DE MULTIPROCESSAMENTO

O MÓDULO DE MULTIPROCESSAMENTO

O módulo de multiprocessamento na biblioteca padrão do Python tem muitos recursos poderosos.

Se você quiser ler sobre todas as dicas, truques e detalhes essenciais, use a documentação oficial como um ponto de partida.

Vamos começar com uma breve visão geral de diferentes abordagens para mostrar como o módulo de multiprocessamento pode ser usado para programação paralela.

O MÓDULO DE MULTIPROCESSAMENTO

Tarefas simultâneas são legais e agora você tem as ferramentas para liberar todo o poder do seu sistema multicore / cluster / supercomputador, mas ...

Teremos que cuidar de várias situações:

- Você não tem absolutamente nenhuma garantia sobre o momento em que partes específicas de suas tarefas são executadas.
- Todas as variáveis de cada processo (escopo) só são visíveis dentro desse processo
- O acesso a uma variável de processo é explícito (por exemplo, passando-a como um argumento para o processo ou via mensagem)

EXEMPLO ZERO
HELLO WORLD

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob filho', ))
    p.start()
    p.join()
```

EXEMPLO ZERO
HELLO WORLD

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob filho', ))

    print('hello', 'bob pai') ## Note que temos 2 procs!

    p.start()
    p.join()
```

EXEMPLO 0
4X HELLO WORLD

multiprocessing.cpu_count()

```
from multiprocessing import Process

def f(name, id):
    print('hello', name, id)

if __name__ == '__main__':
  procs = []
  for i in range(4):
    p = Process(target=f, args=('bob filho', i, ))
    procs.append(p)

  print('hello', 'bob pai')
  for i in range(4):
    procs[i].start()
  for i in range(4):
    procs[i].join()
```

EXERCÍCIO

Transforme o programa Pi em paralelo!

PI PARALELO

```
from multiprocessing import Process, Queue
import time
PROCS = 2
def pi(start, end, step):
    print "Start: " + str(start)
    print "End: " + str(end)
    sum = 0.0
    for i in range(start, end):
        x = (i+0.5) * step
        sum = sum + 4.0/(1.0+x*x)
    print(sum)
if __name__ == "__main__":
    num_steps = 100000000 #100.000.000
    sum = 0.0
    step = 1.0/num_steps
    proc_size = num_steps / PROCS

    tic = time.time()
    workers = []
    for i in range(PROCS):
        worker = Process(target=pi, args=(i*proc_size, (i+1)*proc_size - 1, step, ))
        workers.append(worker)
    for worker in workers :
        worker.start()
    for worker in workers :
        worker.join()
    toc = time.time()

    print "Pi: %.8f s" %(toc-tic)
```

PI PARALELO

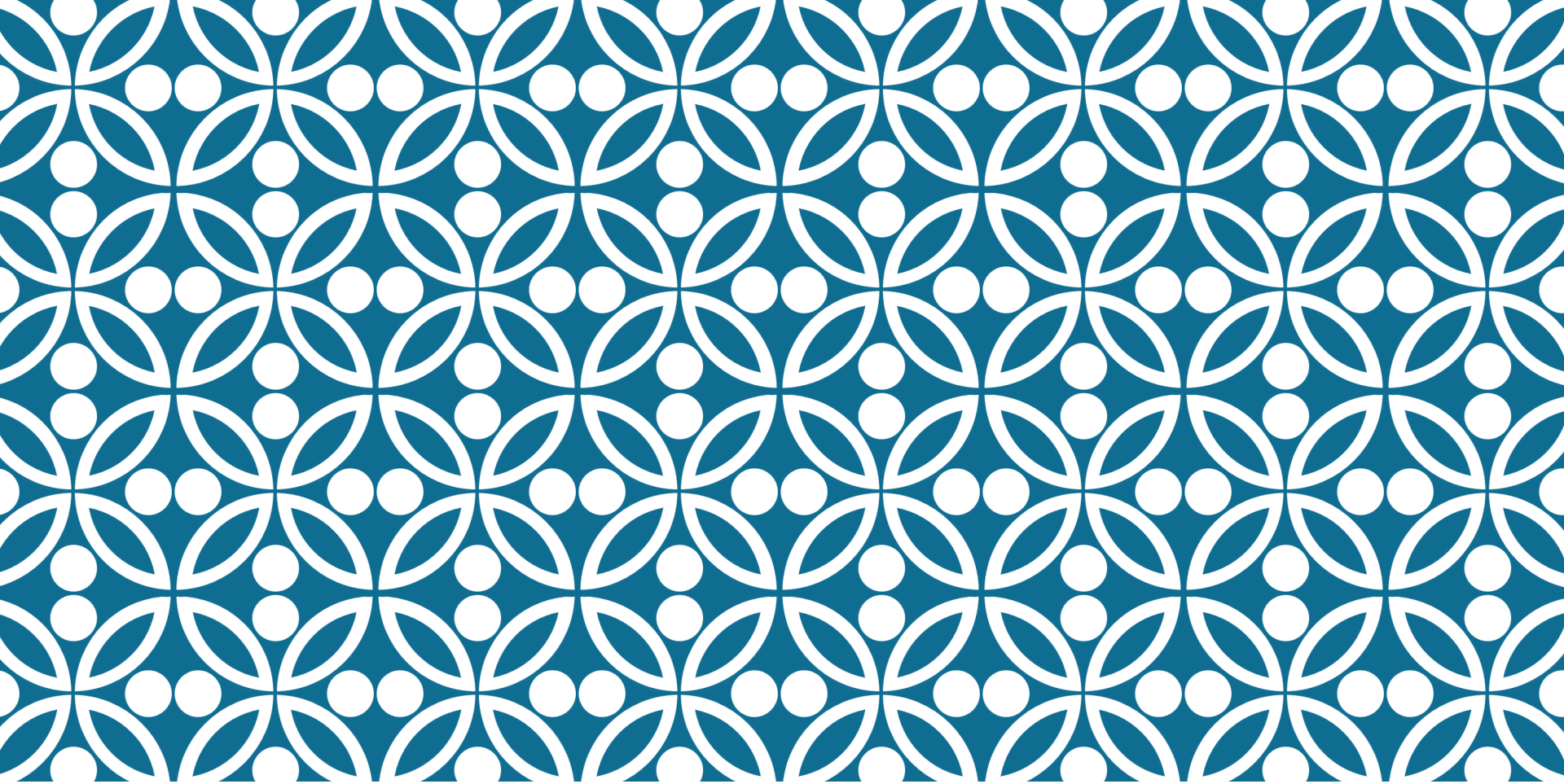
```
from multiprocessing import Process, Queue
import time
PROCS = 2
def pi(start, end, step):
    print "Start: " + str(start)
    print "End: " + str(end)
    sum = 0.0
    for i in range(start, end):
        x = (i+0.5) * step
        sum = sum + 4.0/(1.0+x*x)
    print(sum)
if __name__ ==
    num_steps
    sum = 0.0
    step = 1.0
    proc_size

    tic = time.time()
    workers = []
    for i in range(PROCS):
        worker = Process(target=pi, args=(i*proc_size, (i+1)*proc_size - 1, step, ))
        workers.append(worker)
```

```
    for worker in workers :
        worker.start()
    for worker in workers :
        worker.join()
    toc = time.time()

    print "Pi: %.8f s" %(toc-tic)
```

Fácil, agora é só somar os resultados da saída. Depois múltiplicar por step.



TROCANDO DADOS ENTRE PROCESSOS

EXEMPLO 1
PROCESSOS NÃO COMPARTILHAM MEMÓRIA!

```
from multiprocessing import Process, current_process
import itertools
ITEMS = [1, 2, 3, 4, 5, 6]

def worker(items) :
  for i in itertools.count():
    try :
      items.pop()
    except IndexError :
      break
  print current_process().name, "processed %i items . " %i

if __name__ == "__main__" :
  workers = [ Process (target = worker, args = (ITEMS, ) ) for i in range (3) ]
  for worker in workers :
    worker.start()
  for worker in workers :
    worker.join()
  print "ITEMS after all workers finished : " , ITEMS
```

Saída:
Process 1 processed 6 items .
Process 2 processed 6 items .
Process 3 processed 6 items .
ITEMS after all workers finished : [1, 2, 3, 4, 5, 6]

INTER PROCESS COMMUNICATION (IPC) PIPES E QUEUES

Queue (Fila)

- Multi-consumidor, multi-consumidor FIFO
- Vários processos podem colocar itens na Queue, outros podem obtê-los

Pipe (Tubo)

- Para comunicação entre dois processos
- Um Pipe tem duas extremidades: o processo A escreve algo em sua extremidade do Pipe e o processo B pode lê-lo de sua
- Pipes são bidirecionais

EXEMPLO 2
USANDO MULTIPROCESSAMENTO.QUEUE

```
from multiprocessing import Process, current_process, Queue
import itertools

ITEMS = Queue()
for i in [1, 2, 3, 4, 5, 6, 'end' , 'end' , 'end' ] :
  ITEMS.put(i)

def worker(items) :
  for i in itertools.count() :
    item = items.get()
    if item == 'end':
      break
  print current_process().name, "processed %i items . " %i

if __name__ == "__main__" :
  workers = [ Process ( target=worker , args =(ITEMS, ) ) for i in range(3) ]
  for worker in workers :
    worker.start( )
  for worker in workers :
    worker.join( )
print "#ITEMS after all workers finished : ", ITEMS.qsize( )
```

Saída:
Process 1 processed 1 items .
Process 2 processed 5 items .
Process 3 processed 0 items .
#ITEMS after all workers finished : 0

EXERCÍCIO

Transforme o programa Pi em paralelo + QUEUE !

PIPES

Um pipe tem duas extremidades: a, b = Pipe()

Um processo envia algo para um lado e o outro processo pode receber do outro

recv irá bloquear se o pipe estiver vazio

Fato engraçado

As filas são implementadas usando Pipes e bloqueios.

EXEMPLO 3: USANDO MULTIPROCESSAMENTO PIPE

```
from multiprocessing import Process , Pipe

def worker(conn) :
  while True :
    item = conn.recv()
    if item == 'end' :
      break
    print item
def master(conn) :
  conn.send(' Is')
  conn.send(' this')
  conn.send(' on?')
  conn.send('end')
if __name__ == '__main__' :
  a, b = Pipe()
  w = Process(target = worker, args = (a, ) )
  m = Process(target = master, args = (b, ) )
  w.start()
  m.start()
  w.join()
  m.join()
```

Saída:
Is
this
on?

EXEMPLO 3:

```
from multiprocessing import Process , Pipe
def worker(conn) :
  while True :
    item = conn.recv()
    if item == 'end' :
      break
    print item
  conn.send('thanks')
def master(conn) :
  conn.send(' Is')
  conn.send(' this')
  conn.send(' on?')
  conn.send('end')
  item = conn.recv()
  print item
if __name__ == '__main__' :
  a, b = Pipe()
  w = Process(target = worker, args = (a, ) )
  m = Process(target = master, args = (b, ) )
  w.start()
  m.start()
  w.join()
  m.join()
```

Saída:
Is
this
on?
Thanks

EXERCÍCIO

Transforme o programa Pi em paralelo + PIPES!

OK, UP HERE!

17.2.1.4. Synchronization between processes

17.2.1.5. Sharing state between processes

17.2.1.6. Using a pool of workers

17.2.2.3. Miscellaneous

- `multiprocessing.current_process()`

17.2.2.5. Synchronization primitives