

Python para HPC

Guilherme Magalhães Gall
gmgall@gmail.com

Arquivos do curso

```
$ # Logando no Santos Dumont  
$ ssh usuario@login.sdumont.lncc.br  
  
$ # Para clonar os exemplos do curso  
$ git clone https://github.com/gmgall/python-hpc.git  
$ cd python-hpc
```

Introdução geral à HPC

- "High Performance Computing (HPC) refere-se geralmente à prática de agregar poder computacional numa forma que entregue muito mais desempenho do que o obtível de um desktop ou workstation a fim de resolver problemas complexos em ciência, engenharia e negócios."
- Na maioria dos casos, um sistema de computação de alto desempenho é um *cluster* de *nós de computação*
- Problemas complexos são divididos em subunidades independentes executadas em paralelo

Introdução geral à HPC

- Especialmente em ciência, a demanda por poder computacional tem crescido rapidamente com a necessidade de analisar e explorar grandes massas de dados
- Exemplos:
 - Modelos climáticos recentes geram 8TB para uma simulação de 100 anos e resolução de 100Km e 8PB para resolução de 3Km
 - O LHC gera 10PB de dados brutos por dia e 15PB de dados tratados por ano

Introdução geral à HPC

- Meramente agrupar várias máquinas numa rede não constitui um *cluster*
- É necessário um software de gerência de recursos para
 - Alocar acesso à recursos (nós de computação)
 - Consultar status de recursos
 - Prover um framework para iniciar, executar e monitorar tarefas no conjunto de recursos alocados
 - Arbitrar requisições conflitantes por recursos através da gerência de uma fila de tarefas pendentes

Fonte: [SLURM User's guide](#) da Bull

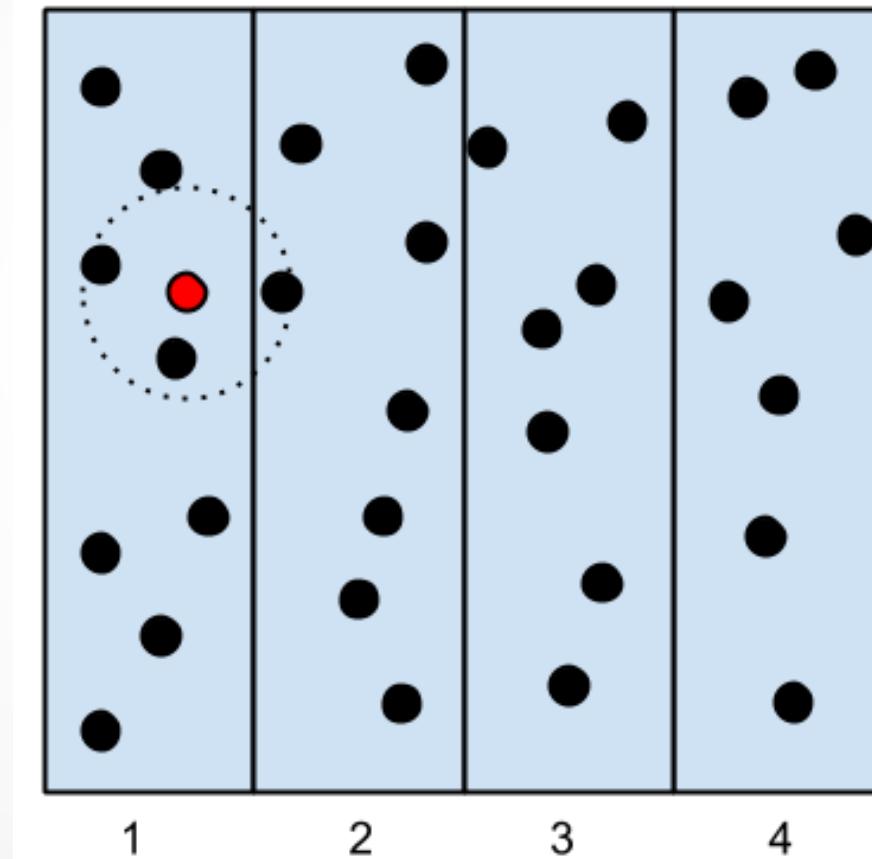
Processamento em memória compartilhada

- Para paralelizar um programa, temos que dividi-lo em subunidades que podem ser executadas independentemente (ou quase)
- Problemas em que as subunidades são completamente independentes são chamados **embaraçosamente paralelos**
 - Exemplo: aplicar uma operação a cada elemento de um array

Processamento em memória compartilhada

- Outros problemas podem ser divididos em subunidades mas precisam compartilhar dados para executarem
 - A implementação é mais difícil
 - Problemas de desempenho podem aparecer devido aos custos de comunicação
 - Exemplo: um simulador de partículas em que as partículas dentro de uma determinada distância se atraem

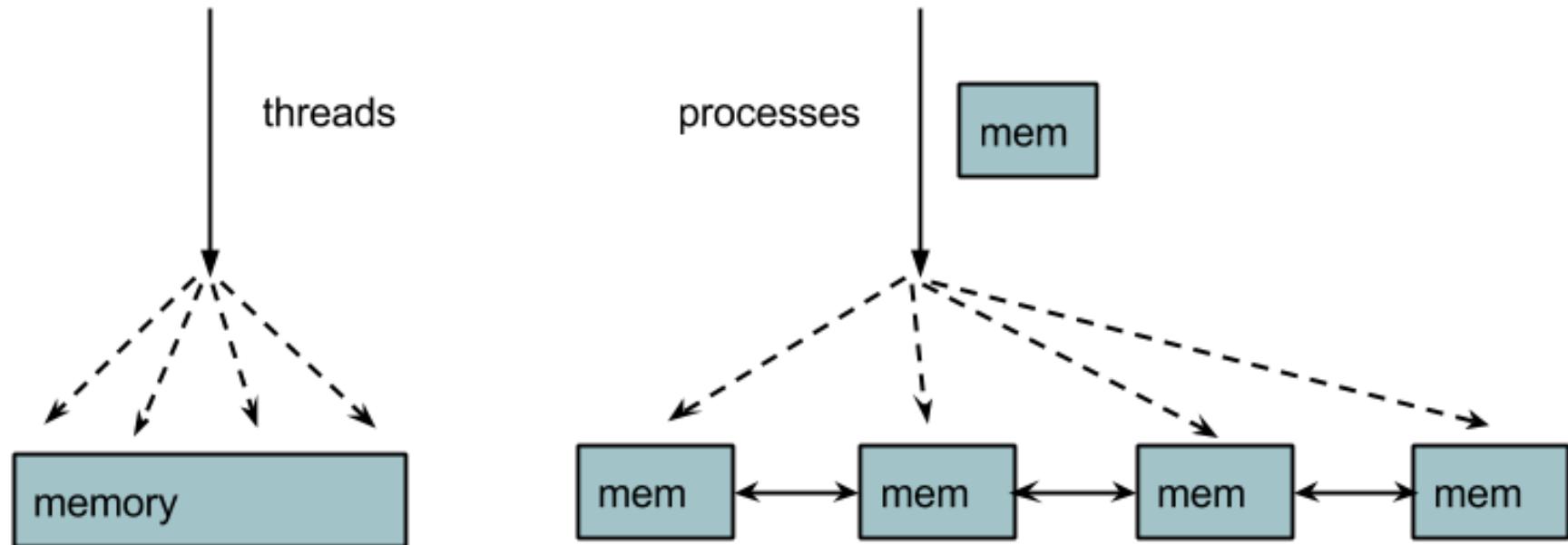
Processamento em memória compartilhada



Processamento em memória compartilhada

- Comunicação entre processos pode acontecer de 2 formas principais
 - **Memória compartilhada**
 - Memória distribuída
- Em memória compartilhada, as subunidades têm acesso ao mesmo espaço de memória
 - Problemas aparecem quando múltiplos processos tentam acessar e mudar a mesma localização na memória ao mesmo tempo
 - Técnicas de sincronização devem ser usadas

Processamento em memória compartilhada



Processamento em memória compartilhada

- Threads de um mesmo processo compartilham o mesmo espaço de memória
- threading
- GIL (Global Interpreter Lock)
= problemas em programas *CPU bound*
- Processos executam em espaços de memória distintos
- multiprocessing
- Comunicação explícita

threading e multiprocessing possuem uma interface muito similar

Módulo multiprocessing

- Crie uma subclasse de `multiprocessing.Process`
- Extenda o método `__init__` para inicializar recursos
- Escreva o código que será executado pelo subprocesso implementando o método `Process.run`
- No código seguinte, é definida uma classe `Processo` que vai esperar 1 segundo e imprimir um id atribuído a ele.

Módulo multiprocessing

```
import multiprocessing
import time

class Processo(multiprocessing.Process):
    def __init__(self, id):
        super(Processo, self).__init__()
        self.id = id

    def run(self):
        time.sleep(1)
        print("Sou o processo com o ID: {}".format(self.id))
```

Módulo multiprocessing

- Para disparar o processo, crie uma instância de Processo
- Chame Processo.start
- Note que Processo.run **não é chamado diretamente**
- Processo.start criará um novo processo, que por sua vez executará o método Processo.run

Módulo multiprocessing

- Adicione o trecho abaixo ao código no slide anterior para testar a criação de um subprocesso

```
if __name__ == '__main__':
    p = Processo(0)
    p.start()
    print("Sou o processo mestre")
```

- Esse código está disponível em multiprocessing1.py

Módulo multiprocessing

- Instruções após Processo.start serão executadas imediatamente sem aguardar que o processo p termine
- Se desejar aguardar o término do processo p, use o método Processo.join conforme o trecho abaixo

```
if __name__ == '__main__':
    p = Processo(0)
    p.start()
    p.join()
    print("Sou o processo mestre")
```

- Esse código está disponível em multiprocessing2.py

Módulo multiprocessing

- Vamos criar 4 subprocessos, conforme o trecho abaixo

```
if __name__ == '__main__':
    processes = Processo(1), Processo(2), Processo(3), Processo(4)
    [p.start() for p in processes]
    print("Sou o processo mestre")
```

- Esse código está disponível em multiprocessing3.py
- Observe que a ordem de execução dos processos é imprevisível. Teste isso executando mais de uma vez multiprocessing3.py

Módulo multiprocessing

- Execute com o comando time do Unix para medir o tempo de execução

```
$ time python3 multiprocessing3.py
Sou o processo mestre
Sou o processo com ID: 1
Sou o processo com ID: 2
Sou o processo com ID: 4
Sou o processo com ID: 3

real    0m1.059s
user    0m0.048s
sys     0m0.004s
```

Módulo multiprocessing

- O módulo `multiprocessing` expõe uma interface conveniente para atribuir tarefas à um conjunto de processos na classe `multiprocessing.Pool`
- `multiprocessing.Pool` dispara um conjunto de processos **workers** e nos permite submeter tarefas à eles com os métodos:
 - `apply` / `apply_async`
 - `map` / `map_async`

Módulo multiprocessing

- O método Pool.map aplica uma função a cada elemento de uma lista e retorna a lista de resultados
- Uso parecido com a função built-in map
- Para usar Pool.map, primeiro é preciso inicializar

```
pool = multiprocessing.Pool()  
pool = multiprocessing.Pool(processes=4)
```

- O *default* é inicializar 1 worker por core

Módulo multiprocessing

- Suponha que você tenha uma função que calcule o quadrado de um número...

```
def square(x):  
    return x * x
```

- e deseja aplicar essa função à uma lista de números em paralelo

```
inputs = [0, 1, 2, 3, 4]  
outputs = pool.map(square, inputs)  
print(outputs)
```

- Esse código está disponível em multiprocessing4.py

Módulo multiprocessing

- O método Pool.map_async funciona como Pool.map mas retorna um objetoAsyncResult no lugar do resultado
- Quando é chamado Pool.map a execução do processo principal é **interrompida** até que todos os workers terminem
- Com Pool.map_async, o objetoAsyncResult é retornado **imediatamente** sem bloquear o processo principal
- UsamosAsyncResult.get para obter os resultados

Módulo multiprocessing

- Exemplo

```
pool = multiprocessing.Pool()  
  
inputs = [0, 1, 2, 3, 4]  
outputs_async = pool.map_async(square, inputs)  
outputs = outputs_async.get()
```

- Esse código está disponível em `multiprocessing5.py`

Módulo multiprocessing

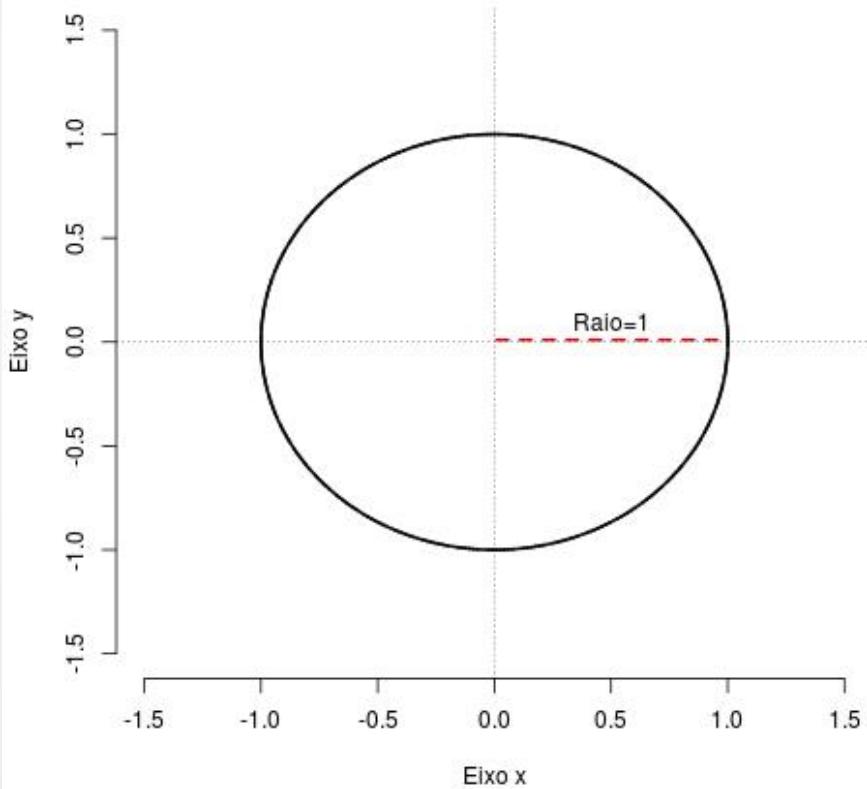
- O método Pool.apply_async atribui uma tarefa que consiste em uma **única função** que é executada por um dos workers
- O retorno é um objetoAsyncResult
- Podemos obter um efeito parecido com map usando apply_async fazendo o seguinte

```
pool = multiprocessing.Pool()

results_async = [pool.apply_async(square, args=(i,)) for i in range(100)]
results = [r.get() for r in results_async]
print(results)
```

- Esse código está disponível em multiprocessing6.py

Exemplo: aproximação de π com Monte Carlo

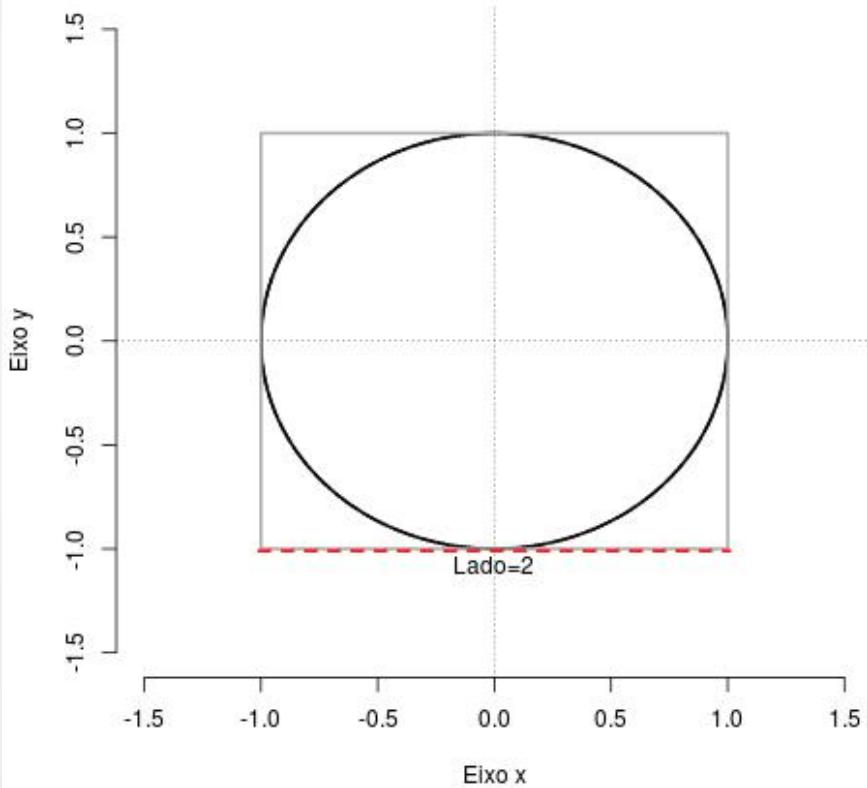


$$A_c = \pi \times r^2$$

$$A_c = \pi \times 1^2$$

$$A_c = \pi$$

Exemplo: aproximação de π com Monte Carlo



$$A_q = l^2$$

$$A_q = 2^2$$

$$A_q = 4$$

Exemplo: aproximação de π com Monte Carlo

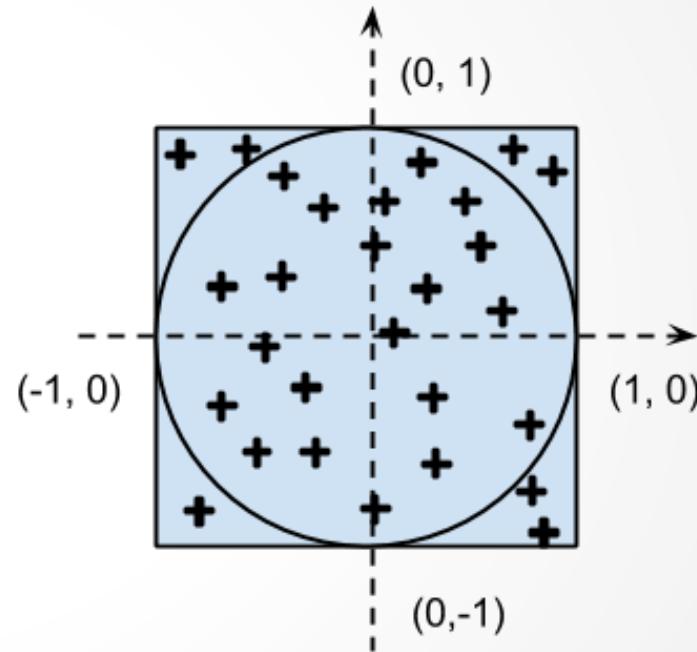
- A razão entre a área do círculo e a área do quadrado nos dará uma proporção que multiplicada por 4 nos dará o valor de π

$$p = \frac{A_c}{A_q} = \frac{\pi}{4}$$

$$\pi = 4 \times p$$

Exemplo: aproximação de π com Monte Carlo

- Para encontrarmos essa proporção, distribuiremos vários pontos aleatórios dentro do quadrado e contaremos quantos caíram dentro do círculo
- Depois diviremos a quantidade de pontos dentro do círculo dentro pelo total de pontos



Exemplo: aproximação de π com Monte Carlo

- Criando pontos aleatórios

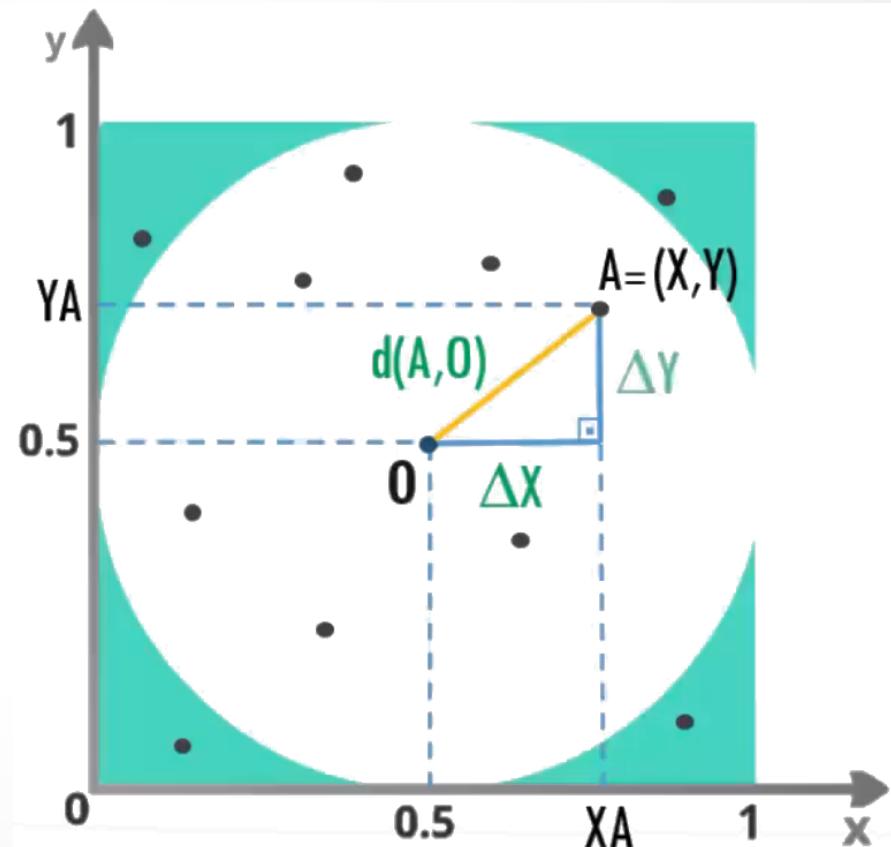
```
import random

x = random.uniform(-1.0, 1.0)
y = random.uniform(-1.0, 1.0)
```

Exemplo: aproximação de π com Monte Carlo

- Testando se estão dentro do círculo

```
if x**2 + y**2 <= 1:  
    hits += 1
```



Exemplo: aproximação de π com Monte Carlo

- Começando com uma versão serial (`pi_serial.py`)

```
import random

total = 1000000
dentro = 0

for i in range(total):
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)

    if x**2 + y**2 <= 1:
        dentro += 1

pi = 4.0 * dentro/total

print("O valor aproximado de pi é {}".format(pi))
```

Exemplo: aproximação de π com Monte Carlo

- Cada iteração do loop é independente dos demais – é um problema *embarrasosamente paralelo*
- Para paralelizar esse código criaremos uma função `ponto_aleatorio`, que corresponde à checagem de um único ponto
- Essa função retornará 1 se o ponto está dentro do círculo e 0 se não estiver
- Executando `ponto_aleatorio` múltiplas vezes e somando os resultados temos o total de pontos dentro do círculo

Exemplo: aproximação de π com Monte Carlo

- A função ponto_aleatorio

```
def ponto_aleatorio():
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)

    if x**2 + y**2 <= 1:
        return 1
    else:
        return 0
```

Exemplo: aproximação de π com Monte Carlo

- Chamando ponto_aleatorio em múltiplos processadores e somando os resultados

```
pool = multiprocessing.Pool()
results_async = [pool.apply_async(ponto_aleatorio) for i in range(total)]
dentro = sum(r.get() for r in results_async)

pi = 4.0 * dentro/total

print("O valor aproximado de pi é {}".format(pi))
```

- Esse código está disponível em pi_apply_async.py

Exemplo: aproximação de π com Monte Carlo

- Nossa versão paralela demora bem mais que a serial

```
$ cat slurm-157698.out
0 valor aproximado de pi é 3.142528

real    0m1.958s
user    0m1.722s
sys     0m0.071s
```

```
$ cat slurm-157699.out
0 valor aproximado de pi é 3.140372

real    2m20.211s
user    3m32.486s
sys     0m28.166s
```

- O tempo gasto fazendo o cálculo é pequeno comparado ao tempo necessário para enviar e distribuir as tarefas para os workers

Exemplo: aproximação de π com Monte Carlo

- Para resolver esse problema colocaremos cada worker para processar mais de um ponto ao mesmo tempo fazendo esse *overhead* de comunicação insignificante
- Começamos definindo uma função `multiplos_pontos` que processa mais de um ponto...

```
def multiplos_pontos(qtd_pontos):  
    return sum(ponto_aleatorio() for i in range(qtd_pontos))
```

Exemplo: aproximação de π com Monte Carlo

- ...e modificamos nosso programa dividindo nosso problema por 10 tarefas, mais intensivas no uso da CPU

```
tamanho_tarefa = total//n_tarefas

pool = multiprocessing.Pool()
results_async = [pool.apply_async(multiplos_pontos, args=(tamanho_tarefa,))  
    for i in range(n_tarefas)]
dentro = sum(r.get() for r in results_async)

pi = 4.0 * dentro/total
```

- Esse código está disponível em `pi_apply_async_chunked.py`

Exemplo: aproximação de π com Monte Carlo

- Agora nossa versão paralela é mais rápida que a serial

```
$ cat slurm-157698.out
0 valor aproximado de pi é 3.142528

real    0m1.958s
user    0m1.722s
sys     0m0.071s
```

```
$ cat slurm-157701.out
0 valor aproximado de pi é 3.13836

real    0m0.713s
user    0m1.887s
sys     0m0.403s
```

- Tudo é simples ao lidar com problemas *embarrasosamente paralelos*. Entretanto, eventualmente é necessário compartilhar informações entre os processos.

Módulo multiprocessing: locks e sincronização

- O módulo `multiprocessing` permite definir variáveis numa área de memória compartilhada
- Defina uma variável compartilhada usando `multiprocessing.Value` e defina seu tipo como uma string
- Para modificar o conteúdo da variável, use o atributo `value`
- Exemplo:

```
shared_variable = multiprocessing.Value('f')
shared_variable.value = 0
```

Módulo multiprocessing: locks e sincronização

- Ao usar memória compartilhada, é necessário ficar atento à possibilidade de acessos concorrentes
- Imagine que você possui uma variável inteira compartilhada e que vários processos a incrementem várias vezes
- Imagine um processo como o seguinte:

```
class Processo(multiprocessing.Process):  
  
    def __init__(self, contador):  
        super(Processo, self).__init__()  
        self.contador = contador  
  
    def run(self):  
        for i in range(1000):  
            self.contador.value += 1
```

Módulo multiprocessing: locks e sincronização

- E no programa principal:

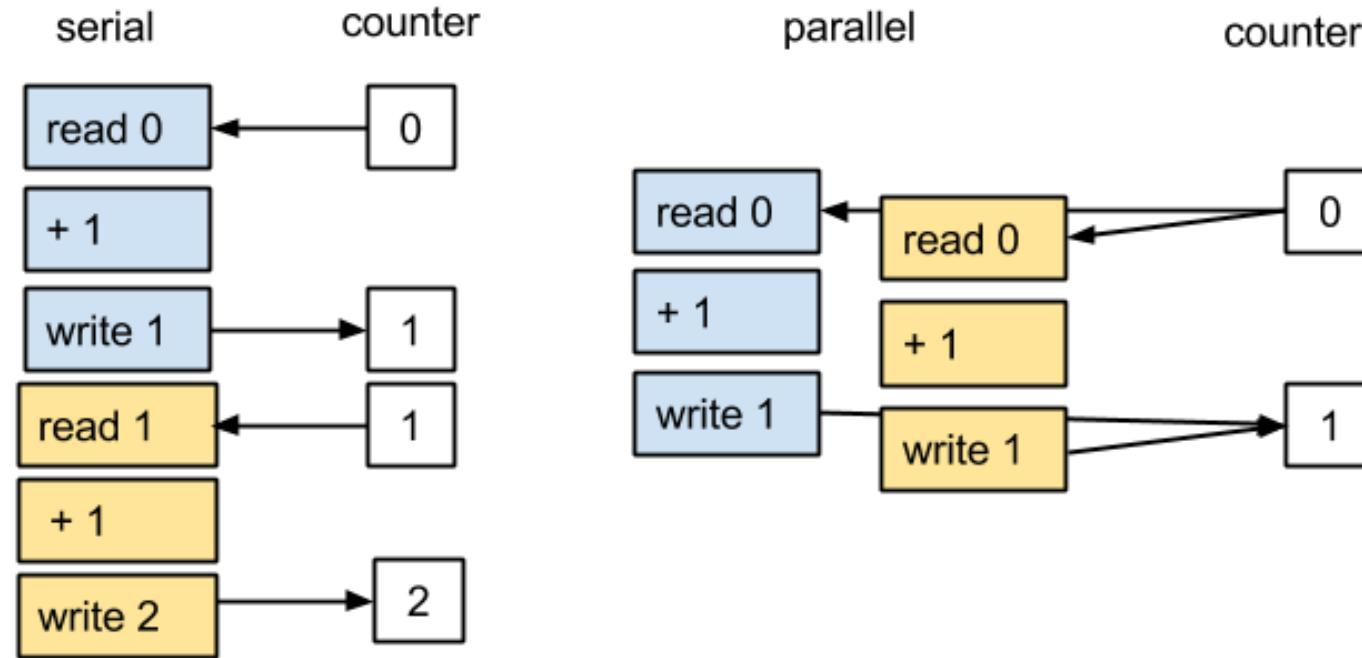
```
var_compartilhada = multiprocessing.Value('i')
var_compartilhada.value = 0

processos = [Processo(var_compartilhada) for i in range(4)]
[p.start() for p in processos]
[p.join() for p in processos]
print(var_compartilhada.value)
```

- A saída esperada é 4000 (4 processos incrementando a variável 1000 vezes), mas o resultado é um número aleatório
- Esse código está disponível em shared1.py

Módulo multiprocessing: locks e sincronização

- Ocorreram incrementos concorrentes da variável



Módulo multiprocessing: locks e sincronização

- Para resolver esse problema, devemos controlar o acesso a essa variável de forma que **apenas um processo por vez** possa lê-la, incrementá-la e escrevê-la
- Isso pode ser feito por uma *lock* fornecida por `multiprocessing.Lock`
- Uma vez que um processo adquiriu a *lock*, tentativas subsequentes de a adquirir vão bloquear até que a lock seja liberada

Módulo multiprocessing: locks e sincronização

- Definindo uma *lock* global e usando-a para restringir o acesso à contador

```
lock = multiprocessing.Lock()

class Processo(multiprocessing.Process):

    def __init__(self, contador):
        super(Processo, self).__init__()
        self.contador = contador

    def run(self):
        for i in range(1000):
            with lock: #adquire a lock
                self.contador.value += 1
            # libera a lock
```

Módulo multiprocessing: locks e sincronização

- O código do slide anterior está disponível em shared2.py
- Essa nova versão retorna o valor esperado:

```
$ python3.5 shared2.py  
4000
```

- Mais primitivas de sincronização em
<https://docs.python.org/3/library/multiprocessing.html>

Processamento em memória distribuída

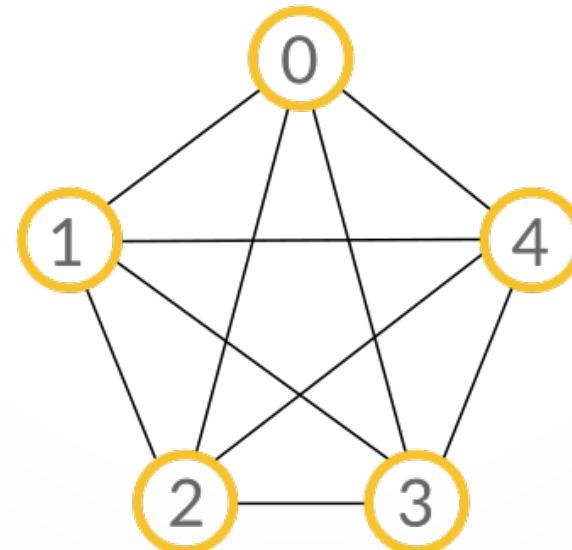
- Comunicação entre processos pode acontecer de 2 formas principais
 - Memória compartilhada
 - **Memória distribuída**
- Cada processo está completamente separado dos demais e cada um possui seu próprio espaço de memória
 - Podem estar inclusive em máquinas diferentes
- A comunicação entre os processos é feita explicitamente e os dados podem trafegar pela rede

MPI

- MPI (Message Passing Interface) é um padrão para comunicação de dados em computação paralela
- Nele, uma aplicação é constituída por uma ou mais tarefas que se comunicam através da chamada de funções para o recebimento e envio de mensagens
- MPI é apenas um padrão, uma definição de uma interface
- Existem diversas implementações dessa definição (OpenMPI, MVAPICH, MPICH)

MPI: communicators e ranks

MPI_COMM_WORLD



mpi4py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print("This is process", rank)
```

- Esse código está disponível em `mpi1.py`

mpi4py

- Todos os processos recebem o mesmo código, mas podemos definir instruções diferentes para cada processo pelo *rank* (`mpi2.py`)

```
a = 6.0
b = 3.0
if rank == 0:
    print(a + b)
if rank == 1:
    print(a * b)
if rank == 2:
    print(a - b)
if rank == 3:
    print(a / b)
if rank == 4:
    print(a ** b)
```

mpi4py: comunicação ponto a ponto

- A comunicação mais simples possível se dá entre um processo *sender* e um *receiver* (mpi3.py)

```
numero = numpy.zeros(1)

if rank == 1:
    numero = numpy.random.random_sample(1)
    print("O processo {} sorteou o número {}.".format(rank, numero[0]))
    comm.Send(numero, dest=0)

if rank == 0:
    print("O processo {} tem o número {} antes de receber o número sorteado."
          .format(rank, numero[0]))
    comm.Recv(numero, source=1)
    print("O processo {} recebeu o número {}.".format(rank, numero[0]))
```

mpi4py + numpy

- Uma breve digressão para explicar porque usamos numpy no exemplo anterior
- mpi4py suporta comunicações de objetos Python genéricos pelo módulo `pickle` de serialização...
 - métodos cujos nomes são todos em minúsculas como send, recv e bcast
- e comunicações de objetos *buffer-like*
 - métodos cujos nomes começam com uma letra maiúscula como Send, Recv e Bcast
- Mais sobre isso em
<http://mpi4py.scipy.org/docs/mpi4py.pdf#subsection.2.1>

mpi4py + numpy

- Objetos *buffer-like* (como os arrays do numpy) oferecem
 - comunicação muito rápida
 - a funcionalidade de descoberta automática de tipo
- numpy é o padrão *de facto* para computação científica no Python
- Permite trabalhar com vetores e matrizes de forma semelhante ao software proprietário MATLAB

mpi4py + numpy

- Funcionalidades do numpy que usaremos nos exemplos seguintes

```
>>> # Criação de um array com 1 posição, cada um com número aleatório
>>> numpy.random.random_sample(1)
array([ 0.68869191])

>>> # Criação de um array com 10 posições, cada um com número aleatório
>>> # variando de -1 a 1
>>> numpy.random.uniform(-1, 1, 10)
array([-0.02849434,  0.8331388 , -0.35301915,  0.90242583, -0.51152935,
       0.8343597 ,  0.95781367, -0.37609906,  0.24319912,  0.48297868])

>>> # Criação de um array com 1 posição, cada uma com 0
>>> numpy.zeros(1)
array([ 0.])
```

mpi4py + numpy

- Funcionalidades do numpy que usaremos nos exemplos seguintes

```
>>> # Operações vetoriais

>>> x = numpy.random.uniform(-1, 1, 10)
>>> y = numpy.random.uniform(-1, 1, 10)
>>> x
array([ 0.02424211,  0.2964557 , -0.10675464,  0.2455312 ,  0.69424796,
        0.28536884,  0.75992597,  0.27647851, -0.55414027, -0.99421686])
>>> y
array([-0.53858176, -0.29274091, -0.48850674, -0.66673882,  0.74072266,
        0.06370495,  0.77738873,  0.70036948, -0.78910735, -0.51626293])
>>>
>>> dentro = x**2 + y**2 <= 1
>>> dentro
array([ True,  True,  True,  True, False,  True,  True, False]
      , dtype=bool)
>>> dentro.sum()
```

mpi4py: comunicação ponto a ponto

- De volta ao nosso exemplo (mpi3.py)

```
numero = numpy.zeros(1)

if rank == 1:
    numero = numpy.random.random_sample(1)
    print("O processo {} sorteou o número {}.".format(rank, numero[0]))
    comm.Send(numero, dest=0)

if rank == 0:
    print("O processo {} tem o número {} antes de receber o número sorteado."
          .format(rank, numero[0]))
    comm.Recv(numero, source=1)
    print("O processo {} recebeu o número {}.".format(rank, numero[0]))
```

mpi4py: comunicação ponto a ponto

- Send e Recv são funções bloqueantes
- Se um processo chama Recv, ele ficará parado até receber uma mensagem de um Send correspondente
- Só então o processo continuará
- As versões não bloqueantes de Send e Recv são Isend e Irecv, respectivamente
- Numa chamada a Recv não é necessário especificar uma origem (source), é possível aceitar mensagens de qualquer processo que estiver enviando (source=MPI.ANY_SOURCE)

mpi4py: comunicação ponto a ponto

- Isend e Irecv retornam um objeto da classe Request, **imediatamente**
- O objeto identifica unicamente a operação iniciada
- Request.Wait aguarda até a comunicação ser completada
- Request.Test testa se a comunicação foi completada (retorna True ou False)
- Request.Cancel cancela uma comunicação pendente

mpi4py: comunicação ponto a ponto

- Reimplementando o exemplo anterior com comunicação não bloqueante (mpi4.py)

```
numero = numpy.zeros(1)

if rank == 1:
    numero = numpy.random.random_sample(1)
    print("O processo {} sorteou o número {}.".format(rank, numero[0]))
    requisicao = comm.Isend(numero, dest=0)
    requisicao.Wait()

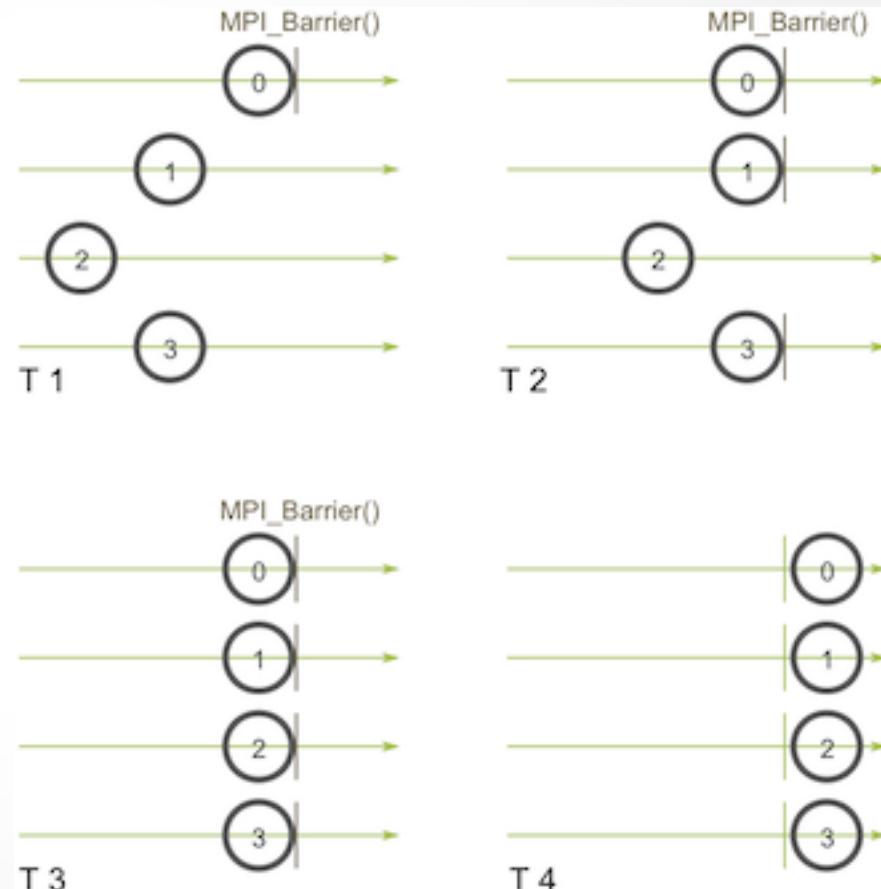
if rank == 0:
    print("O processo {} tem o número {} antes de receber o número sorteado."
          .format(rank, numero[0]))
    requisicao = comm.Irecv(numero, source=1)
    requisicao.Wait()
    print("O processo {} recebeu o número {}.".format(rank, numero[0]))
```

mpi4py: comunicação coletiva

- Operações de comunicação coletiva permitem a transmissão de dados entre múltiplos processos de um grupo
- As operações de comunicação coletiva mais comuns são
 - Sincronização
 - Funções de comunicação global
 - Reduções
 - Entrada/Saída coletiva

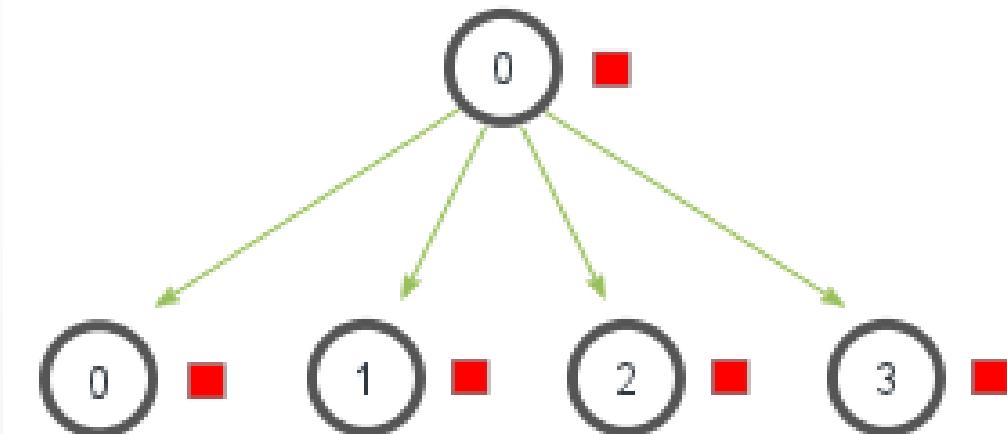
mpi4py: sincronização (Comm.Barrier)

- `Comm.Barrier` forma uma barreira, de forma que nenhum processo no *communicator* possa ultrapassá-la até que todos os outros processos chamem `Comm.Barrier`



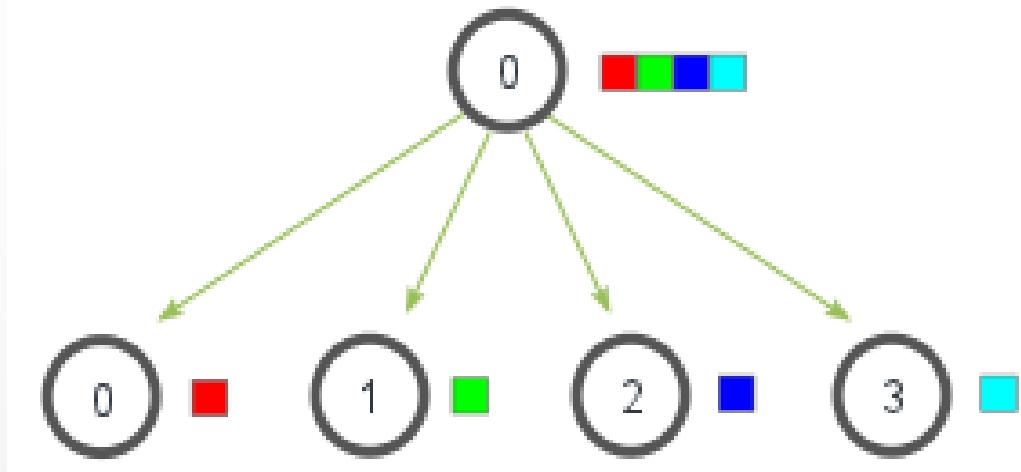
mpi4py: comunicação global (Comm.Bcast)

- Durante um *broadcast*, um processo manda a mesma informação para todos os processos num *communicator*
- Apesar do processo root e os receptores fazerem trabalhos diferentes, todos chamam a mesma Comm.Bcast



mpi4py: comunicação global (Comm.Scatter)

- Similar à um *broadcast*, um processo manda informação para todos os processos num *communicator*
- Mas um *broadcast* manda a mesma informação e um *scatter* manda partes diferentes de um array

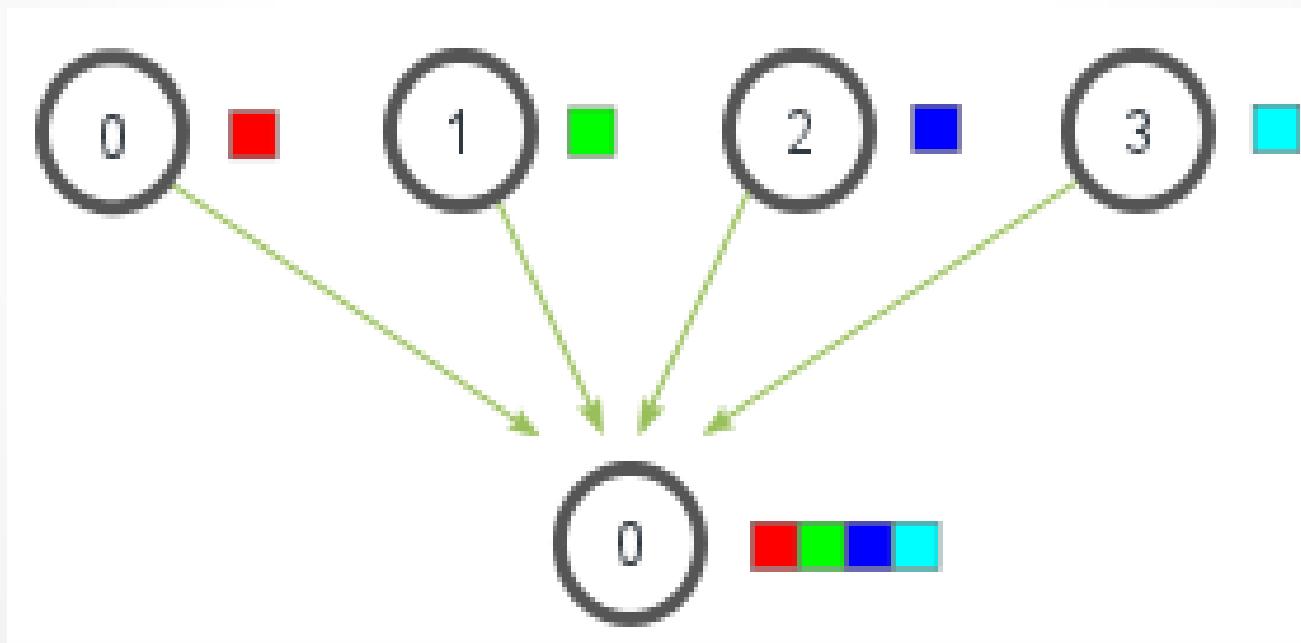


mpi4py: comunicação global (Comm.Scatter)

- Comm.Scatter recebe 3 argumentos
 - O array que reside no processo root
 - Onde guardar a informação recebida
 - Indica o processo que está distribuindo os dados (root)

mpi4py: comunicação global (Comm.Gather)

- Operação inversa a uma operação de *scatter*
- Comm.Gather recebe os mesmos parâmetros que Comm.Scatter



Exemplo: média multiprocesso

- Geraremos um array de números no processo 0
- Distribuiremos os números por todos os processos, cada um recebendo um número igual de números
 - via Comm.Scatter
- Cada processo calculará a média dos números que receber
- O processo 0 receberá essas médias parciais e calculará a média final
 - via Comm.Gather
- Se ficou difícil entender a partição dos dados, veja scatter_ex.py

Exemplo: média multiprocessso

- Exemplo disponível em mpi5.py

```
rand_nums = None

if rank == 0:
    rand_nums = np.random.uniform(1, 10, [size, num_elements_per_proc])
    print("Avg of all elements is", rand_nums.mean())

sub_rand_nums = np.zeros(num_elements_per_proc)

comm.Scatter(rand_nums, sub_rand_nums, root=0)
sub_avg = sub_rand_nums.mean()

sub_avgs = None

if rank == 0:
    sub_avgs = np.zeros(size)

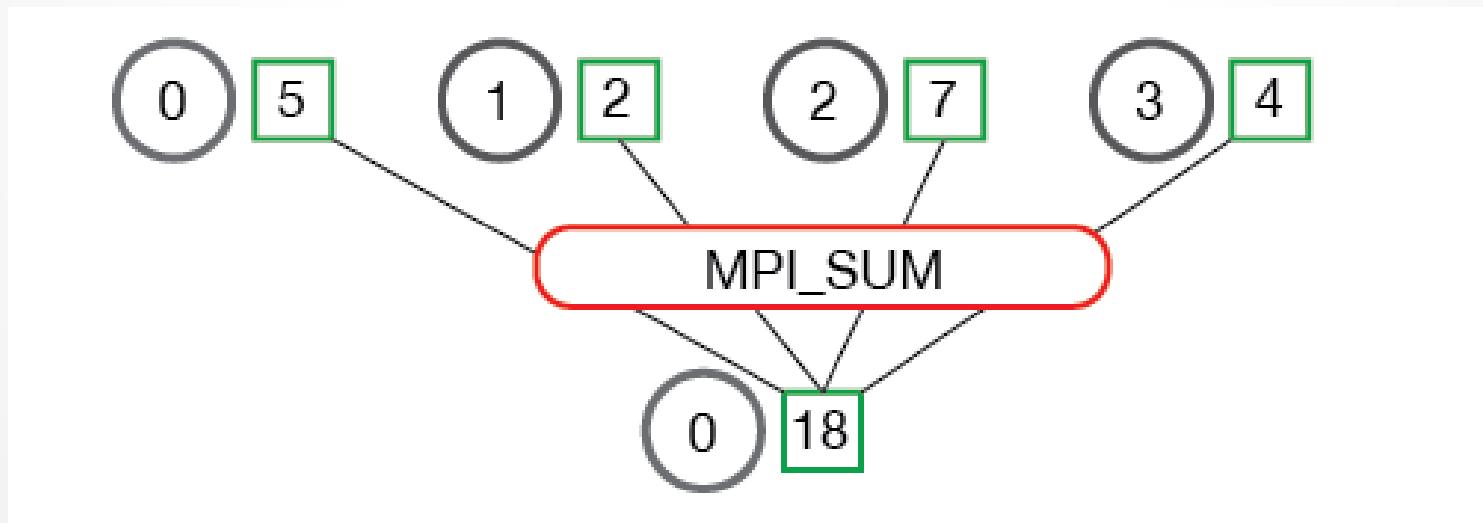
comm.Gather(sub_avg, sub_avgs, root=0)
```

mpi4py: reduções (Comm.Reduce)

- Reduções são conceitos clássicos da programação funcional
- Envolve reduzir um conjunto de números via uma função
- Ex.: supondo uma lista [1, 2, 3, 4, 5] algumas reduções seriam
 - `sum([1, 2, 3, 4, 5]) = 15`
 - `max([1, 2, 3, 4, 5]) = 5`
 - `min([1, 2, 3, 4, 5]) = 1`
 - `prod([1, 2, 3, 4, 5]) = 120`

mpi4py: reduções (Comm.Reduce)

- `Comm.Reduce` recebe um array de elementos de entrada e uma operação de redução e retorna um array de elementos de saída para o processo root



Exemplo: aproximação de π com Monte Carlo

- Vamos reimplementar nossa aproximação de π com MPI e reduções (`pi_mpi.py`)

```
import numpy as np

total = 100000000

n_procs = comm.Get_size()

print("Esse é o processo", rank)

# Create an array
x_parcial = np.random.uniform(-1, 1, int(total/n_procs))
y_parcial = np.random.uniform(-1, 1, int(total/n_procs))

dentro_parcial = x_parcial**2 + y_parcial**2 <= 1
dentro_parcial_soma = dentro_parcial.sum()
```

Exemplo: aproximação de π com Monte Carlo

- Vamos reimplementar nossa aproximação de π com MPI e reduções (pi_mpi.py)

```
print("Contagem parcial de pontos dentro do círculo:", dentro_parcial_soma)
contagem_final = comm.reduce(dentro_parcial_soma, op=MPI.SUM, root=0)

if rank == 0:
    print("Contagem final de pontos dentro do círculo:", contagem_final)
    print("Aproximação de pi:", 4 * contagem_final/total)
```