centro universitário FEI

IAR
Inteligência Artificial Aplicada à Automação e Robótica

Programação Científica

Prof. Dr. Danilo H. Perico

PROGRAMAÇÃO PARALELA COM MPI

Objetivos desta aula

- Introdução à Programação de Alto Desempenho
- Introdução à Programação Paralela
- Programação em MPI

Programação Científica

- ***Computational Science*** *is concerned with constructing mathematical models and quantitative analysis techniques and using computers to analyze and solve scientific problems*
    - *In practical use, it is typically the application of computer simulation and other forms of computation from numerical analysis and theoretical computer science to problems in various scientific disciplines*

Programação de Alto Desempenho

- Uso de HPC para solucionar problemas
- **HPC =** ***High Performance Computing***

Programação de Alto Desempenho

- O que é ***High Performance Computing***?
  - Uso de hardware de alto desempenho
  - Supercomputadores
  - Clusters
  - GPUs
  - Clusters de GPUs

Abordagens para HPC

- A maioria das abordagens para HPC utiliza Supercomputadores ou Clusters
- **Supercomputadores:**
  - Um computador na fronteira da capacidade de processamento e desempenho, geralmente com memória compartilhada
- **Clusters:**
  - Conjunto de computadores conectados por rede de alta velocidade

Supercomputadores

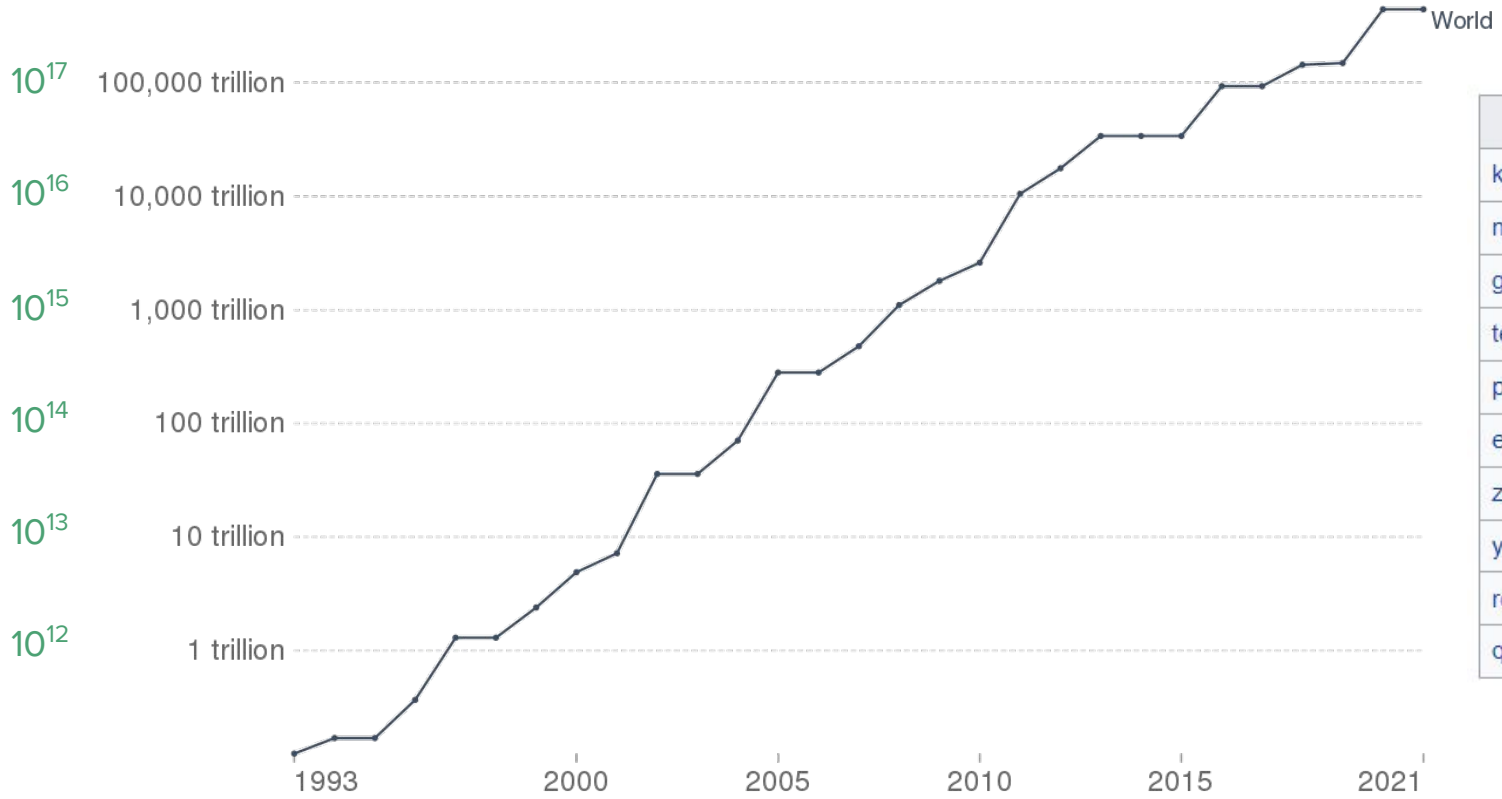
- Computador com um alto nível de desempenho
- O desempenho é comumente medido em ***operações de ponto flutuante por segundo (FLOPS)***
- Desde 2017, existem supercomputadores que podem realizar mais de 10^{17} FLOPS (100 quatrilhões de FLOPS, 100 petaFLOPS ou 100 PFLOPS)
- Para comparação, um desktop tradicional tem desempenho na faixa de centenas de gigaflops (10^{11}) a dezenas de teraflops (10^{13})

Fonte: https://en.wikipedia.org/wiki/Supercomputer

The exponential growth of supercomputers

Number of floating-point operations carried out per second by the largest supercomputer in any given year

Our World in Data



Computer performance

Name	Unit	Value
kiloFLOPS	kFLOPS	10^3
megaFLOPS	MFLOPS	10^6
gigaFLOPS	GFLOPS	10^9
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}
ronnaFLOPS	RFLOPS	10^{27}
quettaFLOPS	QFLOPS	10^{30}

Source: TOP500 Supercomputer Database

Note: Floating-point operations are needed for very large or very small real numbers, or computations that require a large dynamic range. Floating-point operations per second are therefore a more accurate measure than instructions per second.

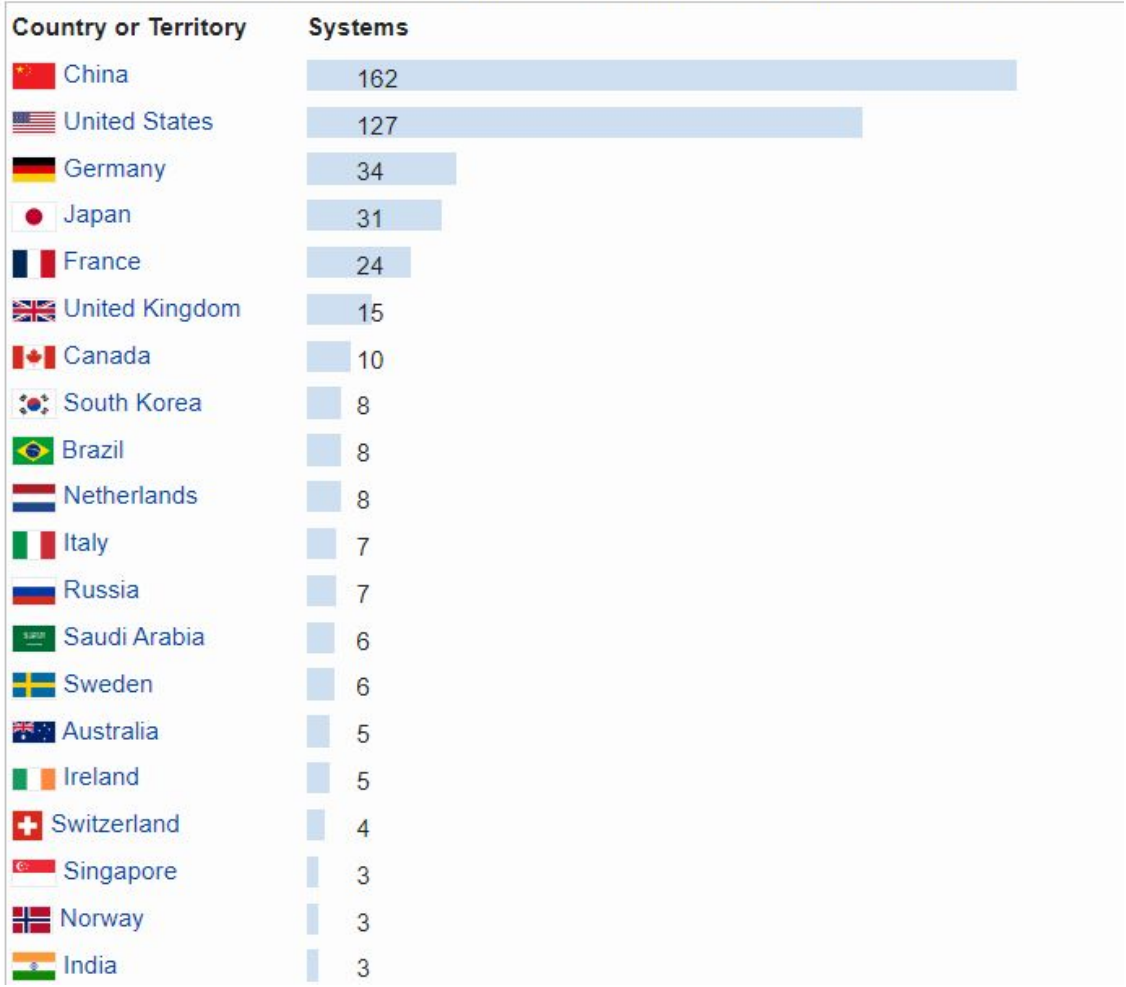
Rank (previous)	Rmax Rpeak (PetaFLOPS)	Name	Model	CPU cores	Accelerator (e.g. GPU) cores	Interconnect	Manufacturer	Site country	Year	Operating system
1 —	1,102.00 1,685.65	Frontier	HPE Cray EX235a	591,872 (9,248 × 64-core Optimized 3rd Generation EPYC 64C @2.0 GHz)	36,992 × 220 AMD Instinct MI250X	Slingshot-11	HPE	Oak Ridge National Laboratory United States	2022	Linux (HPE Cray OS)
2 —	442.010 537.212	Fugaku	Supercomputer Fugaku	7,630,848 (158,976 × 48-core Fujitsu A64FX @2.2 GHz)	0	Tofu interconnect D	Fujitsu	RIKEN Center for Computational Science Japan	2020	Linux (RHEL)
3 —	309.10 428.70	LUMI	HPE Cray EX235a	150,528 (2,352 × 64-core Optimized 3rd Generation EPYC 64C @2.0 GHz)	9,408 × 220 AMD Instinct MI250X	Slingshot-11	HPE	EuroHPC JU European Union, location: Kajaani, Finland.	2022	Linux (HPE Cray OS)
4 NEW	174.70 255.75	Leonardo	BullSequana XH2000	110,592 (3,456 × 32-core Xeon Platinum 8358 @2.6 GHz)	13,824 × 108 Nvidia Ampere A100	Nvidia HDR100 Infiniband	Atos	EuroHPC JU European Union, location: Bologna, Italy.	2022	Linux
5 ▼ (4)	148.600 200.795	Summit	IBM Power System AC922	202,752 (9,216 × 22-core IBM POWER9 @3.07 GHz)	27,648 × 80 Nvidia Tesla V100	InfiniBand EDR	IBM	Oak Ridge National Laboratory United States	2018	Linux (RHEL 7.4)
6 ▼ (5)	94.640 125.712	Sierra	IBM Power System S922LC	190,080 (8,640 × 22-core IBM POWER9 @3.1 GHz)	17,280 × 80 Nvidia Tesla V100	InfiniBand EDR	IBM	Lawrence Livermore National Laboratory United States	2018	Linux (RHEL)
7 ▼ (6)	93.015 125.436	Sunway TaihuLight	Sunway MPP	10,649,600 (40,960 × 260-core Sunway SW26010 @1.45 GHz)	0	Sunway[31]	NRCPC	National Supercomputing Center in Wuxi China[31]	2016	Linux (RaiseOS 2.0.5)

Fonte: https://en.wikipedia.org/wiki/TOP500

#1 - Frontier


FRONTIER
OAK RIDGE
National Laboratory
ENERGY
Hewlett Packard
Enterprise
AMD

Distribution of supercomputers in the TOP500 list by country (as of November 2022)



Supercomputadores

Brasil:

- Pégaso - Petrobras
- Dragão - Petrobras
- Atlas - Petrobras
- IARA - SiDi
- NOBZ1 - MBZ
- Fênix - Petrobras
- A16A - MBZ
- Santos Dumont - LNCC *(Laboratório Nacional de Computação Científica)*

Fonte: https://top500.org/lists/top500/list/2022/11/

Pégaso

Supercomputador Pégaso
(imagem: divulgação/Petrobras)

Pégaso

- 21 petaFLOPS ($21x10^{15}$)
- 678 TB de memória RAM (terabytes)
- 2.016 GPUs
- Link de 400 Gb/s (gigabits por segundo) para conectividade
- 233.856 cores (núcleos de processamento)
- Sistema Operacional: Linux CentOS

Clusters

- É um conjunto de computadores, os quais são ligados em rede e trabalham como se fossem uma única máquina de grande porte
- Os clusters se popularizaram nos anos 1990 por 3 motivos:
  - Microprocessadores de alta performance
  - Redes de alta velocidade
  - Ferramentas para computação distribuída

Tipos de Cluster

- De alto desempenho: permite uma grande carga de processamento
- De alta disponibilidade: resistente a falhas de hardware, software e energia, tem o objetivo de manter os serviços disponibilizados o máximo de tempo possível
- Para balanceamento de carga: esse tipo de cluster tem como função controlar a distribuição equilibrada do processamento

Cluster Beowulf

- O tipo mais simples:
  - Constituído por diversos nós trabalhadores (workers) gerenciados por um só computador (master), geralmente PCs comuns
  - Utiliza rede ethernet or gigabit ethernet
  - Off-the-shelf components
- O maior problema:
  - Comunicação entre os nós

Programando em HPC

- **Como usar os supercomputadores ou os clusters?**
  - Dividindo o trabalho em diversos núcleos de processamento
    - Usar um modelo que divide uma tarefa em várias subtarefas e as executa simultaneamente para aumentar a velocidade e a eficiência
    - **Necessidade de programação paralela ou distribuída**

Computação Paralela x Computação Distribuída

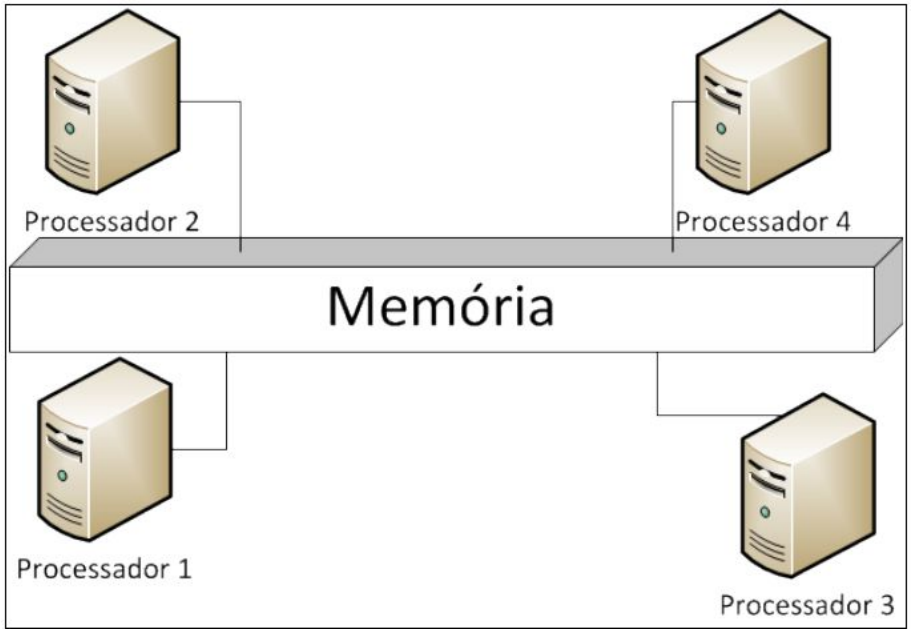
- Na **computação paralela**, todos os processadores podem ter acesso a uma memória compartilhada para trocar informações entre os processadores
- Na **computação distribuída**, cada processador possui sua própria memória privada (memória distribuída). As informações são trocadas pela passagem de mensagens entre os processadores

Computação Paralela x Computação Distribuída

- **Computação paralela** em um único computador usa vários processadores para processar tarefas em paralelo
- **Computação paralela distribuída** usa vários dispositivos de computação (normalmente computadores) para processar tarefas

Computação Paralela x Computação Distribuída

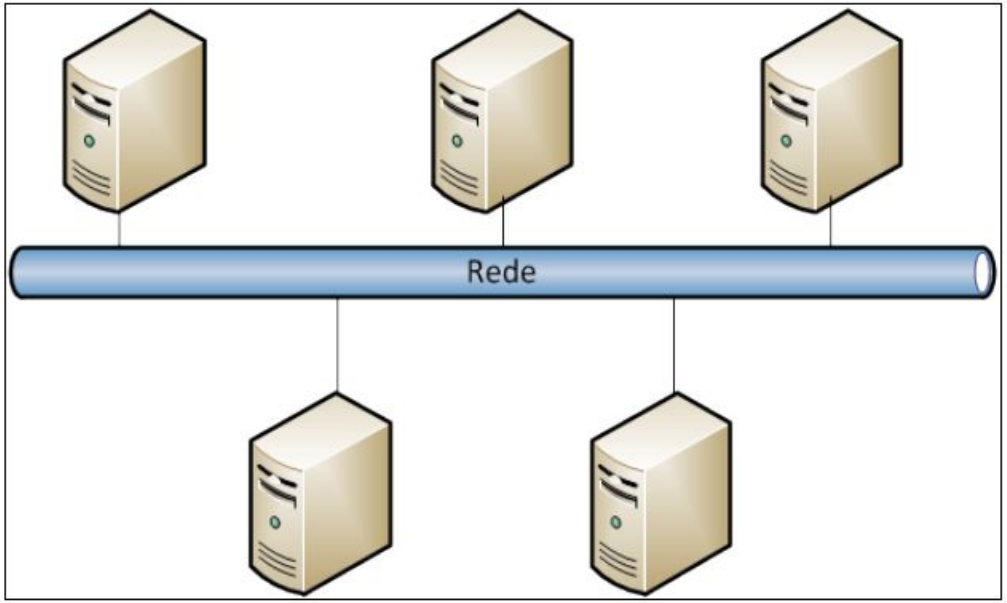
- **Computação paralela com memória compartilhada**



Fonte da imagem: Sandro Athaide Coelho
https://www.ufjf.br/getcomp/files/2013/03/Introdu%C3%A7%C3%A3o-a-Computa%C3%A7%C3%A3o-Paralela-com-o-OpenMPI.pdf

Computação Paralela x Computação Distribuída

- **Computação paralela distribuída**



Fonte da imagem: Sandro Athaide Coelho
https://www.ufjf.br/getcomp/files/2013/03/Introdu%C3%A7%C3%A3o-a-Computa%C3%A7%C3%A3o-Paralela-com-o-OpenMPI.pdf

Temos o hardware,

falta o software

Message Passing Interface - MPI

- **MPI:** é um padrão para comunicação de dados em computação distribuída
- É uma biblioteca de funções que oferece uma infraestrutura para essa tarefa
- Multilinguagem: C, C++, Java, Fortran, Perl, Python
- Padrão livre, com diversas implementações

Distribuições MPI

- Existem diversas implementações deste padrão:
  - MPICH: a mais padrão
  - OpenMPI: a segunda mais usada
  - IBM MPI: para Blue Gene computers
  - Intel MPI
  - Microsoft-MPI: MS-MPI

Características Básicas de Programação MPI

- **Um mesmo programa é iniciado em diversos computadores / núcleos diferentes**
- Em cada instância, uma parte da tarefa é realizada
- MPI gerencia a criação e provém a maneira pela qual todas as instâncias se comunicam

Características Básicas de Programação MPI

- A passagem de mensagens é adequada para lidar com computação onde uma tarefa pode ser dividida em subtarefas
- Casa subtarefa será, então, resolvida por um processo
- Geralmente um processo gerencia as tarefas:
    - Esse processo é chamado de ***“master”*** e os demais processos de ***“workers”***

Em resumo

- MPI é uma biblioteca potente para realização de Computação Distribuída
- Gerencia processos, threads e comunicação de forma transparente e independente de linguagem
- Apresenta bons resultados em problemas dos mais variados

Instalando o MPI no Windows

- Download: https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi?redirectedfrom=MSDN

- Encontrar o link para: **MS-MPI v10.1.2**

- Seguir os passos da instalação

Instalando o MPI no Linux (Ubuntu)

- Descarregando e instalando o MPI
  - `sudo apt-get install libcr-dev`
  - `sudo apt-get install mpich`
  - `sudo apt-get install mpich-doc`
- Ou tudo de uma vez:
  - `sudo apt-get install libcr-dev mpich mpich-doc`

Usando o MPI em Python

```
from mpi4py import MPI
```

- Precisamos antes instalar o mpi4py:

```
pip install mpi4py
```

Meu primeiro programa

```
1  from mpi4py import MPI
2
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5  size = comm.Get_size()
6
7  print('Hello from process {} out of {}'.format(rank, size))
```

- Na programação paralela com MPI, precisamos do chamado **comunicador (comm)**, que é um grupo de processos que podem conversar entre si

Meu primeiro programa

```
1  from mpi4py import MPI
2
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5  size = comm.Get_size()
6
7  print('Hello from process {} out of {}'.format(rank, size))
```

- Cada processo do grupo recebe uma **identificação** ou uma **classificação única (rank)** dentro do comunicador

Meu primeiro programa

```
1  from mpi4py import MPI
2
3  comm = MPI.COMM_WORLD
4  rank = comm.Get_rank()
5  size = comm.Get_size()
6
7  print('Hello from process {} out of {}'.format(rank, size))
```

- O número total de processos é referido como o **tamanho (size)** do comunicador

Execução do primeiro programa

- Uma maneira usual de executar o código em paralelo é por meio do **mpiexec**:

```
mpiexec -n 8 python .\exemplo1.py
```

- O comando acima executará o código em **8 processos**, assumindo que o código é salvo em um arquivo chamado *“exemplo1.py”*
- Às vezes, ***mpiexec*** é chamado de ***mpirun***

Execução do primeiro programa

```
Hello from process 2 out of 8
Hello from process 7 out of 8
Hello from process 5 out of 8
Hello from process 6 out of 8
Hello from process 3 out of 8
Hello from process 0 out of 8
Hello from process 4 out of 8
Hello from process 1 out of 8
```

Exemplo 1 - Repetição

```
1  numeros = 10
2
3  for n in range(numeros):
4      print("Exibindo número: ", n)
```

Exemplo 1 - Repetição - MPI

```python
from math import ceil
from mpi4py import MPI

numeros = 10

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

numeros_por_processo = ceil(numeros / size)

primeiro = rank * numeros_por_processo
ultimo = primeiro + numeros_por_processo

for i in range(primeiro, ultimo):
    if i < numeros:
        print("Processo {0} exibindo numero {1}".format(rank, i))
```

Comunicando dados

- A comunicação de dados é essencial para o funcionamento de um sistema distribuído
- No MPI, ela pode ser feita de algumas maneiras, como por exemplo:
  - Ponto-a-Ponto
    - Com ou sem bloqueio
  - Coletiva

Exemplo 2 - Comunicação Ponto-a-Ponto (com bloqueio)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# processo master é igual a 0
if rank == 0:
    coordenada = {'x': 0.5, 'y': 8.5, 'z': 3.7}
    # master envia 'coordenada' para os todos os processos workers
    for i in range(1, size):
        comm.send(coordenada, dest=i, tag=i)
        print('Processo {} envia coordenada:'.format(rank), coordenada)

# workers
else:
    # cada processo worker recebe 'coordenada' com o nome de
    # 'data' do processo master (0)
    data = comm.recv(source=0, tag=rank)
    print('Processo {} recebe coordenada:'.format(rank), data)
```

Exemplo 2 - Comunicação Ponto-a-Ponto (com bloqueio)

- Os dados do dicionário ***“coordenada”*** são enviados do processo ***master*** para cada processo ***worker***
- O valor de ***rank*** é utilizado para determinar se um processo é o ***master*** ou um ***worker***
- O loop for é executado apenas para o processo ***master*** e i (começando em 1) percorre todos os processos ***workers***
- O parâmetro ***tag*** nos métodos ***send*** e ***recv*** pode ser usado para distinguir entre as mensagens se houver várias comunicações entre dois processos

Exemplo 2 - Comunicação Ponto-a-Ponto (com bloqueio)

- Esse exemplo utilizou métodos de comunicação com bloqueio (***send*** e ***recv***):
  - Isso significa que a execução do código não continuará até que a comunicação seja concluída

Exemplo 3 - Comunicação Ponto-a-Ponto (sem bloqueio)

- O comportamento de bloqueio nem sempre é desejável na programação paralela
- Em alguns casos, é benéfico usar métodos de comunicação sem bloqueio
- Para isso, os métodos ***isend*** e ***irecv*** podem ser utilizados
- O método ***wait*** pode ser usado para gerenciar a conclusão da comunicação

Exemplo 3 - Comunicação Ponto-a-Ponto (sem bloqueio)

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    coordenada = {'x': 0.5, 'y': 8.5, 'z': 3.7}
    for i in range(1, size):
        req = comm.isend(coordenada, dest=i, tag=i)
        req.wait()
        print('Processo {} enviou coordenada:'.format(rank), coordenada)

else:
    req = comm.irecv(source=0, tag=rank)
    data = req.wait()
    print('Processo {} recebeu coordenada:'.format(rank), data)
```

Comunicação coletiva

- Muitas vezes é útil realizar o que é conhecido como comunicação coletiva:
  - Transmitir um objeto Python do processo *master* para todos os processos *workers*
- Algumas funções permitem enviar dados para vários processos ao mesmo tempo:
  - `bcast`, `reduce`
    - Envia e recebe mensagem para todos os processos
  - `scatter`, `gather`
    - Espalham e reúnem dados

Exemplo 4 - `bcast` (broadcast)

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    data = np.arange(4.0)
else:
    data = None

data = comm.bcast(data, root=0)
if rank == 0:
    print('Processo {} enviou data:'.format(rank), data)
print('Processo {} recebeu data:'.format(rank), data)
```

Exemplo 4 - `bcast` (broadcast)

- **`bcast`** transmite um objeto Python do processo *master* para todos os processos *workers*
- O código anterior transmite uma matriz Numpy usando o método **`bcast`**

Exemplo 5 - `scatter`

- Se for necessário dividir uma tarefa e distribuir as subtarefas aos processos, o método **`scatter`** (dispersão) pode ser utilizado
- **`scatter`** vai dividir uma lista ou array e enviar as diferentes partes para diferentes processos
- Contudo, não é possível distribuir mais elementos do que o número de processadores
- Para uma lista ou array, é necessário fazer fatias conforme a quantidade de processos antes de chamar o método **`scatter`**

Exemplo 5 - `scatter`

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
nprocs = comm.Get_size()

if rank == 0:
    data = np.arange(15.0)

    # tamanho de cada subtarefa
    quociente, resto = divmod(data.size, nprocs)
    counts = [quociente + 1 if p < resto else quociente for p in range(nprocs)]

    # inicio e fim dos indices para cada subtarefa
    starts = [sum(counts[:p]) for p in range(nprocs)]
    ends = [sum(counts[:p+1]) for p in range(nprocs)]

```

Exemplo 5 - `scatter`

```
19      # converte data para ser uma lista de listas
20      # cada uma dessas listas internas será enviada para um processo
21      data = [data[starts[p]:ends[p]] for p in range(nprocs)]
22  else:
23      data = None
24
25  data = comm.scatter(data, root=0)
26
27  print('data no processo {}:'.format(rank), data)
```

Exemplo 6 - `gather`

- O método **`gather`** faz o oposto do **`scatter`**
- Se cada processo tiver um elemento, pode-se usar **`gather`** para coletá-los em uma lista de elementos no processo *master*

Exemplo 6 - Cálculo de π por Integração de Monte Carlo

```python
import random
import time

def calculo_pi(n):
    n_inside_circle = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x**2 + y**2 < 1.0:
            n_inside_circle += 1
    return n_inside_circle

n = 10**7

t0 = time.perf_counter()
n_inside_circle = calculo_pi(n)
t = time.perf_counter() - t0

pi = 4.0 * (n_inside_circle / n)
print("pi:", pi, "tempo:", t, "s")
```

Exemplo 6 - Cálculo de π por Integração de Monte Carlo com Programação Paralela e `gather`

```python
import random
import time
from mpi4py import MPI

def calculo_pi(n):
    n_inside_circle = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x ** 2 + y ** 2 < 1.0:
            n_inside_circle += 1
    return n_inside_circle

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

n = 10**7

if size > 1:
    n_task = int(n / size)
else:
    n_task = n
```

Exemplo 6 - Cálculo de π por Integração de Monte Carlo com Programação Paralela e `gather`

```
24
25  t0 = time.perf_counter()
26  n_inside_circle = calculo_pi(n_task)
27  t = time.perf_counter() - t0
28
29  print(f"Antes do gather: rank {rank}, n_inside_circle: {n_inside_circle}")
30  n_inside_circle = comm.gather(n_inside_circle, root=0)
31  print(f"Depois do gather: rank {rank}, n_inside_circle: {n_inside_circle}")
32
33  if rank == 0:
34      pi = 4.0 * sum(n_inside_circle) / n
35      print("pi:", pi, "tempo:", t, "s", "pontos:", sum(n_inside_circle))
```

Exemplo 7 - `reduce`

- Pode-se usar também o método **`reduce`** para coletar resultados
- **`reduce`** trata a redução de um conjunto de dados em um conjunto menor
- O processo de redução pode ser feito, dentre outras formas, com ***soma*** ou ***multiplicação***
- Exemplo: considere a *lista* ***[1, 2, 3, 4, 5]***
  - Redução por soma: ***sum([1, 2, 3, 4, 5]) = 15***
  - Redução por multiplicação: ***multiply([1, 2, 3, 4, 5]) = 120***

Exemplo 7 - `reduce`

- No MPI, **`reduce`** inclui as seguintes possibilidades:
- **`MPI.MAX`** - Retorna o elemento máximo
- **`MPI.MIN`** - Retorna o elemento mínimo
- **`MPI.SUM`** - Soma os elementos
- **`MPI.PROD`** - Multiplica todos os elementos
- **`MPI.LAND`** - Executa lógica *and* entre os elementos
- **`MPI.LOR`** - Executa lógica *or* entre os elementos
- **`MPI.MAXLOC`** - Retorna o valor máximo e o rank do processo que o possui
- **`MPI.MINLOC`** - Retorna o valor mínimo e o rank do processo que o possui

Exemplo 7 - `reduce` - Cálculo de π

- Até a linha 27, é o mesmo código desenvolvido para o `gather`

```
25  t0 = time.perf_counter()
26  n_inside_circle = calculo_pi(n_task)
27  t = time.perf_counter() - t0
28
29  n_inside_circle_total = comm.reduce(n_inside_circle, op=MPI.SUM, root=0)
30
31  if rank == 0:
32      pi = 4.0 * n_inside_circle_total/ n
33      print("pi:", pi, "tempo:", t, "s", "pontos:", n_inside_circle_total)
```

Finalmente

- Esta foi apenas uma breve introdução ao MPI
- Vejam tutoriais, exemplos, que tem muito mais:
  - Sincronização de tarefas, cálculos vetoriais e matriciais etc.

Conclusão

- MPI é uma biblioteca potente para realização de Computação Distribuída
- Muito usado em HPC
- Gerencia processos e comunicação de forma transparente e independente de linguagem
- Apresenta bons resultados em problemas dos mais variados

Exercício

- Implemente o Cálculo das Integrais abaixo pelo Método de Monte Carlo usando Programação Paralela e MPI

a) $$\int_0^1 \frac{4}{1+x^2} dx$$

b) $$\int_0^1 \sqrt{x+\sqrt{x}} dx$$