Programação em sistemas de memória distribuída

(Message-Passing Computing)

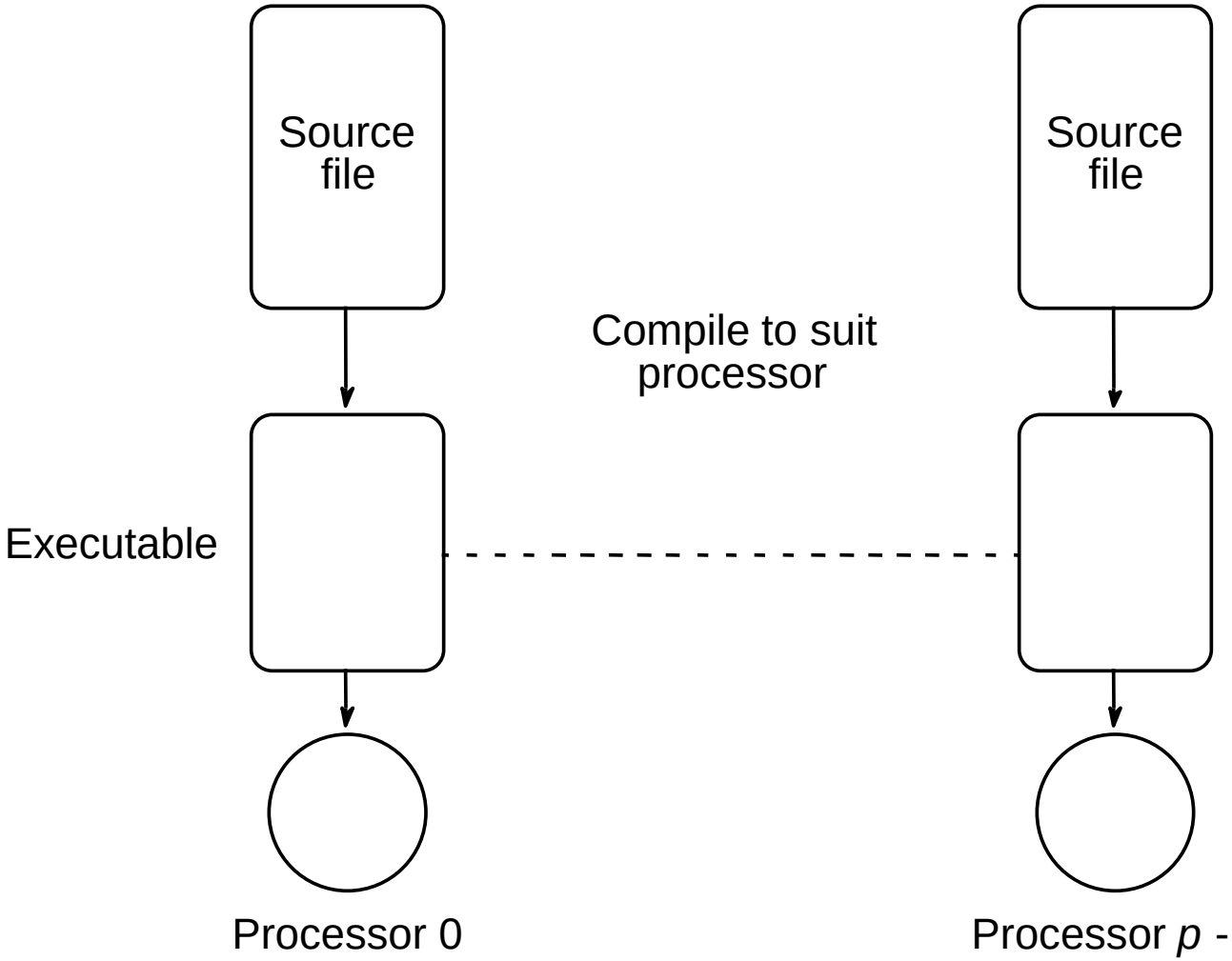
Programação distribuída requer bibliotecas com rotinas para comunicação por mensagens.

São necessários dois mecanismos base:

1. Um método para criar processos para execução em computadores/processadores diferentes

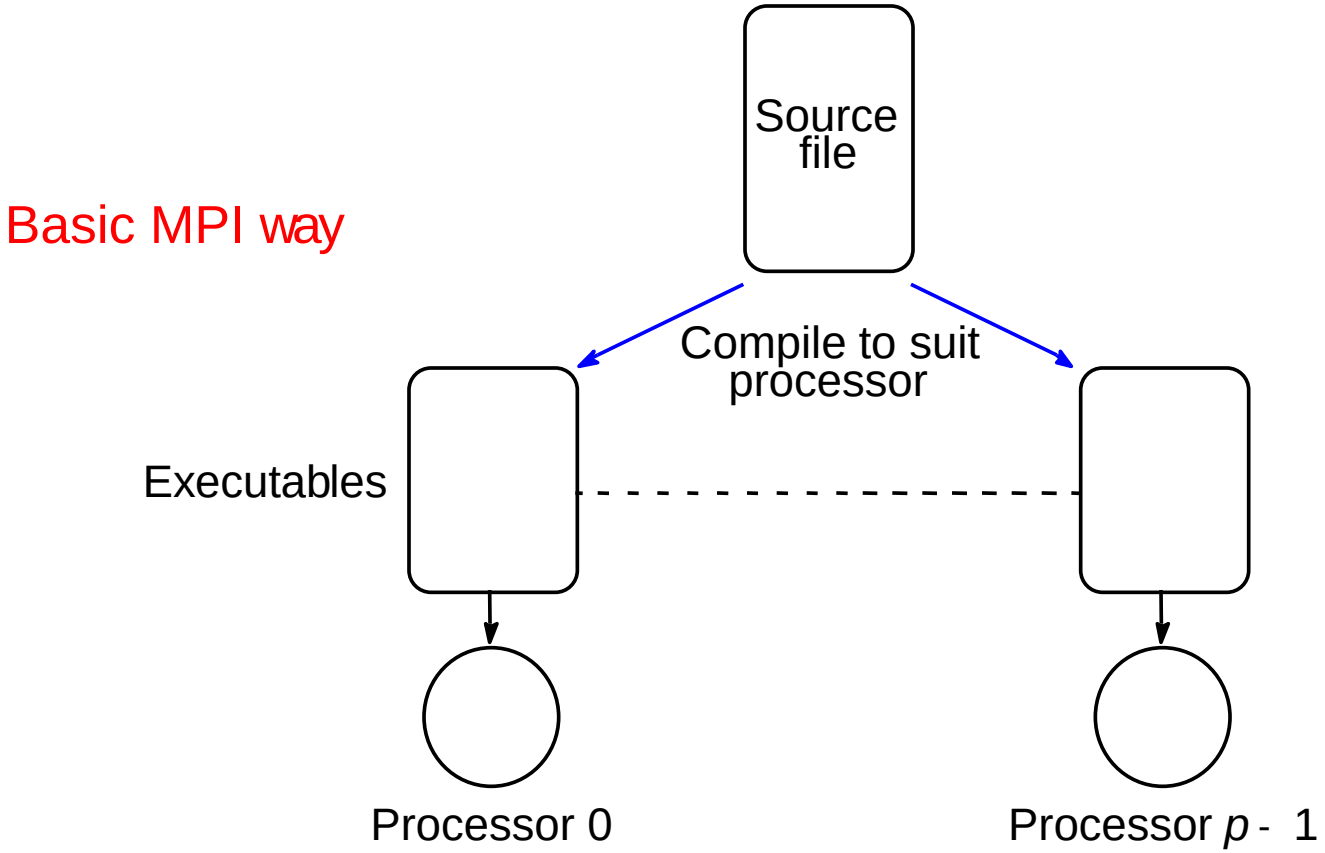
2. Um método para enviar e receber mensagens.

Multiple program, multiple data (MPMD) model



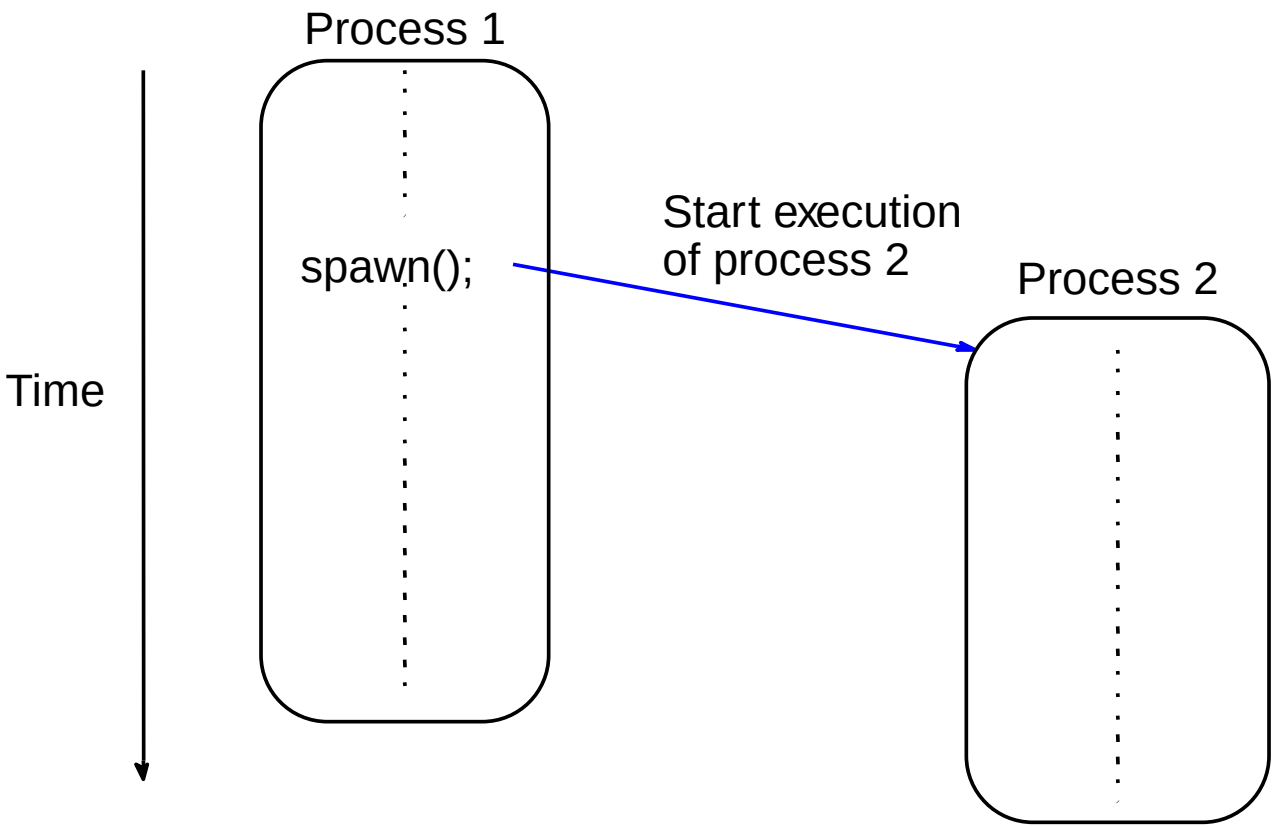
Single Program Multiple Data (SPMD) model

Diferentes processos construídos num único programa. Instruções de controlo selecionam diferentes partes do código a serem executadas por cada processo. Todos os executáveis começam ao mesmo tempo (criação estática).



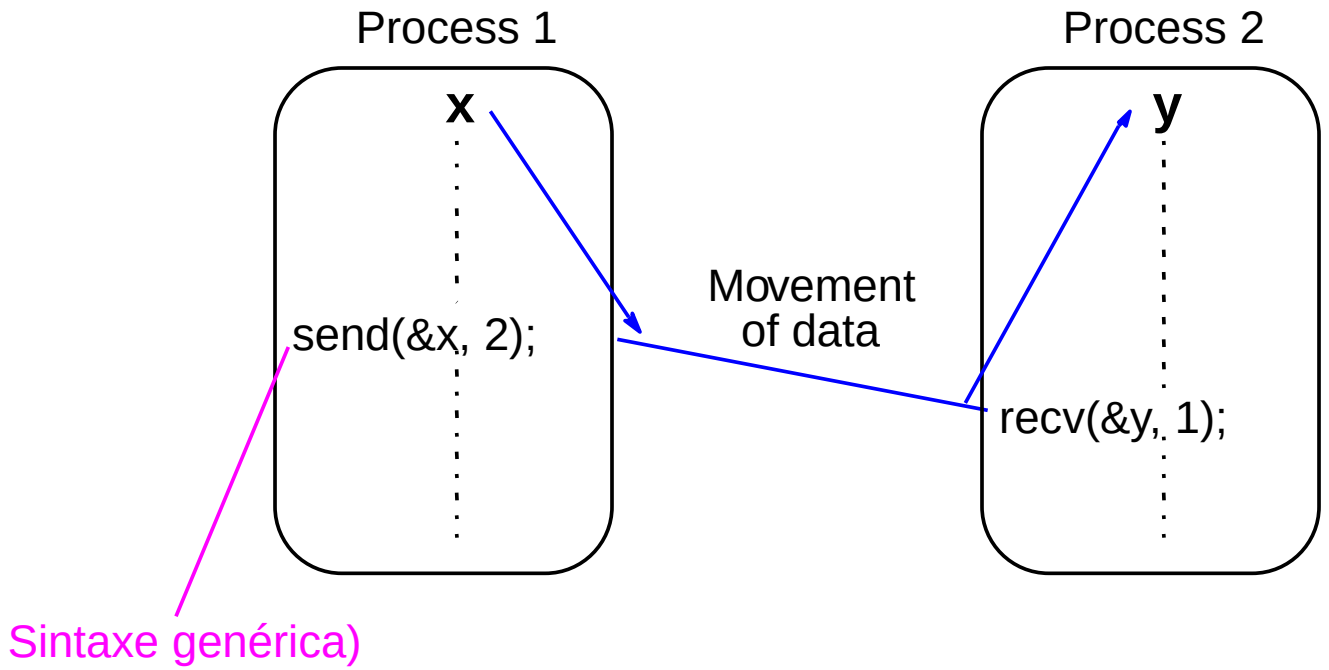
Multiple Program Multiple Data (MPMD) Model

Programas separados para cada processador. Um processador executa o processo master. Os outros processos são criados dinamicamente pelo master.



Rotinas para comunicação ponto a ponto: Send and Receive

Envio de uma mensagem entre processos:



Comunicação por mensagens **síncrona**

Rotinas que só retornam quando a transferência da mensagem está completa.

Rotina de envio (send) síncrona

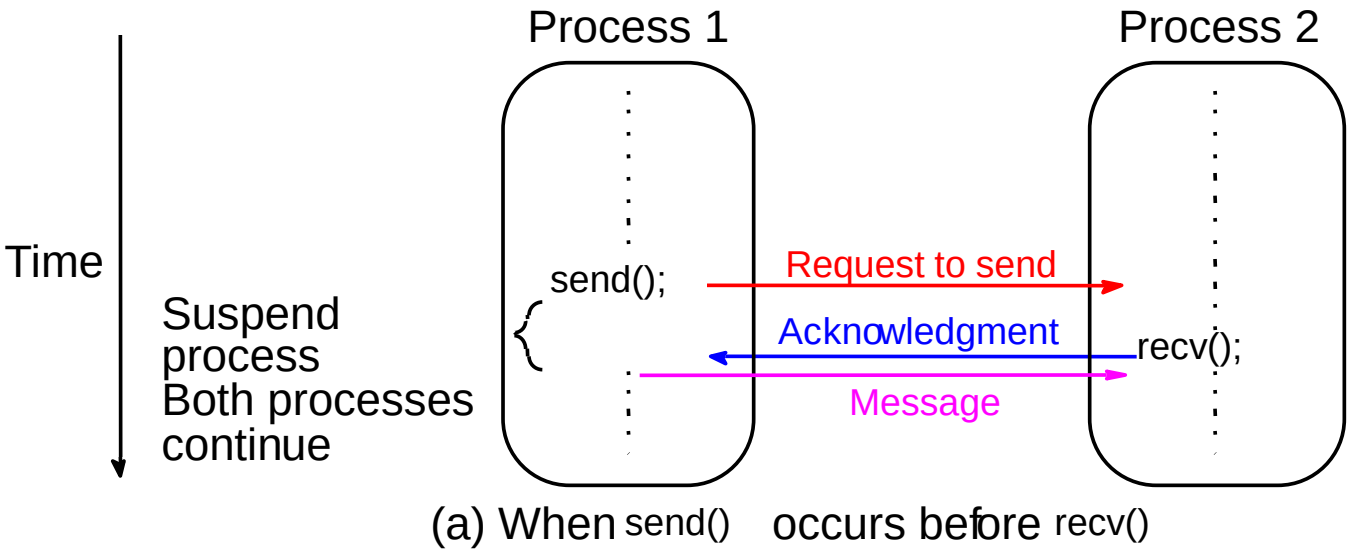
- Antes de enviar a mensagem, espera até que a mesma possa ser aceite pelo processo receptor.

Rotina de receção (receive) síncrona

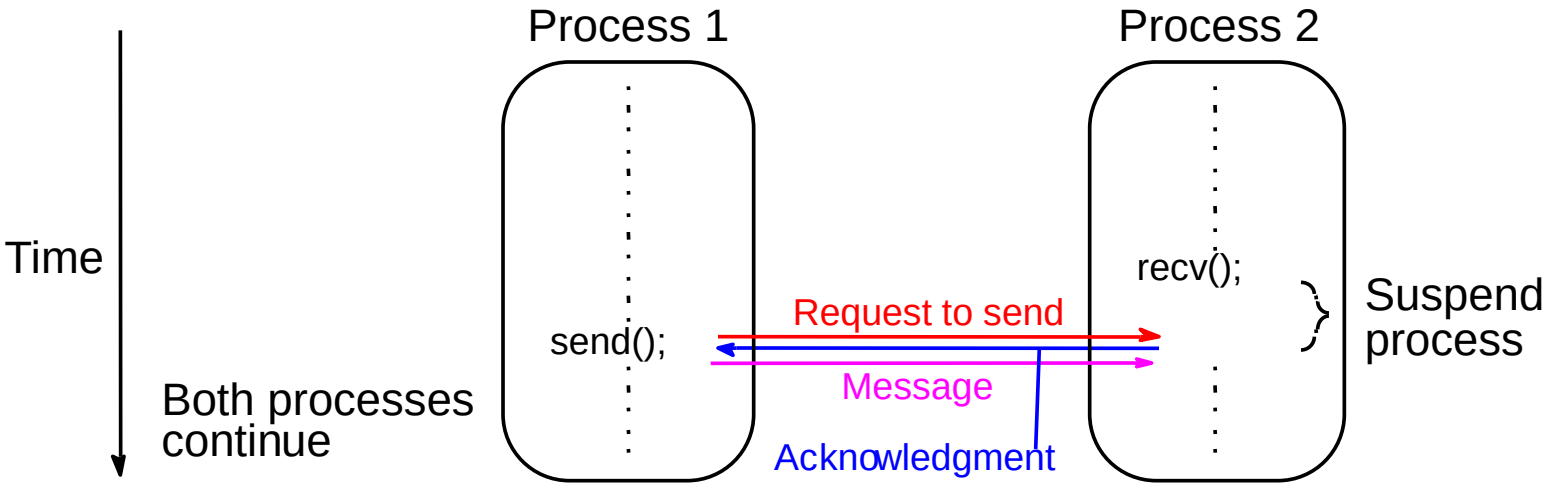
- Espera até que a mensagem esperada chegue.

Rotinas síncronas executam duas ações: transferem dados e sincronizam processos.

Synchronous send() and recv() using 3-way protocol



(a) When send() occurs before recv()



(b) When recv() occurs before send()

Comunicação por mensagens **assíncrona**

- Rotinas não esperam que a comunicação termine antes de retornarem. É necessário um sistema de armazenamento de mensagens.

- Mais do que uma versão, dependendo da semântica de retorno.

- Geralmente não sincronizam processos mas permitem aumentar o paralelismo. Devem ser usadas com cuidado.

Blocking versus Non-Blocking em MPI

- Blocking – retornam após as operações locais serem completadas, a transferência da mensagem pode não estar completa.

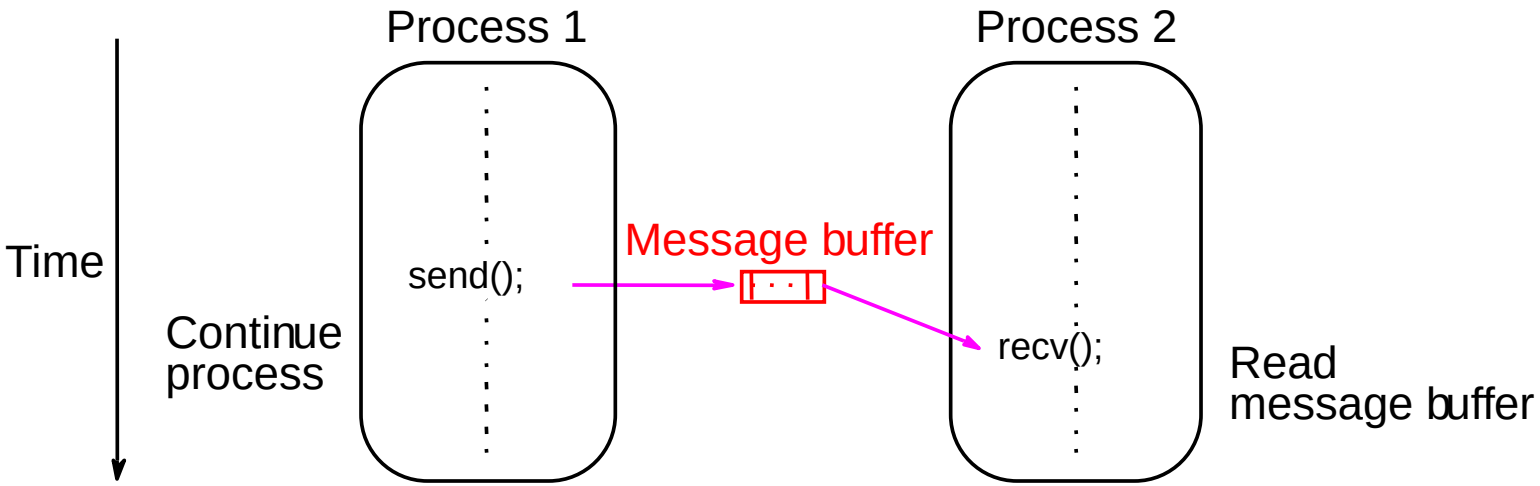
- Non-blocking – retornam imediatamente.

Assume que os dados a transferir não são alterados por outras instruções antes de serem transferidos. Assegurar isso é da responsabilidade do programador.

Os termos blocking / non-blocking podem ter diferentes interpretações, dependendo da implementação.

Rotinas que retornam antes da mensagem ser transferida:

Necessário um Buffer de mensagens entre a origem e o destino:



Rotinas assíncronas (blocking) que passam a rotinas síncronas

- Quando as operações locais estão completas e a mensagem está armazenada em segurança, o emissor prossegue o seu trabalho.

- O buffer tem tamanho finito, pode acontecer que a rotina de envio bloqueie porque o buffer está cheio.

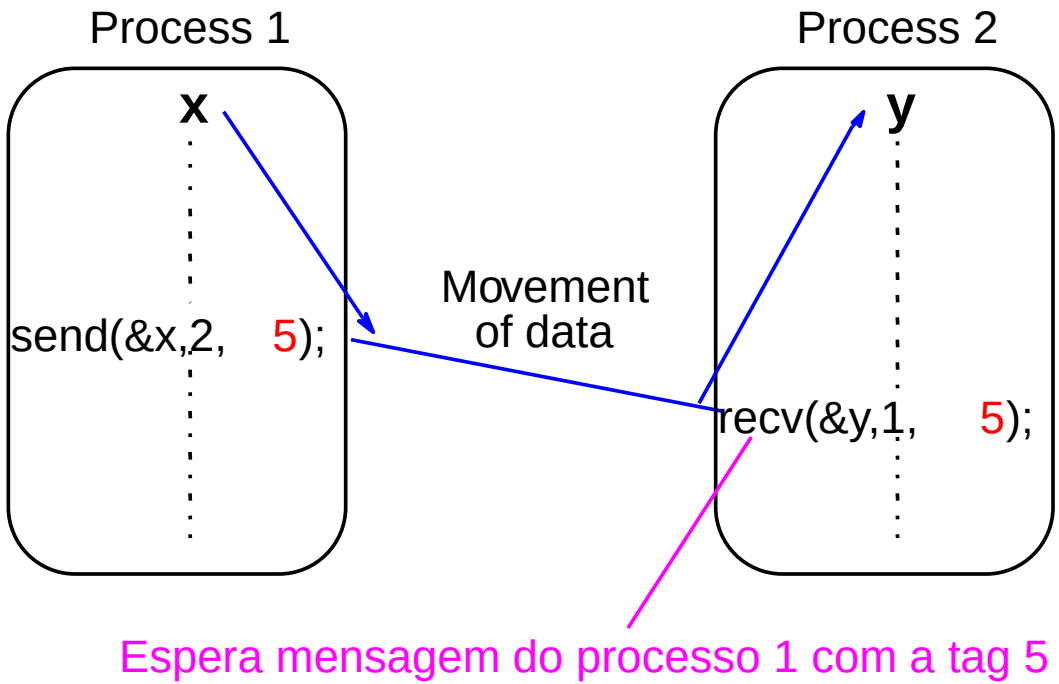
- Nesse caso, a rotina de send é suspensa até que o buffer esteja disponível, i.e., a rotina comporta-se como se fosse síncrona.

Etiquetas (tags) em mensagens

- Usadas para diferenciar mensagens de diferentes tipos.
- A etiqueta é enviada com a mensagem.
- Se a mensagem pode ser qualquer, poderá ser usado um “wild card” na mensagem. Assim um receive vai corresponder a qualquer send.

Exemplo de uma “Message Tag”

Envio da mensagem, x, com a *tag* 5 pelo processo 1 para o processo 2 sendo atribuída a y:



Rotinas de “Grupo”

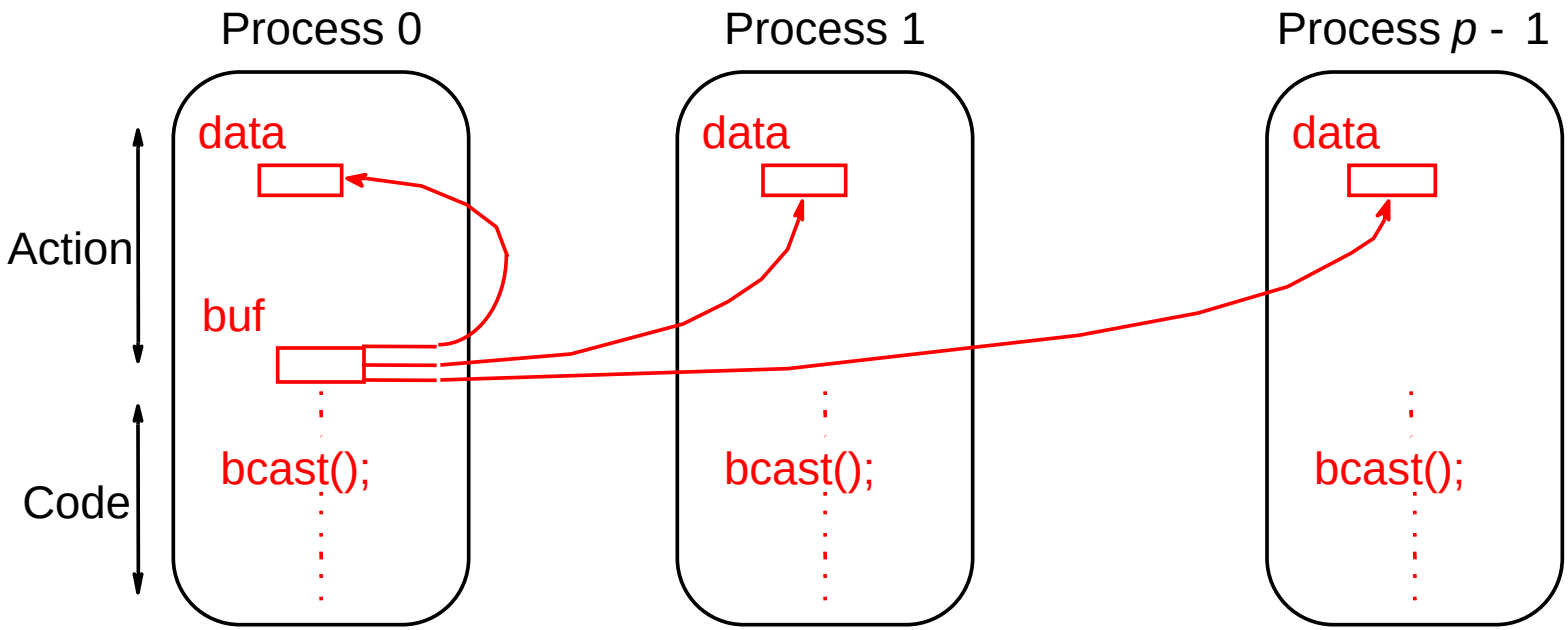
Existem rotinas que enviam mensagens para um grupo de processos ou que recebem mensagens de um grupo de processos.

Mais eficiente do que usar rotinas de comunicação ponto a ponto separadamente.

Broadcast

Envio de mensagens a todos os processos relacionados com o problema a resolver.

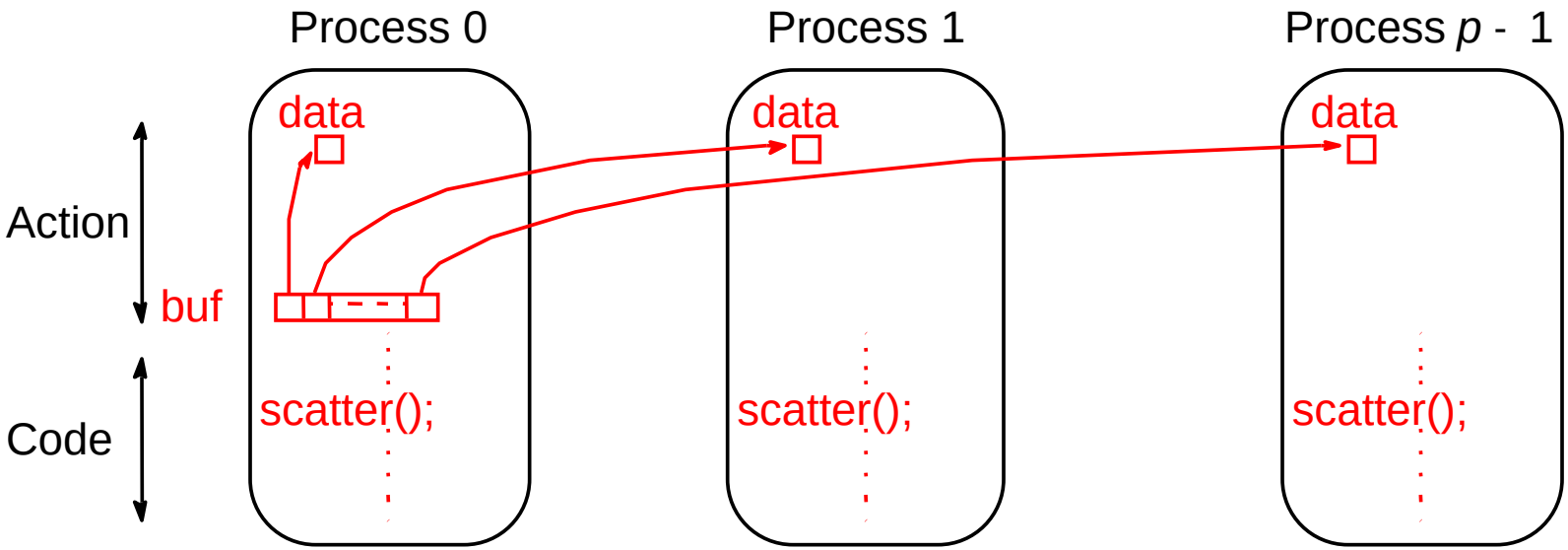
Multicast – enviar a mesma mensagem a um grupo pré-definido de processos.



MPI form

Scatter

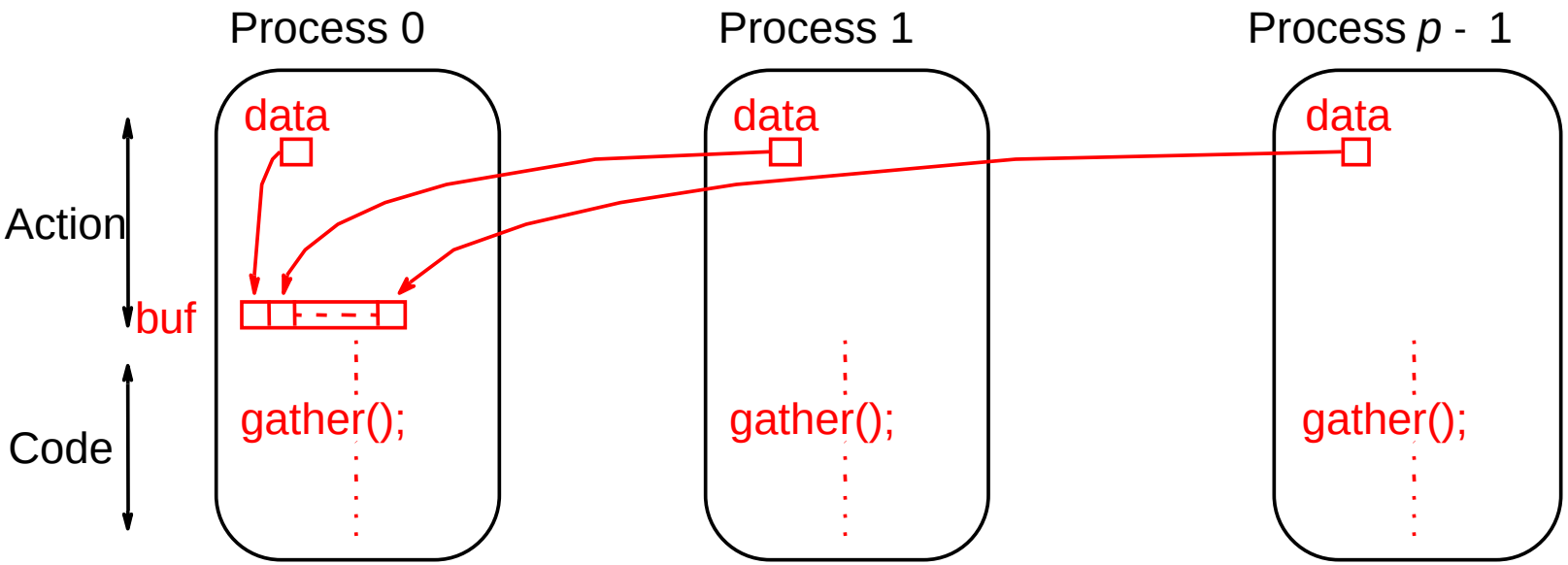
Enviar cada elemento de um array no processo root para outro processo. Conteúdo da posição *i* do array é enviado para o processo *i*.



MPI form

Gather

Um processo recolhe valores individuais de um conjunto de processos.

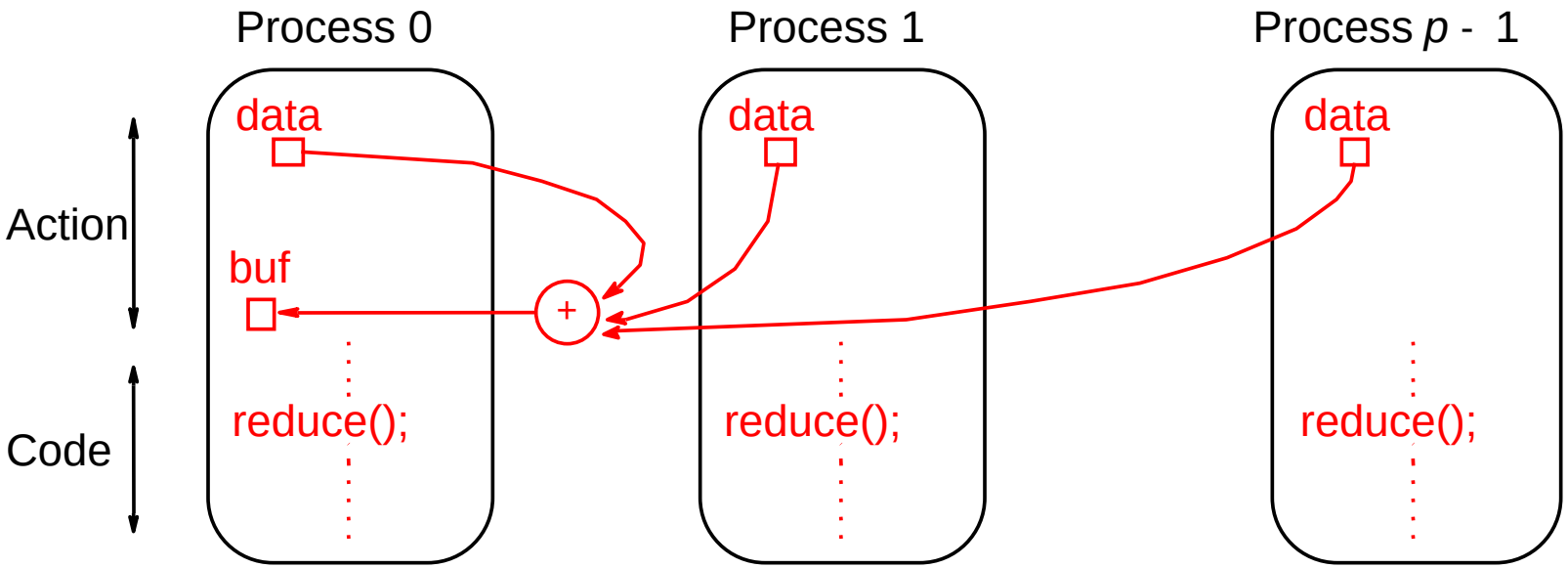


MPI form

Reduce

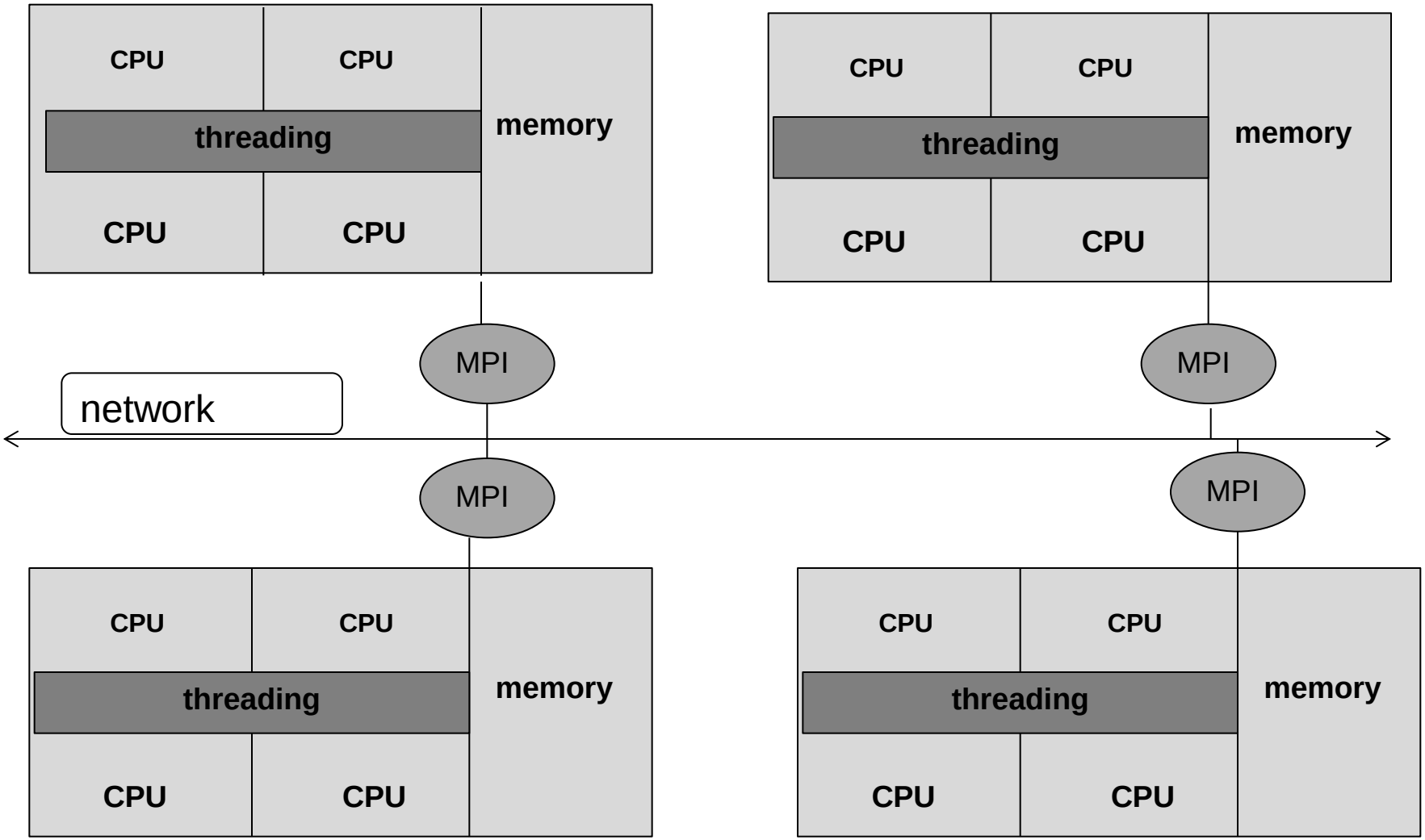
Operação de *gather* combinada com uma operação aritmética ou lógica.

Exemplo: valores são recolhido e adicionados no processo root:

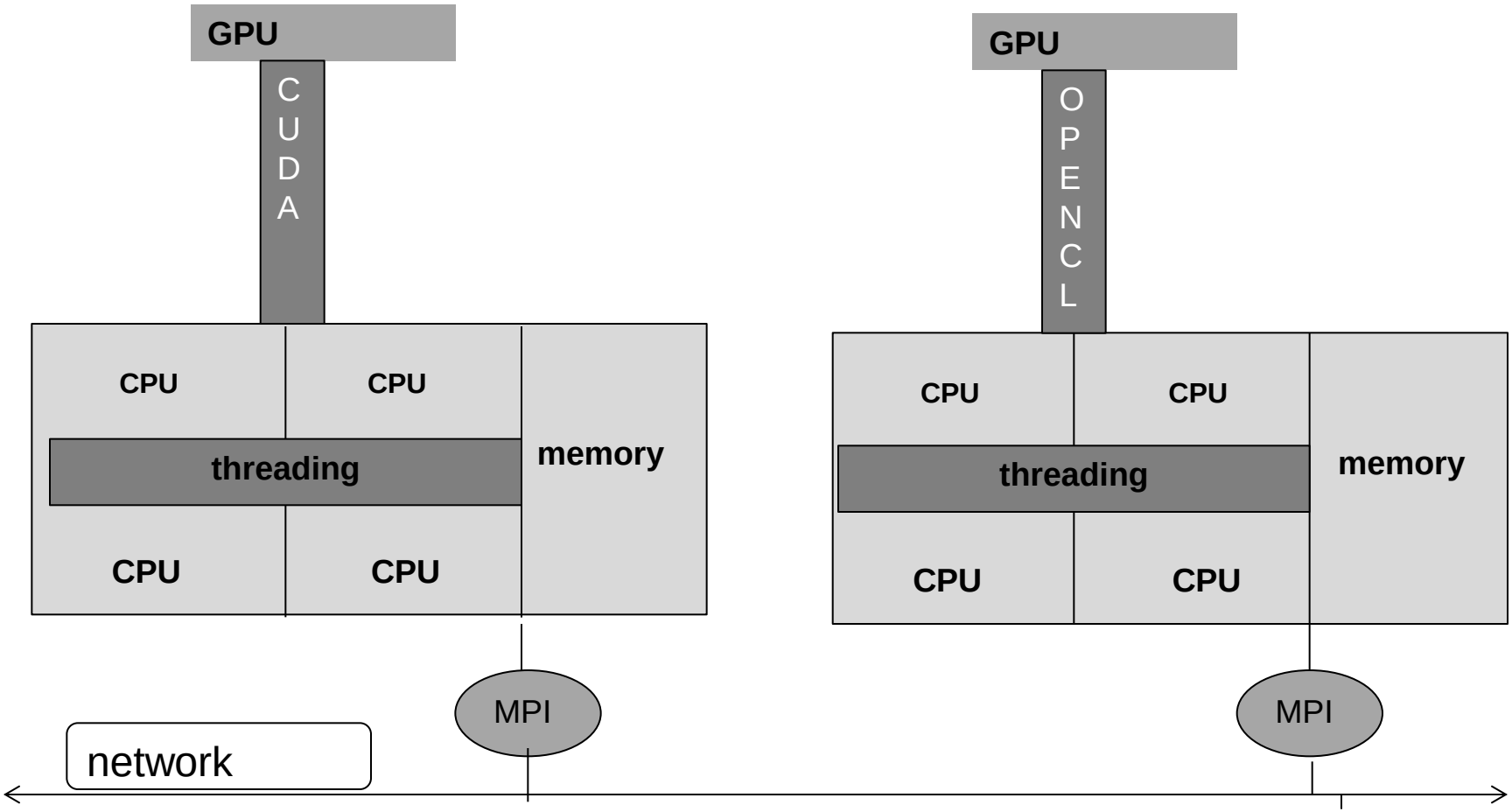


MPI form

Modelos híbridos de programação



Modelos híbridos de programação



Programação distribuída em python módulo **multiprocessing**

```
#Criar um processo:
import multiprocessing

def function(i):
    print ('called function in process: %s' %i)
    return

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=function, args=(i,))
        Process_jobs.append(p)
        p.start()    # iniciar execução
        p.join()    # esperar que termine     !!!!
```

Programação distribuída em python módulo **multiprocessing**

```
#output

called function in process: 0
called function in process: 1
called function in process: 2
called function in process: 3
called function in process: 4
```

Programação distribuída em python módulo **multiprocessing**

#Classe Process:

class multiprocessing.Process(*group=None*, *target=None*, *name =None*, *args=()*, *kwargs={}*, *, *daemon=None*)

Permite criar um fluxo de atividade que executará num processo separado.

group - só existe por compatibilidade com threading.Thread

target - ”callable object” invocado pelo método run.

name - nome do processo.

Programação distribuída em python módulo **multiprocessing**

#Classe Process:

args - tuplo com os argumentos para a invocação do “target”. invocation.

daemon - True ou False, por omissão, o processo herda o valor do processo pai.

Programação distribuída em python módulo **multiprocessing**

```
#Dar nome a um processo
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Starting %s \n" %name)
    time.sleep(3)
    print ("Exiting %s \n" %name)
```

Programação distribuída em python módulo **multiprocessing**

```
#Dar nome a um processo

if __name__ == '__main__':
    process_with_name = multiprocessing.Process\
            (name='foo_process', target=foo)

    process_with_default_name = multiprocessing.Process \
                                            (target=foo)

    process_with_name.start()
    process_with_default_name.start()
```

Programação distribuída em python módulo **multiprocessing**

#Output

Starting Process-2

Exiting Process-2

Starting foo_process

Exiting foo_process

Programação distribuída em python módulo **multiprocessing**

```
#Terminar um processo

import multiprocessing
import time

def foo():
    print ('Starting function')
    time.sleep(0.1)
    print ('Finished function')
```

Programação distribuída em python módulo **multiprocessing**

```
if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print ('Process before execution:', p, p.is_alive())

    p.start()
    print ('Process running:', p, p.is_alive())

    p.terminate() # cuidado se recursos part. Locks, Sem.
    print ('Process terminated:', p, p.is_alive())

    p.join()
    print ('Process joined:', p, p.is_alive())
    print ('Process exit code:', p.exitcode)
```

Programação distribuída em python módulo **multiprocessing**

```
#output

Process before execution: <Process(Process-1, initial)> False
Process running: <Process(Process-1, started)> True
Process terminated: <Process(Process-1, started)> True
Process joined: <Process(Process-1, stopped[SIGTERM])> False
Process exit code: -15
```

Programação distribuída em python módulo **multiprocessing**

```
if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print ('Process before execution:', p, p.is_alive())

    p.start()
    print ('Process running:', p, p.is_alive())

    time.sleep(5)
    p.terminate()
    print ('Process terminated:', p, p.is_alive())

    p.join()
    print ('Process joined:', p, p.is_alive())
    print ('Process exit code:', p.exitcode)
```

Programação distribuída em python módulo **multiprocessing**

#output

Process before execution: <Process(Process-1, initial)> False
Process running: <Process(Process-1, started)> True
Starting function
Finished function
Process terminated: <Process(Process-1, stopped)> False
Process joined: <Process(Process-1, stopped)> False
Process exit code: 0

O que aconteceu de diferente?

Programação distribuída em python módulo **multiprocessing**

```
# Criar um processo como subclasse de Process
import multiprocessing

class MyProcess(multiprocessing.Process):

    #sobrepor métod run

    def run(self):
        print ('called run method in %s' %self.name)
        return
```

Programação distribuída em python módulo **multiprocessing**

```
# Criar um processo como subclasse de Process



if __name__ == '__main__':
    jobs = []

    for i in range(5):
        p = MyProcess()
        jobs.append(p)
        p.start()
        p.join()
```

Programação distribuída em python módulo **multiprocessing**

```
# output:

called run method in MyProcess-1
called run method in MyProcess-2
called run method in MyProcess-3
called run method in MyProcess-4
called run method in MyProcess-5
```

Exemplo: Somar os inteiros de uma lista, sequencialmente, com threads, com processos:

```
import threading
import multiprocessing
import random
import time

SIZE = 10000000

def somaParcial ( lista, p, u ):
    soma = 0
    for i in range(p, u):
        soma += lista[i]
    print (soma)
    return soma
```

Exemplo: Somar os inteiros de uma lista, sequencialmente, com threads, com processos:

```
if __name__ =="__main__":
    start= time.time()
    lista = [random.randint(1,10) for i in range(SIZE)]
    print ("random values %s" % (time.time() - start) )

    print ("starting")
    start= time.time()
    somaParcial(lista, 0, SIZE)
    print ("Sequential time = %s" %(time.time() - start))
```

Exemplo: Somar os inteiros de uma lista, sequencialmente, com threads, com processos:

```
start= time.time()
t1 = threading.Thread (target = somaParcial, args =(lista, 0,  \
                                                    int(SIZE/2)) )
t2 = threading.Thread (target = somaParcial, args =(lista, \
                                                  int(SIZE/2), SIZE))
t1.start()
t2.start()
t1.join()
t2.join()
print ("Multithreaded time = %s" % (time.time() - start))
```

Exemplo: Somar os inteiros de uma lista, sequencialmente, com threads, com processos:

```
print ("Multithreaded time = %s" % (time.time() - start))
start= time.time()
t1 = multiprocessing.Process (target = somaParcial, args = \
                                        (lista, 0, int(SIZE/2)) )
t2 = multiprocessing.Process (target = somaParcial, args = \
                                        (lista, int(SIZE/2), SIZE))
t1.start()
t2.start()
t1.join()
t2.join()
print ("Multiprocessing time = %s" % (time.time() - start))
```

Exercícios:

- Testar o programa anterior, analisando os tempos de execução.

- Como obter o valor parcial calculado por cada thread e cada processo?

- Quais os tempos de execução, se na versão multiprocessador, só passarmos para cada Processo a parte da lista que vai ser somada?

Alterar código para usar arrays (numpy em vez de listas)