Arquiteturas Paralelas e Distribuídas

Ferramentas para programação paralela: bibliotecas MPI

Eduardo Furlan Miranda

Baseado em: IGNÁCIO, A. A. V.; FERREIRA FILHO, V. J. M. MPI: uma ferramenta para implementação paralela. 2002. DOI 10.1590/S0101-74382002000100007.

Message Passing Interface (MPI)

- Padrão de interface para a troca de mensagens em máquinas paralelas com memória distribuída
- Desenvolvido por várias instituições, principalmente dos EUA e Europa, universidades, laboratórios, e governos
- Uma aplicação é constituída por um ou mais processos que se comunicam, acionando-se funções para o envio e recebimento de mensagens entre os processos
- O número de processos no MPI é normalmente fixo
- A comunicação pode ser ponto a ponto, ou um grupo de processos pode invocar operações coletivas de comunicação para executar operações globais

- Suporta comunicação assíncrona e programação modular, através de mecanismos de comunicadores (communicator)
  - permitem definir módulos que encapsulem estruturas de comunicação interna

- Os algoritmos que criam um processo para cada processador podem ser implementados, diretamente, utilizando-se comunicação ponto a ponto ou coletivas

- Os algoritmos que implementam a criação de tarefas dinâmicas ou que garantem a execução concorrente de muitas tarefas, num único processador, precisam de um refinamento nas implementações com o MPI

MPI - Básico

A sintaxe dos comandos varia conforme a linguagem de programação

import mpi4py	Inicia uma execução MPI
MPI.Finalize()	Finaliza a execução
comm.Get_size()	Determina o número de processos
comm.Get_rank()	Determina a identificação de processos
comm.send(data, dest=1, tag=11)	Envia a mensagens
comm.recv(source=0, tag=11)	Receber mensagens

- Os procedimentos geralmente possuem um manipulador de comunicação como argumento
  - identifica o grupo de processos e o contexto das operações
  - proporciona o mecanismo para identificar um subconjunto de processos, durante o desenvolvimento de programas modulares, assegurando que as mensagens, planejadas para diferentes propósitos, não sejam confundidas

Exemplo - send, recv

```
from mpi4py import MPI

# Inicialização do MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Processo 0 envia uma mensagem para o processo 1
if rank == 0:
    data = {'a': 1, 'b': 2}
    comm.send(data, dest=1, tag=11)
    print("Processo 0 enviou dados para o processo 1")
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print("Processo 1 recebeu dados do processo 0:", data)

# **Barreira:** Sincronização de todos os processos
comm.Barrier()
print(f"Processo {rank} passou pela barreira")

# **Finalização do MPI**
MPI.Finalize()
```

Exemplo

```c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int world_size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    printf("Olá do processo %d de um total de %d!\n", world_rank, world_size);
    MPI_Finalize();
    return 0;
}
```

- mpicc -o meu_programa mpi_programa.c
- mpirun -np 4 ./meu_programa

MPI

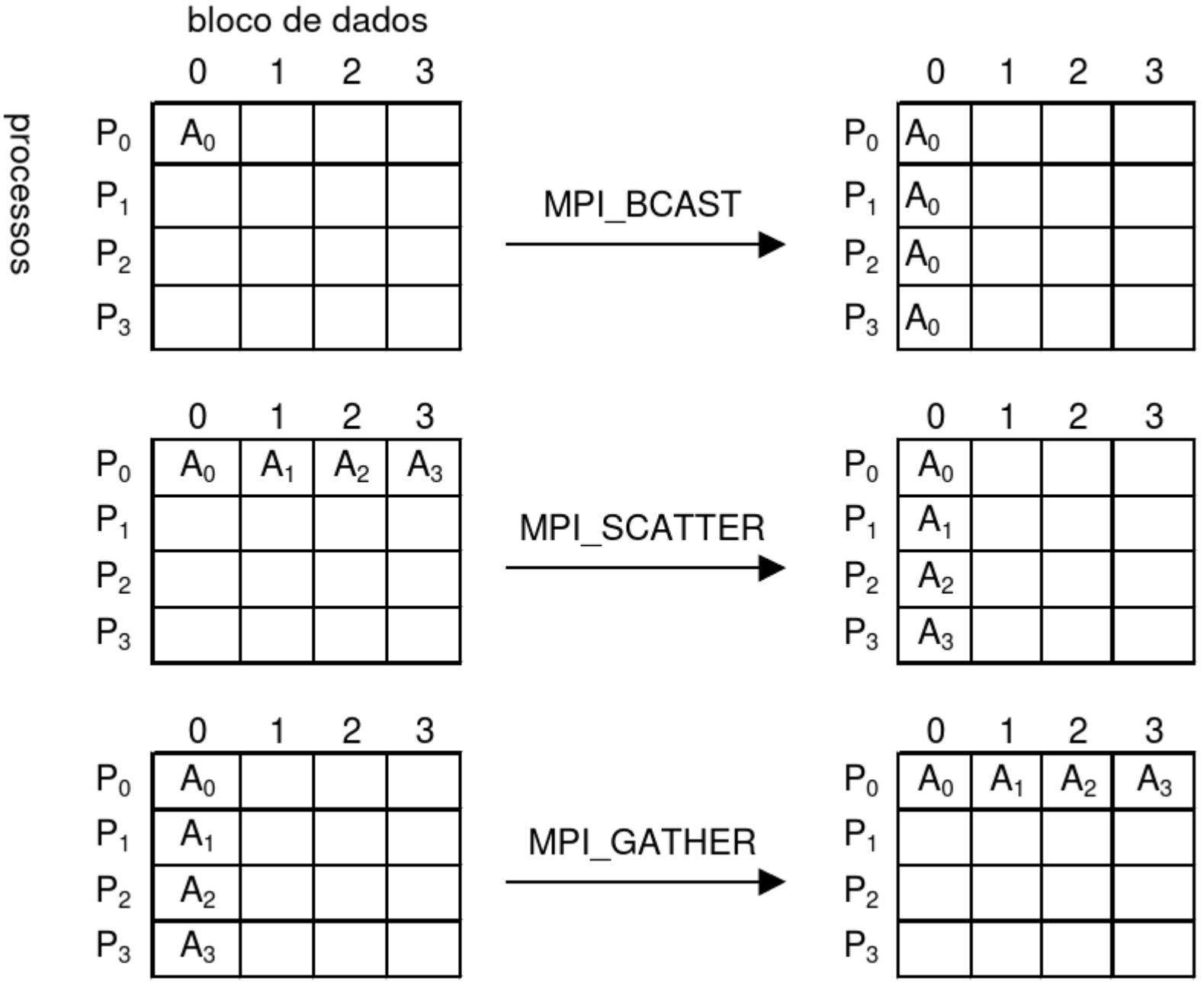
- A programação de troca de mensagens não é determinística ou seja, a chegada das mensagens enviadas por dois processos A e B ao processo C não é definida
- Não garante que uma mensagem, enviada de um processo A e de um processo B, chegue na ordem que foi enviada
- O programador deve assegurar a ordem, se for importante
- RECV pode permitir o recebimento de mensagens vindas de um único processo específico, ou de qualquer processo
- “tag” : mecanismo adicional para distinguir as mensagens

Operações Globais

- Barreira (Barrier): sincroniza todos os processos de um grupo
  - Nenhum processo pode realizar qualquer instrução, até que todos tenham passado por essa barreira
    - Maneira simples de separar as duas etapas da execução para assegurar que as mensagens geradas não se misturem
      - Em alguns casos a necessidade de barreiras explícitas pode ser evitada pelo uso de tag, de origem e/ou contextos específicos
- Difusão (Broadcast): envia dado de um processo a todos os demais processos
  - Dispersão de dados do tipo um para todos, no qual um único processo origem envia um dado para todos os outros processos e cada processo recebe esse dado

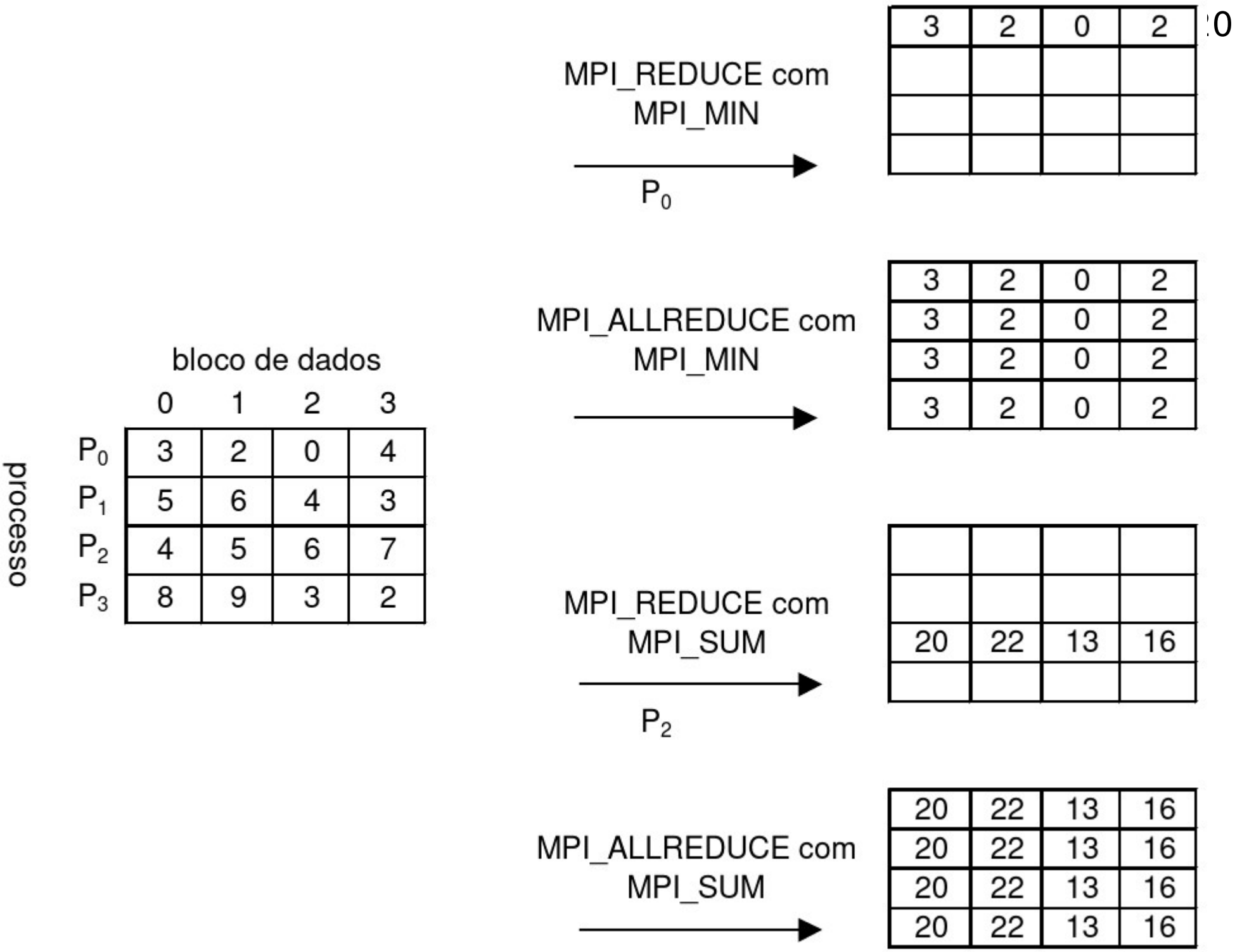
- Juntar (Gather): junta dados de todos os processos em um único processador
  - Todos os processos, incluindo a origem, enviam dados localizados na origem
  - Esse processo coloca os dados em locais contíguos e não opostos, com dados do processo “i”, precedendo os dados do processo “i+1”

- Espalhar (Scatters): distribui um conjunto de dados de um processo para todos os processos (reverso da GATHER)
  - enquanto na BCAST, todo processo recebe o mesmo valor do processo de origem, no SCATTER, todo processo recebe um valor diferente


bloco de dados
0 1 2 3
processos
P0 A0
P1
P2
P3
MPI_BCAST
P0 A0
P1 A0
P2 A0
P3 A0
0 1 2 3
P0 A0 A1 A2 A3
P1
P2
P3
MPI_SCATTER
P0 A0
P1 A1
P2 A2
P3 A3
0 1 2 3
P0 A0
P1 A1
P2 A2
P3 A3
MPI_GATHER
P0 A0 A1 A2 A3
P1
P2
P3

- Operação de redução (Reduction operations): soma, multiplicação, etc., de dados distribuídos

- comm.Reduce() e comm.Allreduce() implementam operações de redução
  - Eles combinam valores fornecidos no bufer de entrada de cada processo, usando operações especificas, e retorna o valor combinado, ou para o bufer de saída de um único processo da origem (REDUCE) ou para os bufer de saída de todos os processo (ALLREDUCE)
  - Essas operações incluem o máximo, o mínimo (MPI.MAX, MPI.MIN), soma, produto (MPI.SUM, MPI.PROD) e as operações lógicas


bloco de dados
0 1 2 3
processo
P0 3 2 0 4
P1 5 6 4 3
P2 4 5 6 7
P3 8 9 3 2
MPI_REDUCE com MPI_MIN
P0
3 2 0 2
MPI_ALLREDUCE com MPI_MIN
3 2 0 2
3 2 0 2
3 2 0 2
MPI_REDUCE com MPI_SUM
P2
20 22 13 16
MPI_ALLREDUCE com MPI_SUM
20 22 13 16
20 22 13 16
20 22 13 16
20 22 13 16

Linguagens Formais e Autômatos

Ferramentas para programação paralela: OpenMP

Eduardo Furlan Miranda

Baseado em: OPENMP. The OpenMP API specification for parallel programming. 2025. https://www.openmp.org/

OpenMP (Open Multi-Processing)

- Geralmente usado dentro de um nó (multi-core) enquanto o MPI é usado entre nós
- Implementação de multithreading, um método de paralelismo pelo qual um thread primário bifurca um número especificado de sub-threads e o sistema divide tarefas entre eles
  - As threads então são executados simultaneamente, com o ambiente de tempo de execução alocando threads para diferentes processadores (em um nó)
- Após a execução do código paralelizado, os threads se juntam novamente ao thread primário, que continua até o final do programa

- Construções de compartilhamento de trabalho podem ser usadas para dividir uma tarefa entre as threads para que cada thread execute sua parte alocada do código

- Tanto o paralelismo de tarefas quanto o paralelismo de dados podem ser alcançados usando OpenMP dessa forma

- O ambiente de tempo de execução aloca threads para processadores dependendo do uso, carga da máquina e outros fatores
  - Pode atribuir o número de threads com base em variáveis de ambiente, ou o código pode fazer isso usando funções
  - As funções OpenMP são incluídas em um arquivo de cabeçalho rotulado omp.h em C/C++

Criação de thread

- O #pragma omp parallel é usado para bifurcar threads adicionais para executar o trabalho incluído na construção em paralelo

- A thread original será denotada como thread mestre com ID 0

```c
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

- Compila com: $ gcc -fopenmp hello.c -o hello -ldl
  - O pacote gcc já inclui a biblioteca OpenMP

- Saída:
  Hello, world.
  Hello, world.

- Pode ocorrer race condition:
  Hello, wHello, woorld.
  rld.

a função printf pode ou não ser atômica, dependendo da implementação

Construções de compartilhamento de trabalho

- Usado para especificar como atribuir trabalho independente a um ou todos os threads
  - omp for ou omp do : usado para dividir iterações de loop entre os threads, também chamados de construções de loop.
  - section : atribuição de blocos de código consecutivos, mas independentes, a diferentes threads
  - single : especificando um bloco de código que é executado por apenas um thread, uma barreira é implícita no final
  - master : semelhante ao single, mas o bloco de código será executado apenas pelo thread master e não haverá nenhuma barreira implícita no final

Exemplo

- Inicializar o valor de uma grande matriz em paralelo, usando cada thread para fazer parte do trabalho

```
int main(int argc, char **argv)
{
   int a[100000];

   #pragma omp parallel for
   for (int i = 0; i < 100000; i++) {
      a[i] = 2 * i;
   }

   return 0;
}
```

- Este exemplo é embaraçosamente paralelo e depende apenas do valor de i

- O sinalizador parallel for do OpenMP diz para dividir esta tarefa entre seus threads de trabalho
- Cada thread receberá uma versão única e privada da variável
- P. ex., com dois threads de trabalho, um thread pode receber uma versão de i que vai de 0 a 49999, enquanto o segundo recebe uma versão que vai de 50000 a 99999