>>> Programação Orientada a Objetos (POO)

... Programação de Alto Desempenho (HPC)

Prof: André de Freitas Smaira

[~]$ _

Como conseguir **bom desempenho** nos seus programas?

[~]$ _

>>> Importância

* Muitos cálculos?
* Muitos dados a serem processados?
* Análise estatística?
* Pouco tempo disponível para ter resultados?
* A resposta mais relevante é: **procure um bom algoritmo**

[~]$ _

>>> Desempenho não é fácil...

* Muitos elementos a considerar:
  * Estrutura da CPU
  * Estrutura do sistema de memória
  * Estrutura do sistema de entrada e saída
  * Características da aplicação
* O compilador se intromete
* A CPU se intromete
* A memória se intromete
* O disco se intromete
* O sistema operacional se intromete
* Outras aplicações simultâneas se intrometem
* ...
* Difícil de prever o efeito de uma alteração no código.
* Difícil de prever qual parte do código tem maior impacto no desempenho.

[~]$ _

>>> Papéis do Paralelismo

* Acelera as computações
* Provê múltiplos caminhos para os dados
* Melhora acesso a dados armazenados (memória e disco)
* Escalabilidade
* Desempenho com melhor custo
* Consumo de energia
* Disponibilidade e tolerância a falhas

[~]$ _

>>> Como programar?

* Usar **estruturas de dados e algoritmos** que tenham bom desempenho para os tamanhos de problema que serão usados
* Durante a escrita do programa, não utilizar código que poderá gerar problemas de desempenho
* Executar o código e **verificar o desempenho**
* Desempenho **apropriado**? O código está pronto
* Se o desempenho **não é apropriado**, prosseguir
* Verificar que **parte**(s) do código são problemas de desempenho
* Verificar **qual o problema** dessas partes
* Propor uma solução para os problemas encontrados
* Testar a solução em um **caso simples**
* Se melhora o **suficiente**, altera o código
* Se **não funciona**, tenta **outra solução**
* **Repete** até o desempenho do programa como um todo estar **apropriado**

[~]$ _

>>> Como NÃO programar?

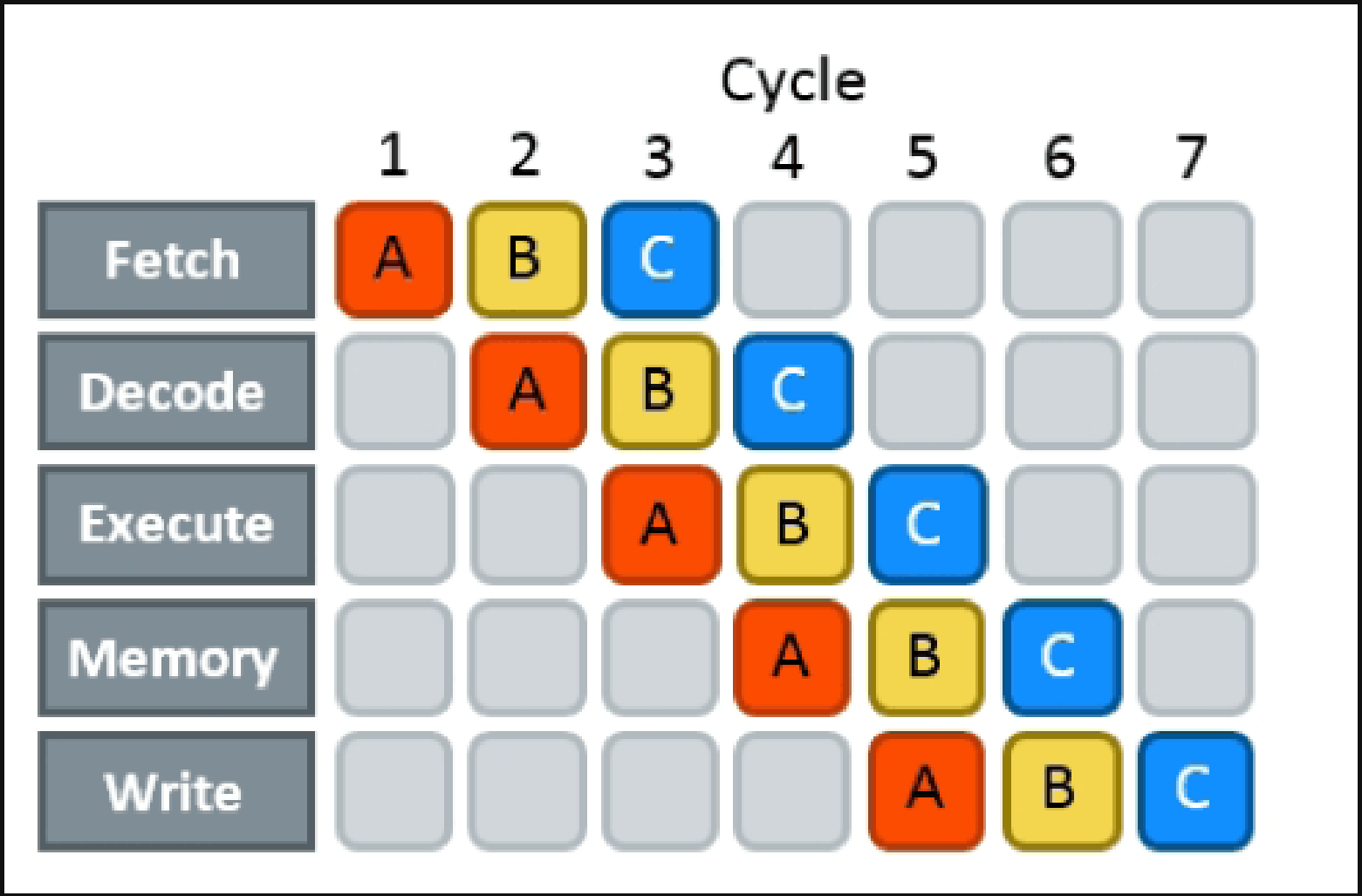
* Colocar **códigos mais complicados** "porque assim é mais eficiente"
* **Evitar** usar bibliotecas ou **abstrações mais altas** da linguagem "porque meu código de baixo nível é mais eficiente".
* Usar a **primeira estrutura de dados ou o primeiro algoritmo que vier na cabeça**, sem antes pensar se não existem algoritmos/estruturas melhores com implementação já pronta na biblioteca.
* **Otimizar** partes que estão lentas fazendo **alterações de baixo nível** no código (reorganização de operações) ao invés de repensando os algoritmos/estruturas de dados.
* **Não fazer testes de desempenho** antes de implementar uma "otimização".

[~]$ _

>>> Características

* Paralelismo é **difícil**:
  * A programação é **muito mais complicada**
  * **Nem sempre** traz os **benefícios esperados**
* Portanto: **Evite paralelismo sempre que possível**!
  * Se seu programa **já tem bom desempenho, não paralelize**
  * Se o programa **tem desempenho ruim**, tente resolver **melhorando algoritmos/estruturas de dados** ou elementos arquiteturais (por exemplo, melhor uso de cache).
  * Se isso **não resolveu**, procure resolver com a forma de **paralelização mais simples possível**
  * Use soluções mais **complexas apenas quando estritamente necessário**

>>> Nível de Processador



* Limitado pelo **estágio mais lento**
* Quanto maior a profundidade da pipeline, maior o **problema de uma prédição mal feita**

[~]$ _

>>> Cache

  * Vamos medir a importância de se saber sobre a **memória cache**

```
int **cria_matriz(int n, bool fill=true) {
    int **m = new int*[n];
    for(int i=0; i<n; i++) {
        m[i] = new int[n];
        if(fill) for(int j=0; j<n; j++);
                    m[i][j] = i*n+j;
    }
    return m;
}

void free_matriz(int **m, int n) {
    for(int i=0; i<n; i++)
        delete[] m[i];
    delete[] m;
}
```

[~]$ _

>>> Cache

Soma de colunas

```
#include<iostream>
#include<chrono>

int main() {
    int n, i, j, k;
    std::cin >> n;
    int **a = cria_matriz(n);
    int *c = new int[n];
    std::cout << "Matrizes " << n << "x" << n << std::endl;
    std::chrono::high_resolution_clock::time_point t0, t1;
    std::chrono::duration<double, std::milli> dt;
    t0 = std::chrono::high_resolution_clock::now();
    for(i = 0; i < n; i++) {
        c[i] = 0.0;
        for(j = 0; j < n; j++) c[i] += a[j][i];
    }
    t1 = std::chrono::high_resolution_clock::now();
    dt = t1 - t0;
    std::cout << "Soma por Coluna: " << dt.count() << " ms" << std::endl;

    t0 = std::chrono::high_resolution_clock::now();
    for(i = 0; i < n; i++) c[i] = 0.0;
    for(j = 0; j < n; j++)
        for (i = 0; i < n; i++) c[i] += a[j][i];
    t1 = std::chrono::high_resolution_clock::now();
    dt = t1 - t0;
    std::cout << "Soma por Linha: " << dt.count() << " ms" << std::endl;

    free_matriz(a, n); free_matriz(b, n); delete[] c;
    return 0;
}
```

[~]$ _

>>> Cache - Resultados

Matrizes 1000x1000
Soma por Coluna: 56.0341 ms
Soma por Linha: 27.4854 ms

Matrizes 3000x3000
Soma por Coluna: 1.39417 s
Soma por Linha: 0.538786 s

Matrizes 10000x10000
Soma por Coluna: 23.0592 s
Soma por Linha: 4.5065 s

Matrizes 20000x20000
Soma por Coluna: 200.754 s
Soma por Linha: 17.2178 s

[~]$ _

>>> Cache

Multiplicação Matricial

```cpp
#include<iostream>
#include<chrono>
int main() {
    int n, i, j, k;
    std::cin >> n;
    int **a = cria_matriz(n), **b = cria_matriz(n);
    int **c = cria_matriz(n, false);
    std::cout << "Matrizes " << n << "x" << n << std::endl;
    std::chrono::high_resolution_clock::time_point t0, t1;
    std::chrono::duration<double, std::milli> dt;
    t0 = std::chrono::high_resolution_clock::now();
    for(i=0; i<n; i++)
        for(j=0; j<n; j++) {
            c[i][j] = 0;
            for(k=0; k<n; k++) c[i][j] += a[i][k] * b[k][j];
        }
    t1 = std::chrono::high_resolution_clock::now();
    dt = t1 - t0;
    std::cout << "Multiplicação Direta: " << dt.count() << " ms" << std::endl;

    t0 = std::chrono::high_resolution_clock::now();
    for(i=0; i<n; i++) for(j=0; j<n; j++) std::swap(b[i][j], b[j][i]);
    for(i=0; i<n; i++)
        for(j=0; j<n; j++) {
            c[i][j] = 0;
            for(k=0; k<n; k++) c[i][j] += a[i][k] * b[j][k];
        }
    for(i=0; i<n; i++) for(j=0; j<n; j++)  std::swap(b[i][j], b[j][i]);
    t1 = std::chrono::high_resolution_clock::now();
    dt = t1 - t0;
    std::cout << "Multiplicação com Transposição: " << dt.count() << " ms" << std::endl;
    free_matriz(a, n); free_matriz(b, n); free_matriz(c, n);
    return 0;
}
```

~]$ _

>>> Cache - Resultados

Matrizes 3x3
Multiplicação Direta: 0.001162 ms
Multiplicação com Transposição: 0.002138 ms

Matrizes 10x10
Multiplicação Direta: 0.016626 ms
Multiplicação com Transposição: 0.022346 ms

Matrizes 100x100
Multiplicação Direta: 129.069 ms
Multiplicação com Transposição: 155.526 ms

[~]$ _

>>> Cache - Resultados

Matrizes 200x200
Multiplicação Direta: 535.28 ms
Multiplicação com Transposição: 579.422 ms

Matrizes 300x300
Multiplicação Direta: 1946.78 ms
Multiplicação com Transposição: 1896.26 ms

Matrizes 400x400
Multiplicação Direta: 4713.81 ms
Multiplicação com Transposição: 5305.93 ms

Matrizes 500x500
Multiplicação Direta: 11572.7 ms
Multiplicação com Transposição: 8370.1 ms

[~]$ _

>>> Cache - Resultados

Matrizes 600x600
Multiplicação Direta: 25100.6 ms
Multiplicação com Transposição: 12377 ms

Matrizes 700x700
Multiplicação Direta: 41629.9 ms
Multiplicação com Transposição: 21933.1 ms

Matrizes 800x800
Multiplicação Direta: 62369.6 ms
Multiplicação com Transposição: 35835.2 ms

Matrizes 900x900
Multiplicação Direta: 91750.7 ms
Multiplicação com Transposição: 50880 ms

[~]$ _

>>> Cache - Resultados

Matrizes 1000x1000
Multiplicação Direta: 134525 ms
Multiplicação com Transposição: 68618.3 ms

Matrizes 1200x1200
Multiplicação Direta: 202156 ms
Multiplicação com Transposição: 120244 ms

Matrizes 1500x1500
Multiplicação Direta: 614938 ms
Multiplicação com Transposição: 260595 ms

Matrizes 4000x4000
Multiplicação Direta: 238.557 min
Multiplicação com Transposição: 70.5077 min

[~]$ _

>>> Multithreading

Qual o **problema** aqui?

```
for(i = 0; i < n; i++)
    c[i] = dot_product(get_row(a, i), b);
```

E como **resolver**?

```
// Apenas explicação teórica
for (i = 0; i < n; i++)
    c[i] = create_thread(dot_product, get_row(a, i), b);
```

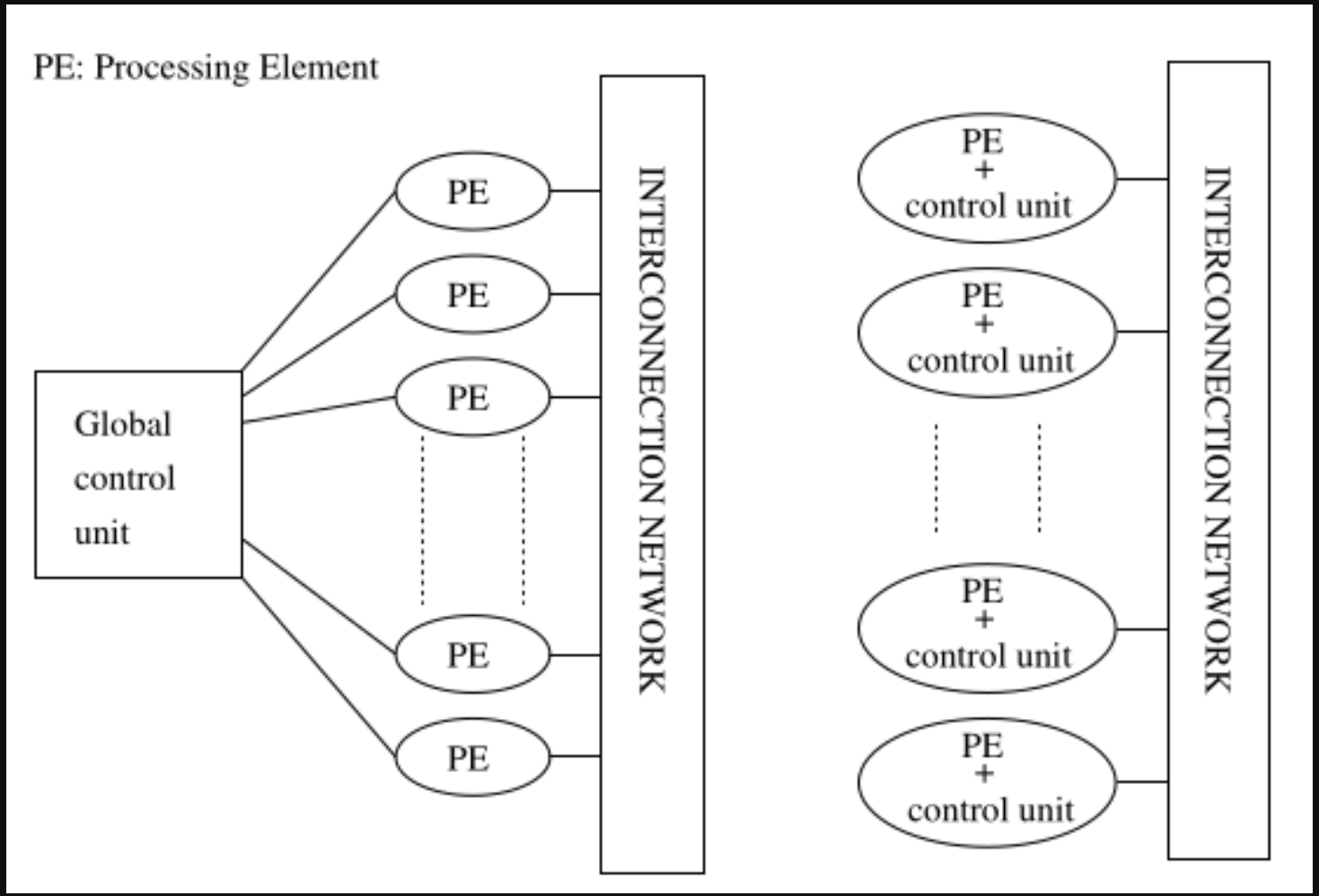
Por que isso resolve?

[~]$ _

>>> SIMD e MIMD

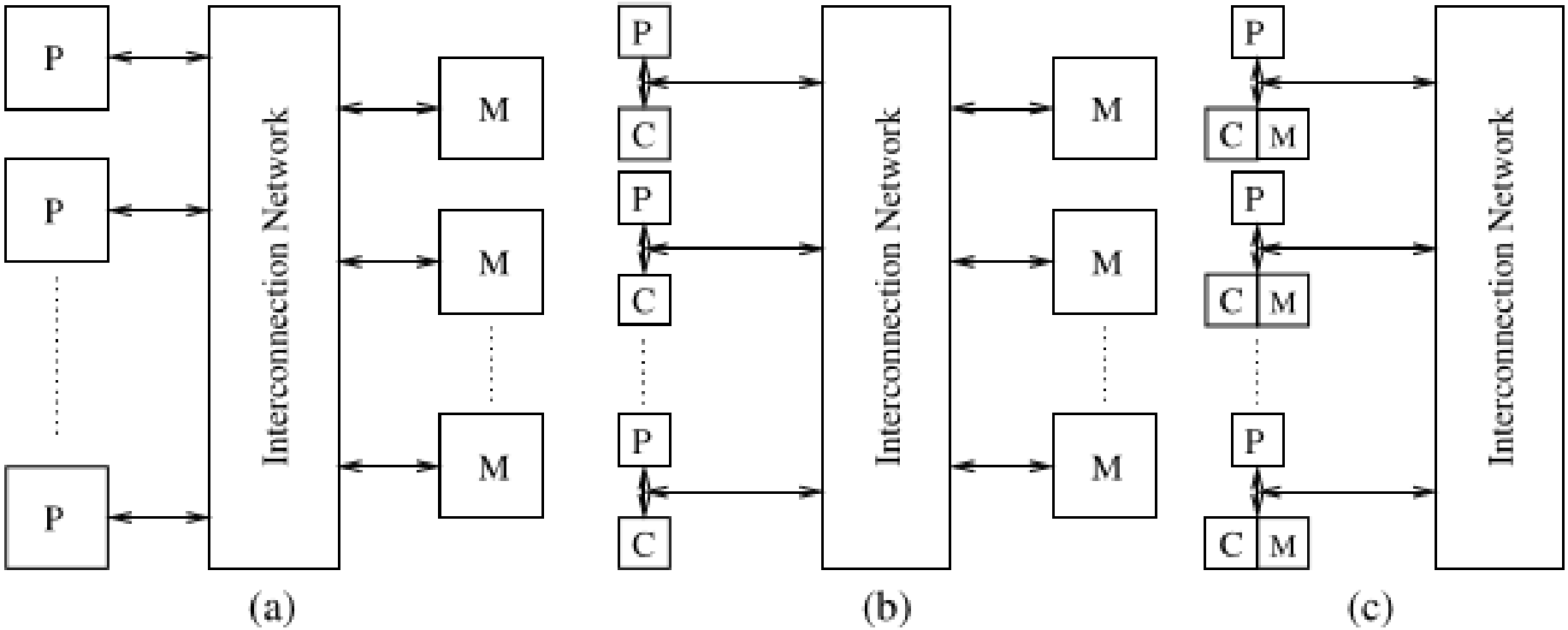
SIMD = Single Intruction Multiple Data

MIMD = Multiple Instruction Multiple Data



[~]$ _

>>> Memória


P
P
P
Interconnection Network
M
M
M
(a)
P
C
P
C
P
C
Interconnection Network
M
M
M
(b)
P
C
M
P
C
M
P
C
M
Interconnection Network
(c)

[~]$ _

>>> Tipos de Rede

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

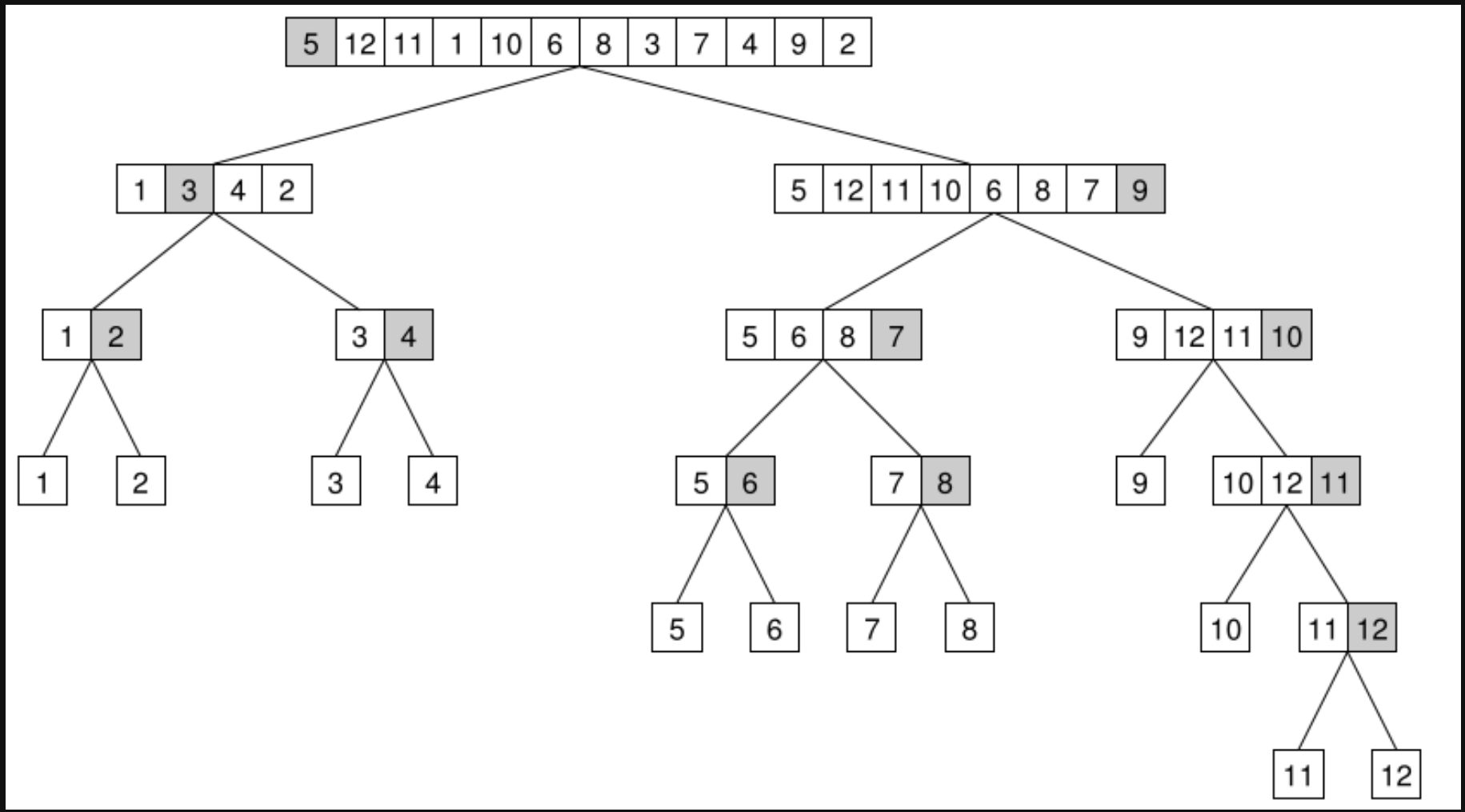
[~]$ _

>>> Conceitos importantes

* **Caminho Crítico**: Ramo da paralelização que demora mais tempo
* **Mapeamento**: Distribuição dos processos para os dados

>>> Técnicas de decomposição

* **Decomposição rescursiva**: Cada processo é dividido em subprocessos



[~]$ _

>>> Técnicas de decomposição

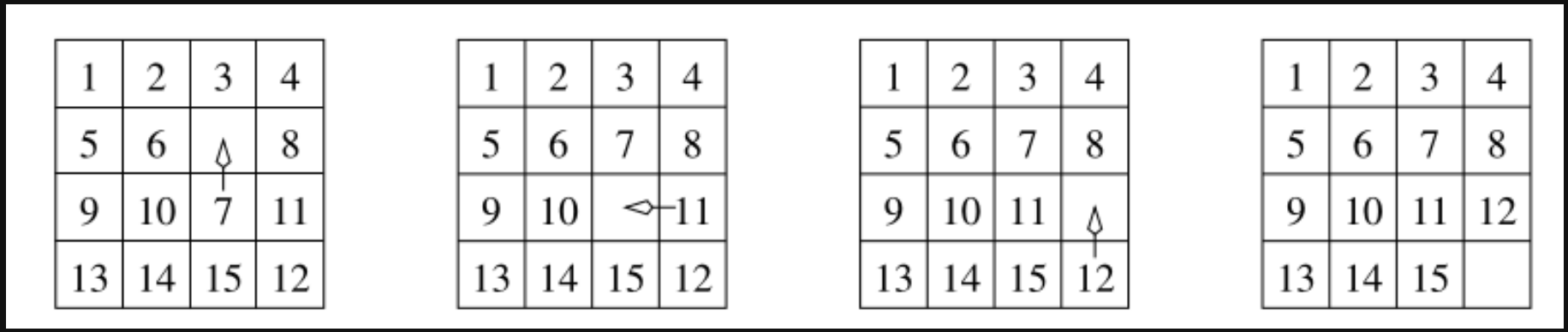
* **Decomposição de dados**: Cada processo faz as contas de uma parte dos dados
  * Decomposição pela saída

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

  * Decomposição pela entrada

>>> Técnicas de decomposição

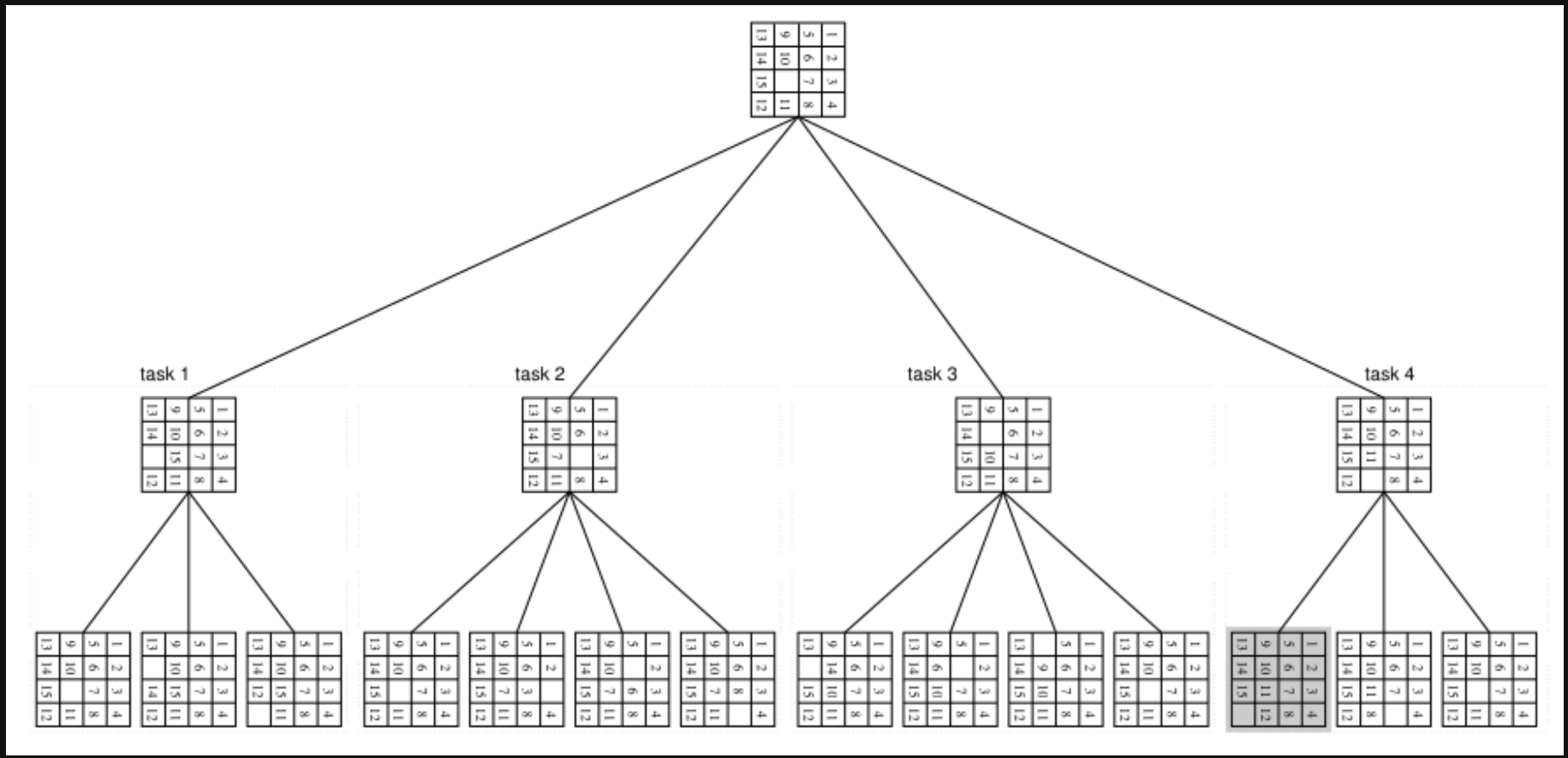
* **Decomposição exploratória**: Cada processo é dividido em subprocessos
  * Problemas de otimização
  * Provas de teoremas
  * Jogos
  * etc



[~]$ _

>>> Técnicas de decomposição

* **Decomposição exploratória**: Cada processo é dividido em subprocessos



[~]$ _

>>> Técnicas de decomposição

* **Decomposição especulativa**: Não se sabe a princípio as dependências entre processos
* **Decomposição híbrida**: Duas ou mais das anteriores ao mesmo tempo

[~]$ _

>>> Passagem de Mensagem

* p processadores com memórias exclusivas
* Comunicação assíncrona
* SPMD (Single Program Multiple Data)
* MPI: Biblioteca de C

MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of the calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.

[~]$ _

>>> MPI

```
int MPI_Init(int *argc, char ***argv);
int MPI_Finalize(); // MPI_SUCCESS.
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status);
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
                  int *count);
int MPI_Barrier(MPI_Comm comm);
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int source, MPI_Comm comm);
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int target,
               MPI_Comm comm)
```

[~]$ _

>>> mpi.h

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

int main(int argc, char *argv[])
{
    int npes, myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d, Hello World!\n", myrank, npes);
    MPI_Finalize();
    return 0;
}
```

Compilação: mpicc mpi.c

Execução: mpirun -np 4 ./a.out

[~]$ _

>>> Saída

```
From process 0 out of 4, Hello World!
From process 3 out of 4, Hello World!
From process 1 out of 4, Hello World!
From process 2 out of 4, Hello World!
```

[~]$ _

>>> Deadlocks

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

[~]$ _

>>> boost/mpi.h

```cpp
#include <boost/mpi.hpp>
#include <iostream>
namespace mpi = boost::mpi;

int main() {
    mpi::environment env;
    mpi::communicator world;
    int npes = world.size();
    int myrank = world.rank();
    std::cout << "From process " << myrank
              << " out of " << npes << ", Hello World!"
              << std::endl;
    return 0;
}
```

Compilação: mpic++ mpi.cpp -lboost_mpi -lboost_serialization

Execução: mpirun -np 4 ./a.out

[~]$ _

>>> Saída

```
From process 0 out of 4, Hello World!
From process 3 out of 4, Hello World!
From process 1 out of 4, Hello World!
From process 2 out of 4, Hello World!
```

[~]$ _

>>> Soma de Vetores

Sequencial

```cpp
#include <iostream>
#include <chrono>
int main() {
    int N = 500000000;
    int* A = new int[N];
    int* B = new int[N];
    int* C = new int[N];
    std::chrono::high_resolution_clock::time_point t0, t1;
    for (int i = 0; i < N; i++) { A[i] = i; B[i] = 2 * i; }
    t0 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; i++) C[i] = A[i] + B[i];
    t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> dt = t1 - t0;
    std::cout << "Tempo de execução da soma: " << dt.count()
              << " segundos" << std::endl;
    std::cout << C[N/2] << std::endl;
    delete[] A; delete[] B; delete[] C;
    return 0;
}
```

30 s

[~]$ _

>>> Soma de Vetores

MPI (4 processos -> 10 s)

```
#include <mpi.h>
#include <iostream>
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int N = 500000000;
    int *shared_A = nullptr, *shared_B = nullptr, *shared_C = nullptr;
    MPI_Win win_A, win_B, win_C;
    MPI_Win_allocate_shared(N * sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &shared_A, &win_A);
    MPI_Win_allocate_shared(N * sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &shared_B, &win_B);
    MPI_Win_allocate_shared(N * sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD, &shared_C, &win_C);
    MPI_Aint size_A; int disp_unit_A;
    MPI_Win_shared_query(win_A, 0, &size_A, &disp_unit_A, &shared_A);
    MPI_Aint size_B; int disp_unit_B;
    MPI_Win_shared_query(win_B, 0, &size_B, &disp_unit_B, &shared_B);
    MPI_Aint size_C; int disp_unit_C;
    MPI_Win_shared_query(win_C, 0, &size_C, &disp_unit_C, &shared_C);
    if (rank == 0)
        for (int i = 0; i < N; ++i) {
            shared_A[i] = i;
            shared_B[i] = 2*i;
        }
    MPI_Barrier(MPI_COMM_WORLD);
    double start_time = MPI_Wtime();
    int start = rank * (N / size);
    int end = (rank == size - 1) ? N : start + (N / size);
    for (int i = start; i < end; ++i) shared_C[i] = shared_A[i] + shared_B[i];
    MPI_Barrier(MPI_COMM_WORLD);
    double end_time = MPI_Wtime();
    double elapsed_time = end_time - start_time;
    if (rank == 0) {
[~]$ _      std::cout << "Tempo de execucao da soma: " << elapsed_time << " segundos." << std::endl;
```

>>> Soma de Vetores

Boost MPI (4 processos -> 10 s)

```cpp
#include <boost/mpi.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>
namespace mpi = boost::mpi;
namespace bip = boost::interprocess;

int main(int argc, char* argv[]) {
    mpi::environment env(argc, argv);
    mpi::communicator world;
    const int N = 500000000;
    const char* shared_memory_name = "shared_memory";
    if(world.rank() == 0)
        bip::shared_memory_object::remove(shared_memory_name);
    bip::managed_shared_memory segment;
    int *A = nullptr, *B = nullptr, *C = nullptr;
    if (world.rank() == 0) {
        segment = bip::managed_shared_memory(bip::create_only, shared_memory_name, sizeof(int) * N * 6);
        A = segment.construct<int>("A")[N](); B = segment.construct<int>("B")[N]();  C = segment.construct<int>("
        for (int i = 0; i < N; ++i) { A[i] = i; B[i] = 2 * i; }
    }
    world.barrier(); mpi::timer timer; timer.restart();
    while(world.rank() != 0 && A == nullptr) {
        segment = bip::managed_shared_memory(bip::open_only, shared_memory_name);
        A = segment.find<int>("A").first; B = segment.find<int>("B").first; C = segment.find<int>("C").first;
    }
    int chunk_size = N / world.size();
    int start = world.rank() * chunk_size;
    int end = (world.rank() == world.size() - 1) ? N : start + chunk_size;
    world.barrier();
    for (int i = start; i < end; ++i) C[i] = A[i] + B[i];
    world.barrier();
    if (world.rank() == 0) {
        std::cout << "Tempo: " << timer.elapsed() << " segundos" << std::endl;
        std::cout << "Elemento do meio de C: " << C[N / 2] << std::endl;
    }
```

[~]$ _

>>> Threads

* Portabilidade de software
* Ocultação de latência
* Agendamento e balanceamento de carga
* Fácil de programar

[~]$ _

>>> Soma de Vetores

Threads (8 threads -> 8,5 s)

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
const int N = 500000000;
void soma_parcial(int* A, int* B, int* C, int start, int end) {
    for (int i = start; i < end; ++i) C[i] = A[i] + B[i];
}

int main() {
    int* A = new int[N];
    int* B = new int[N];
    int* C = new int[N];
    for (int i = 0; i < N; ++i) { A[i] = i; B[i] = 2 * i; }
    const int num_threads = 4;
    int chunk_size = N / num_threads;
    std::vector<std::thread> threads;
    std::chrono::high_resolution_clock::time_point t0, t1;
    t0 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < num_threads; ++i) {
        int start = i * chunk_size;
        int end = (i == num_threads - 1) ? N : (i + 1) * chunk_size;
        threads.push_back(std::thread(soma_parcial, A, B, C, start, end));
    }
    for(auto& t : threads) t.join();
    t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> dt = t1 - t0;
    std::cout << "Tempo de execução: " << dt.count() << " segundos." << std::endl;
    std::cout << "Elemento do meio de C: " << C[N / 2] << std::endl;
    delete[] A; delete[] B; delete[] C;
    return 0;
}
```

[~]$ _

>>> Exclusão Mútua

Processos diferentes não podem modificar o mesmo dado ao mesmo tempo

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

// Mutex para proteger o recurso compartilhado
std::mutex mtx;

// Função que altera um recurso compartilhado
void incrementar(int& contador) {
    // Bloquear o mutex antes de modificar o recurso
    std::lock_guard<std::mutex> lock(mtx);
    ++contador;
    std::cout << "Contador: " << contador << std::endl;
}
```

[~]$ _

>>> Exclusão Mútua

Precisa de maior flexibilidade?

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

std::mutex mtx;

void incrementar(int& contador) {
    std::unique_lock<std::mutex> lock(mtx);

    ++contador;
    std::cout << "Contador: " << contador << std::endl;
    // lock.unlock();  // Caso precise liberar o mutex antes de
}
```

[~]$ _

>>> OpenMP

* Forma **simples** de paralelização
* Adicionamos **diretivas de compilação**
* O compilador troca por **implementação de threads**
  * **if (scalar expression)**: indica se de fato é para paralelizar
  * **num_threads (integer expression)**: número de threads a serem usadas
  * **private (variable list)**: lista de variáveis locais a cada thread
  * **shared (variable list)**: variáveis globais para todas as threads
  * **#pragma omp parallel**: região paralela, onde o bloco de código é executado por múltiplas threads
  * **#pragma omp for**: Paraleliza um loop, dividindo as iterações entre as threads
  * **#pragma omp critical**: Define uma seção crítica para evitar que múltiplas threads acessem um recurso ao mesmo tempo.
  * **#pragma omp barrier**: Sincroniza as threads
  * **#pragma omp sections** e **#pragma omp section**: Permite dividir o código em seções para que diferentes threads executem diferentes partes de um código.

[~]$ _

>>> OpenMP

```cpp
#include <iostream>
#include <omp.h>
#include <chrono>

int main() {
    const int N = 500000000;
    int *A = new int[N], *B = new int[N], *C = new int[N];
    for (int i = 0; i < N; ++i) { A[i] = i; B[i] = 2 * i; }
    auto t0 = std::chrono::high_resolution_clock::now();
    #pragma omp parallel for num_threads(4)
    for (int i = 0; i < N; ++i) C[i] = A[i] + B[i];
    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> dt = t1 - t0;
    std::cout << "Elemento do meio de C: " << C[N / 2] << std::endl;
    std::cout << "Tempo de execução: " << dt.count() << " segundos" <
    delete[] A; delete[] B; delete[] C;
    return 0;
}
// g++ -fopenmp openmp.cpp
// 8 segundos
```

[~]$ _

>>> Referências

* Slides de Addison Wesley

[~]$ _