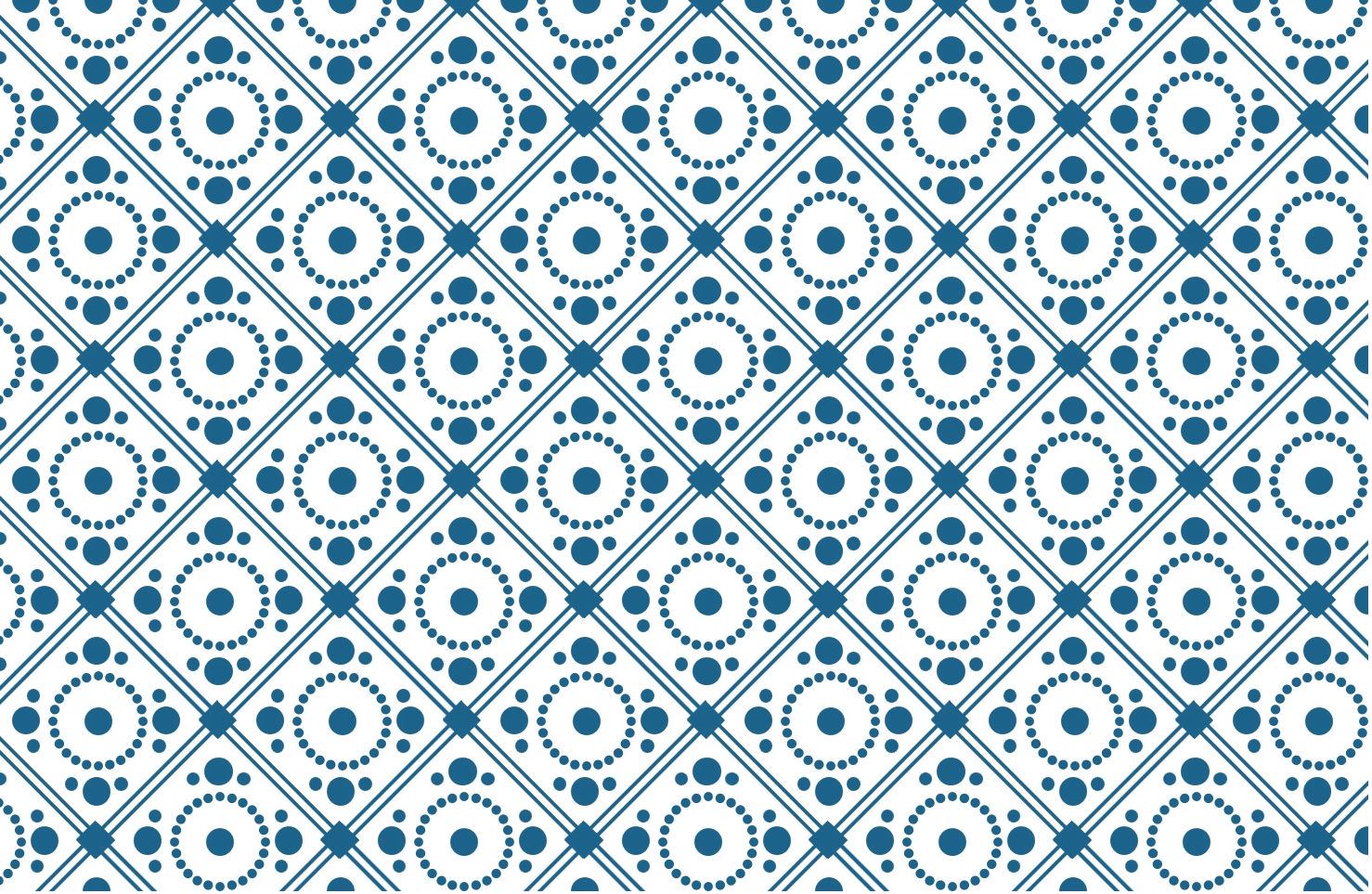


GPPD
Grupo de Processamento Paralelo e Distribuído

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

LNCC

INTRODUÇÃO À PROGRAMAÇÃO PARALELA E VETORIAL

Escola Supercomputador Santos Dumont

Matheus S. Serpa
msserpa@inf.ufrgs.br

MC-SD02-III

APRESENTAÇÃO

Matheus S. Serpa

msserpa@inf.ufrgs.br

https://www.linkedin.com/in/matheusserpa/

Formação:

Graduação em Ciência da Computação (UNIPAMPA 2015)

Mestrado em Computação (UFRGS 2018)

Período sanduíche na Université de Neuchâtel - Suiça

Doutorado em andamento em Computação (UFRGS)

Atividades:

Instrutor de Ciência de Dados na TargetTrust (TT)

Professor na Faculdade São Francisco de Assis (UNIFIN)

UFRGS
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

GRUPO DE PROCESSAMENTO PARALELO E DISTRIBUÍDO

Philippe O. A. Navaux (**Coordenador**)

Big Data Computer Architecture Fog and Edge Computing

Cloud Computing Machine Learning Oil and Gas

MATERIAL DO MINICURSO

Slides

http://www.cenapad-rj.lncc.br/tutoriais/materiais-hpc/semana-sdumont/

Exercicios no Google Colab

https://colab.research.google.com/drive/1UgP1YCfAkaa3P0WxInbnTQP9E2qiNJln

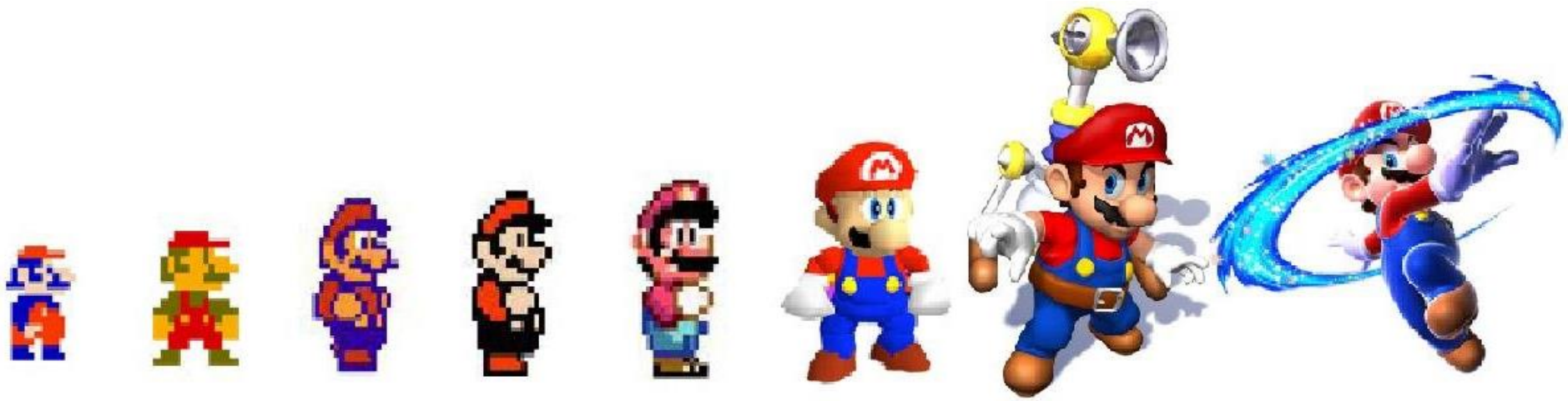
APRESENTAÇÃO DA ÁREA

POR QUE ESTUDAR PROGRAMAÇÃO PARALELA?

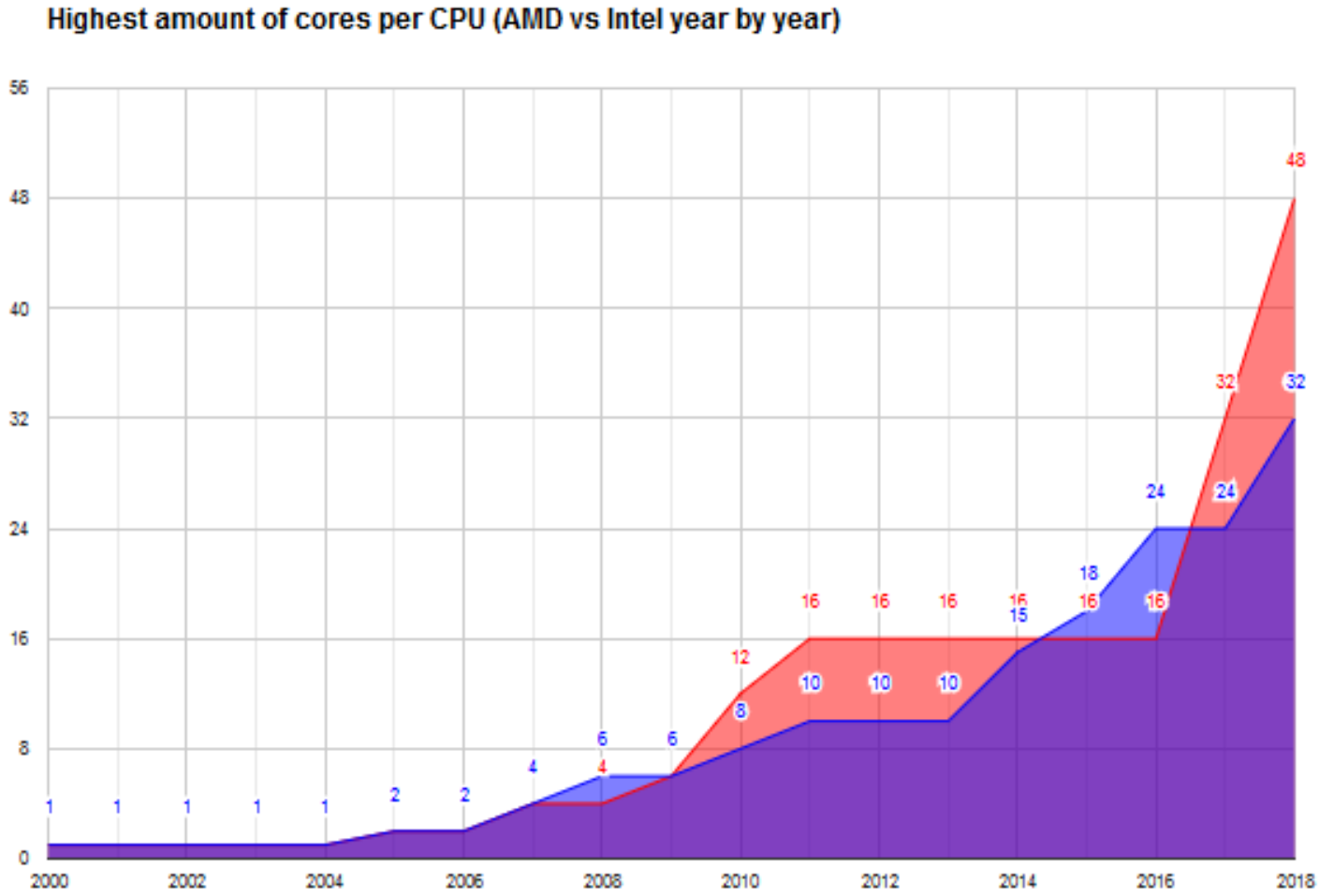
Os programas já não são rápidos o suficiente?

As máquinas já não são rápidas o suficiente?

REQUISITOS SEMPRE MUDANDO



EVOLUÇÃO DA INTEL E AMD



Onde comprar um processador single-core?

EVOLUÇÃO DA INTEL E AMD

Processador Intel® Xeon® Platinum 9282 (cache de 77 M, 2,60 GHz)

- 77 MB Intel® Smart Cache Cache
- 56 Núcleos
- 112 Segmentos
- 3.80 GHz Frequência turbo max

AMD EPYC™ 7H12

Nº de núcleos de CPU: 64

Nº de threads: 128

Clock básico: 2.6GHz

Clock de Max Boost: Até 3.3GHz

Cachê L3 total: 256MB

Package: SP3

Contagem de soquetes: 1P/2P

Versão do PCI Express: PCIe 4.0 x128

TDP / TDP Padrão: 280W

POR QUE PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reduzir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

POR QUE PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reduzir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são:

- Utilizar recursos computacionais subaproveitados.
- Ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema.
- Ultrapassar os limites físicos que atualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

OPÇÕES PARA CIENTISTAS DA COMPUTAÇÃO

1. Crie uma **nova linguagem** para programas paralelos
2. Crie um **hardware** para extrair paralelismo
3. Deixe o **compilador** fazer o trabalho sujo
   - Paralelização automática
   - Ou **crie anotações no código sequencial**
4. Use os recursos do **sistema operacional**
   - Com memória compartilhada – threads
   - Com memória distribuída – SPMD
5. Use a **estrutura dos dados** para definir o paralelismo
6. Crie uma **abstração de alto nível** – Objetos, funções aplicáveis, etc.

PRINCIPAIS MODELOS DE PROGRAMAÇÃO PARALELA

Programação em Memória Compartilhada (OpenMP, CUDA, OpenACC, OpenCL)

- Programação usando processos ou threads.
- Decomposição do domínio ou funcional com granularidade fina, média ou grossa.
- Comunicação através de **memória compartilhada**.
- Sincronização através de mecanismos de exclusão mútua.

Programação em Memória Distribuída (MPI)

- Programação usando processos distribuídos
- Decomposição do domínio com granularidade grossa.
- Comunicação e sincronização por **troca de mensagens**.

FATORES DE LIMITAÇÃO DO DESEMPENHO

Código Sequencial: existem partes do código que são inerentemente sequenciais (e.g. iniciar/terminar a computação).

Concorrência/Paralelismo: o número de tarefas pode ser escasso e/ou de difícil definição.

Comunicação: existe sempre um custo associado à troca de informação e enquanto as tarefas processam essa informação não contribuem para a computação.

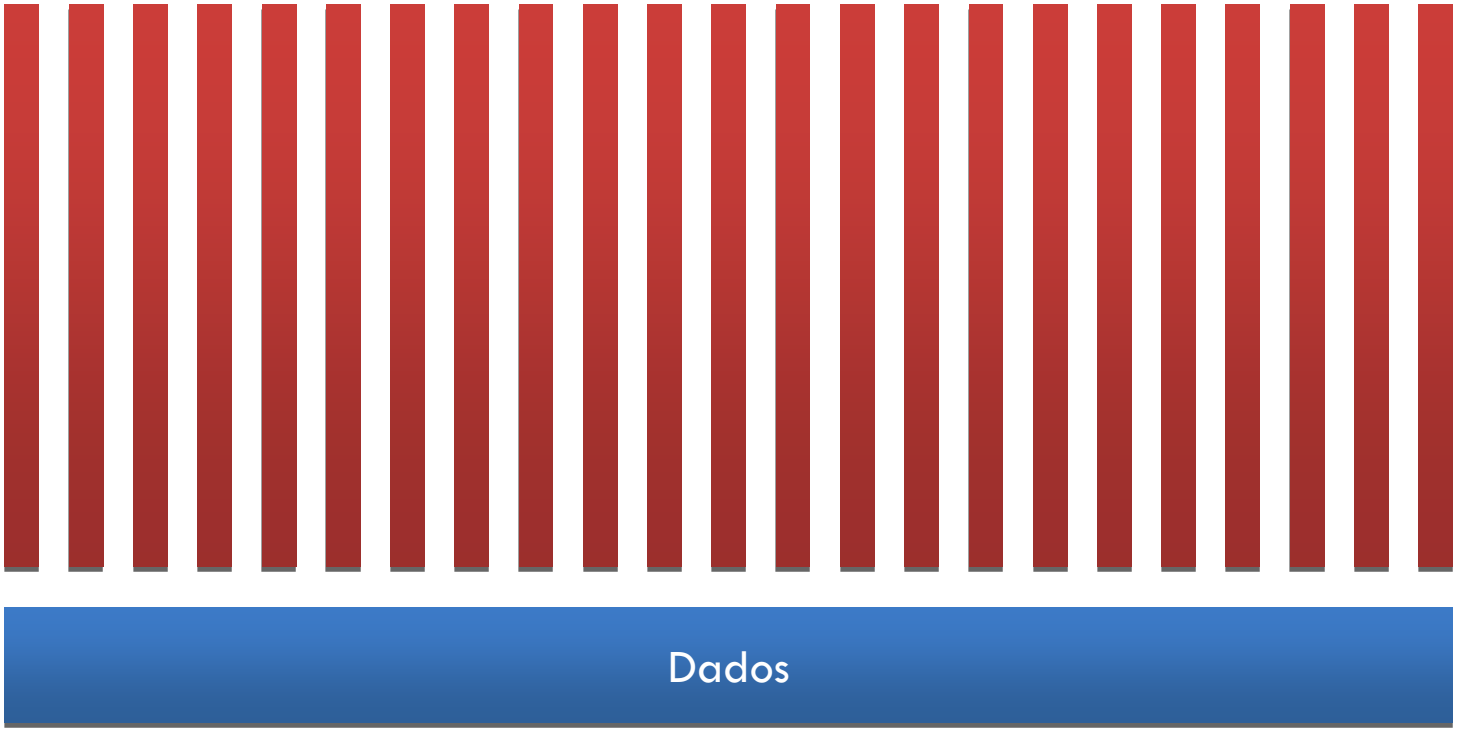
Sincronização: a partilha de dados entre as várias tarefas pode levar a problemas de contenção no acesso à memória e enquanto as tarefas ficam à espera de sincronizar não contribuem para a computação.

Granularidade: o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo (e.g. custos de criação, comunicação e sincronização).

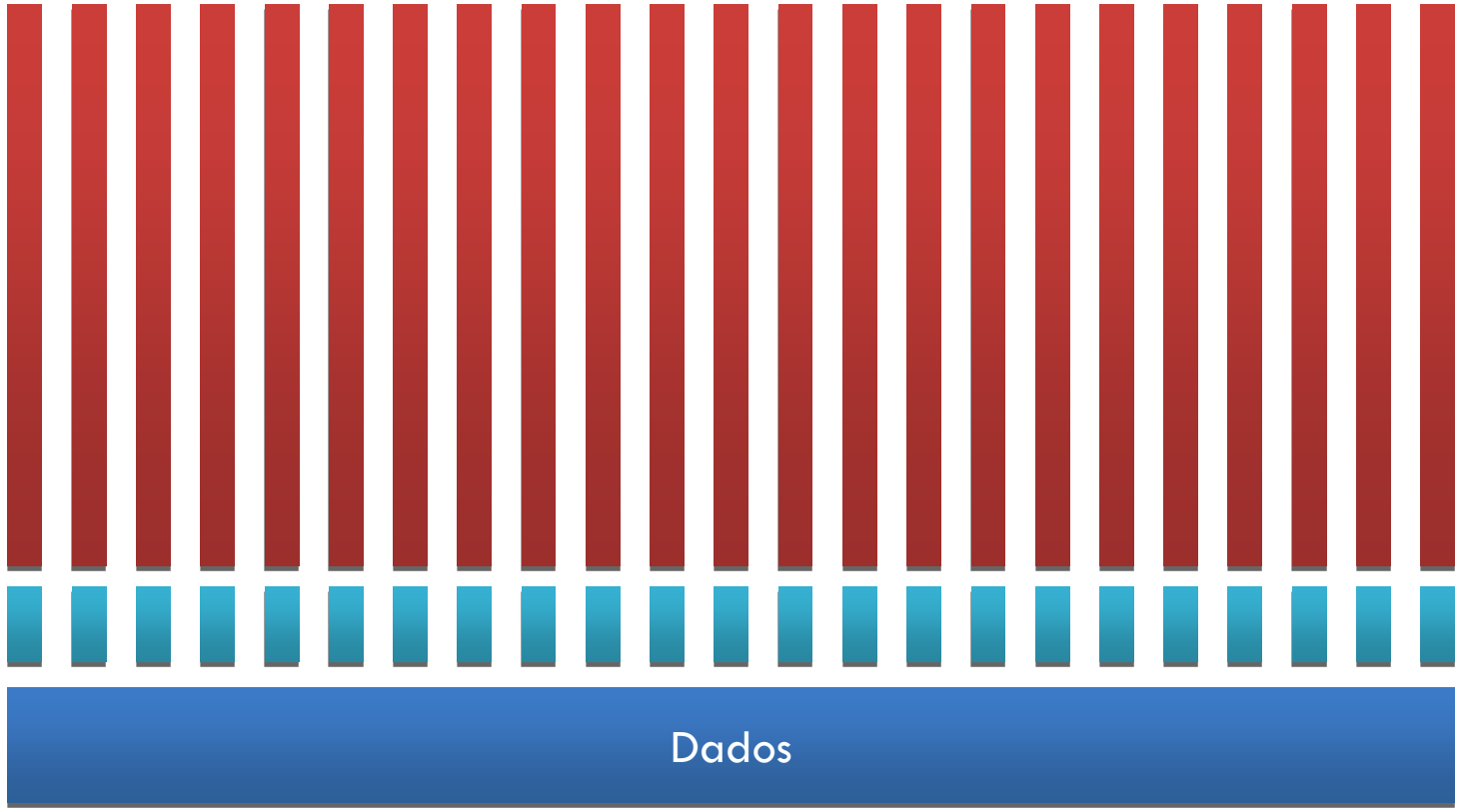
Balanceamento de Carga: ter os processadores maioritariamente ocupados durante toda a execução é decisivo para o desempenho global do sistema.

COMO IREMOS PARALELIZAR? **PENSANDO!**

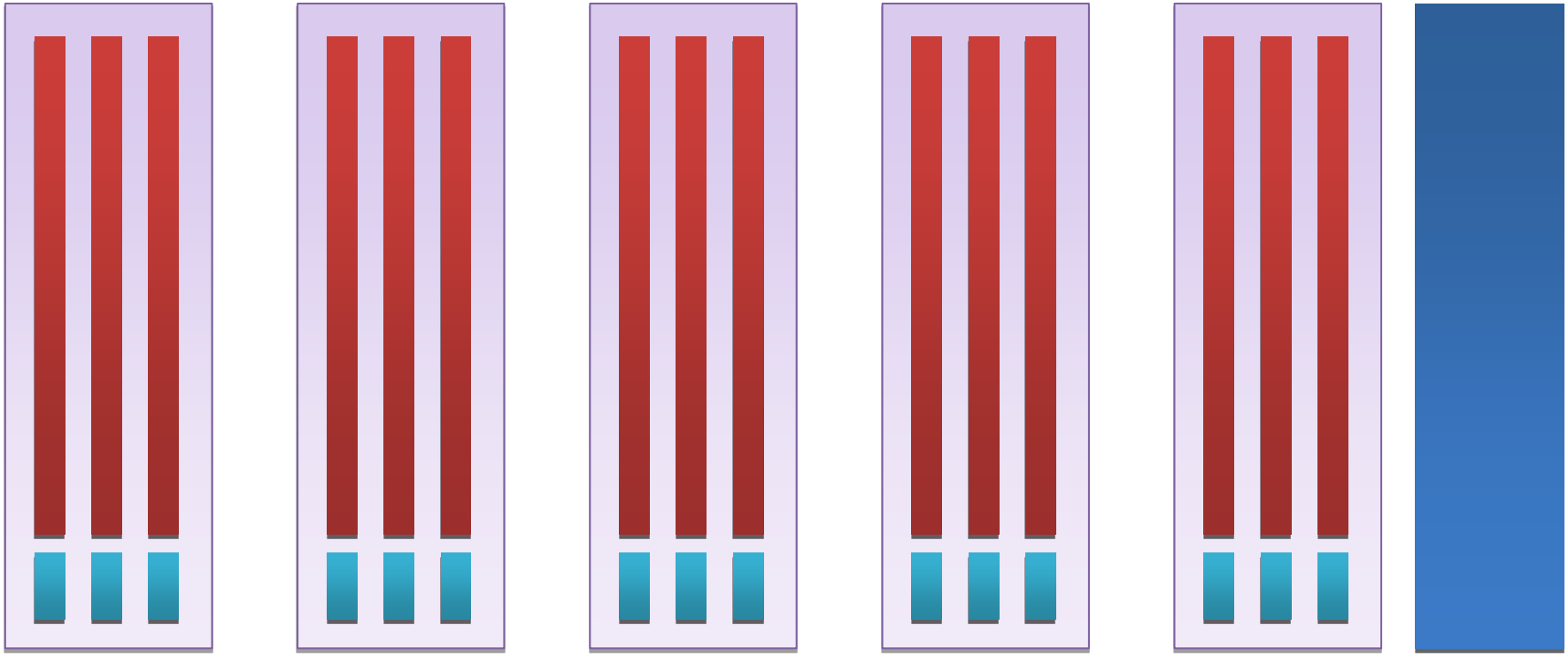
COMO IREMOS PARALELIZAR? PENSANDO!



COMO IREMOS PARALELIZAR? PENSANDO!



COMO IREMOS PARALELIZAR? PENSANDO!



Divisão e Organização lógica do nosso algoritmo paralelo

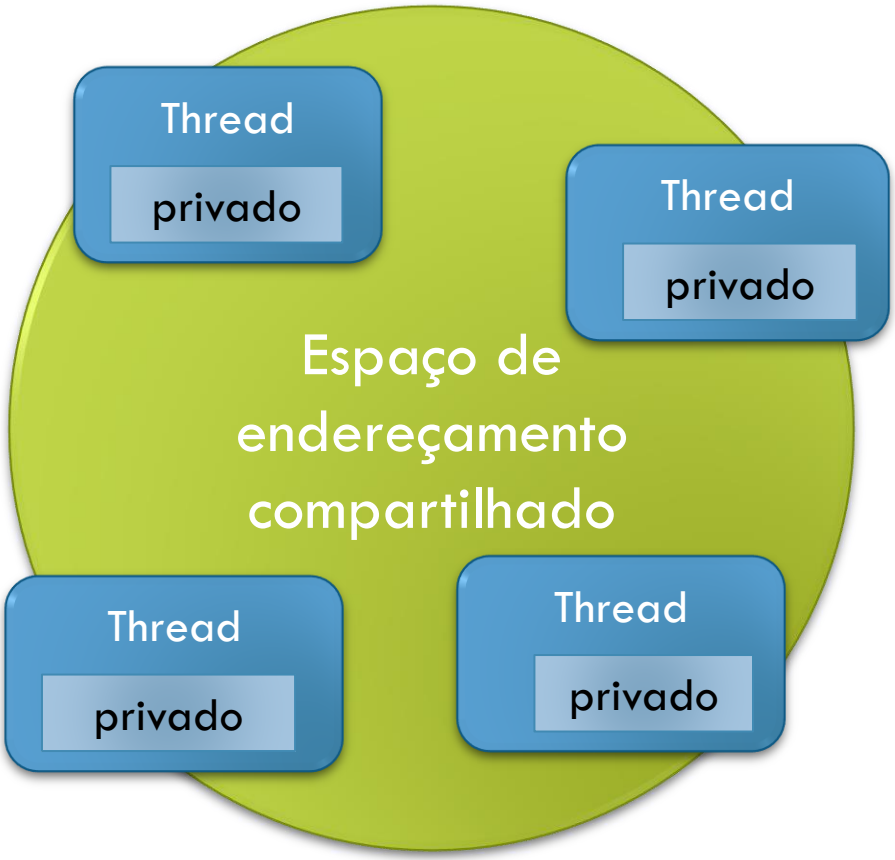
UM PROGRAMA DE MEMÓRIA COMPARTILHADA

Uma instância do programa:

Um processo e muitas threads.

Threads interagem através de leituras/escrita com o espaço de endereçamento compartilhado.

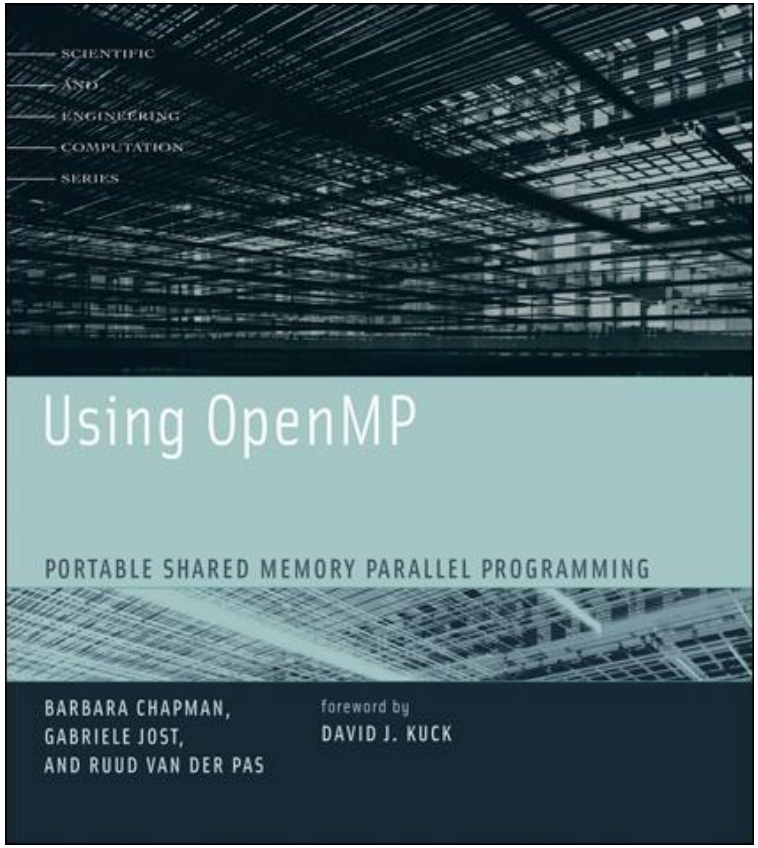
Sincronização garante a ordem correta dos resultados.

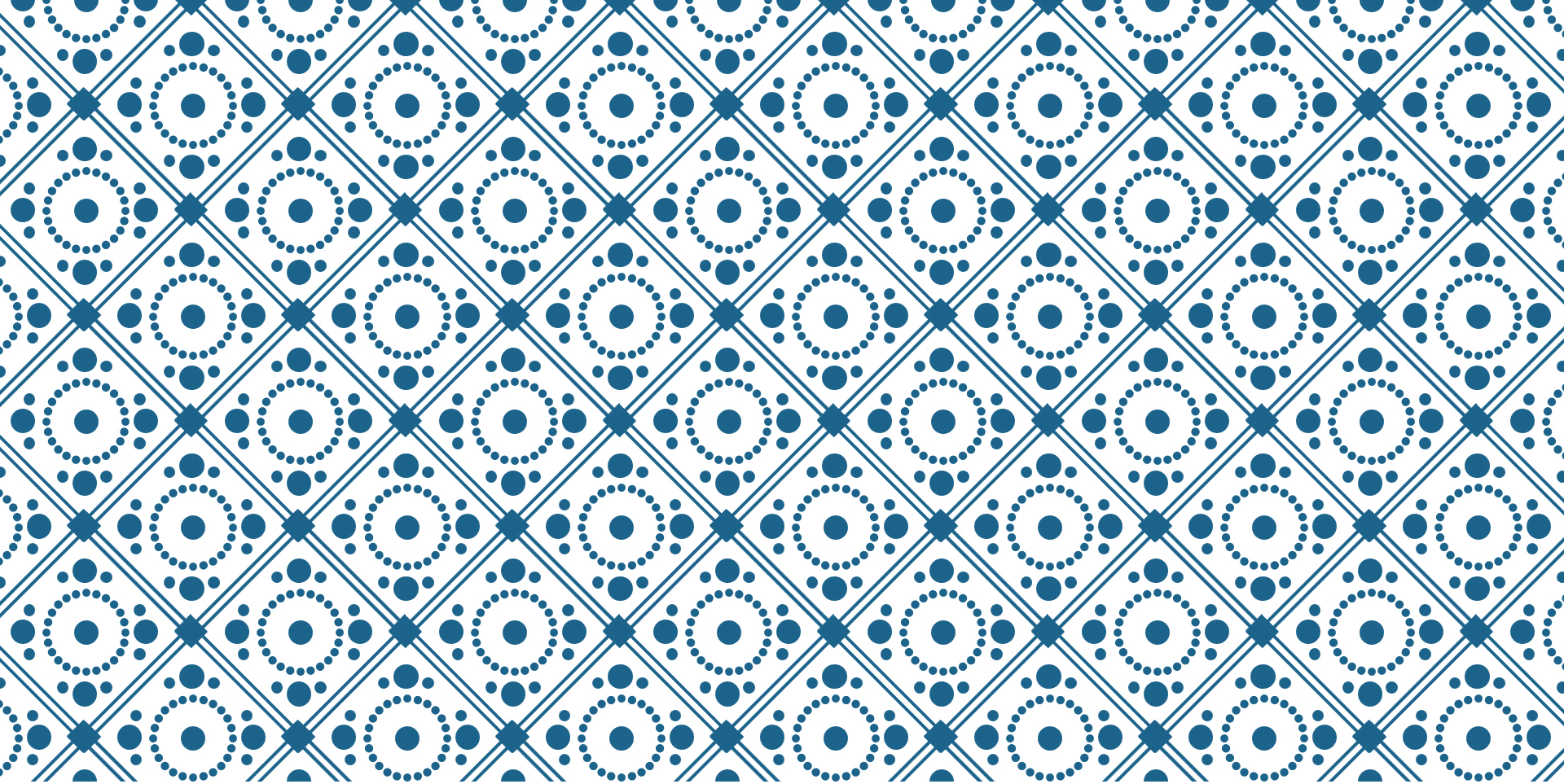


BIBLIOGRAFIA BÁSICA

Using OpenMP - Portable Shared Memory Parallel Programming
Autores: Barbara Chapman, Gabriele Jost and Ruud van der Pas
Editora: MIT Press

Ano: 2007





FUNDAMENTOS E TERMINOLOGIAS

PORQUÊ PROGRAMAÇÃO PARALELA?

Se um único computador (processador) consegue resolver um problema em N segundos, podem N computadores (processadores) resolver o mesmo problema em 1 segundo?

PORQUÊ PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reduzir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são: ???

PORQUÊ PROGRAMAÇÃO PARALELA?

Dois dos principais motivos para utilizar programação paralela são:

- **Reduzir o tempo** necessário para solucionar um problema.
- **Resolver problemas mais complexos** e de maior dimensão.

Outros motivos são:

- Tirar partido de recursos computacionais não disponíveis localmente ou subaproveitados.
- Ultrapassar limitações de memória quando a memória disponível num único computador é insuficiente para a resolução do problema.
- Ultrapassar os limites físicos de velocidade e de miniaturização que atualmente começam a restringir a possibilidade de construção de computadores sequenciais cada vez mais rápidos.

PORQUÊ PROGRAMAÇÃO PARALELA?

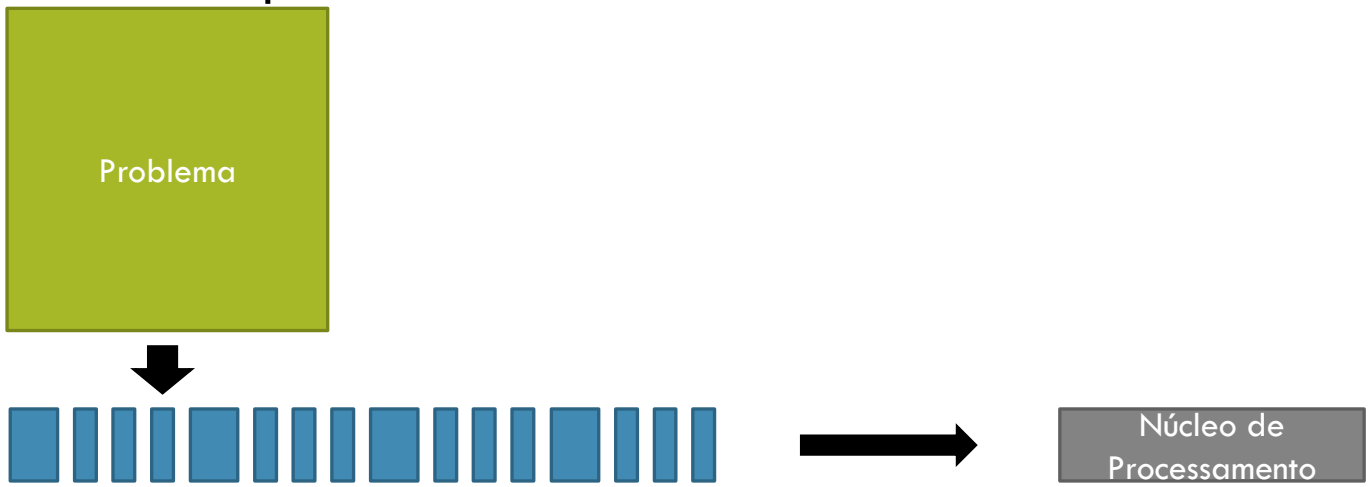
Atualmente, as aplicações que exigem o desenvolvimento de computadores cada vez mais rápidos estão por todo o lado.

Estas aplicações ou requerem um **grande poder de computação** ou requerem o processamento de **grandes quantidades de informação**. Alguns exemplos são:

- Bases de dados paralelas
- Mineração de dados (data mining)
- Serviços de procura baseados na web
- Serviços associados a tecnologias multimídia e telecomunicações
- Computação gráfica e realidade virtual
- Diagnóstico médico assistido por computador
- Gestão de grandes industrias/corporações

PROGRAMAÇÃO SEQUENCIAL

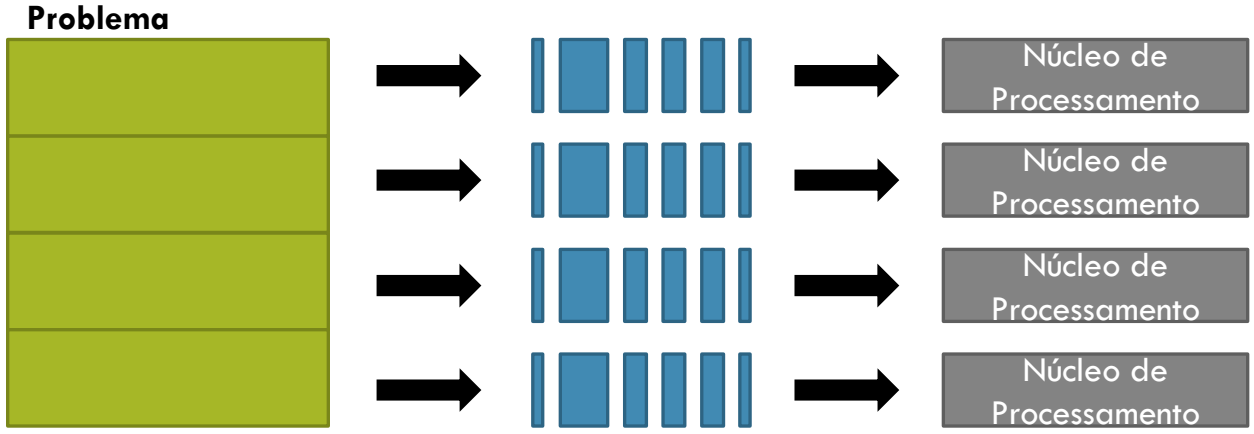
Considerado **programação sequencial** quando este é visto como uma série de instruções sequenciais que devem ser executadas num único processador.



PROGRAMAÇÃO PARALELA

Considerado **programação paralela** quando este é visto como um conjunto de partes que podem ser resolvidas concorrentemente.

Cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente.



COMPUTAÇÃO PARALELA

De uma forma simples, a computação paralela pode ser definida como o uso simultâneo de vários recursos computacionais de forma a reduzir o tempo necessário para resolver um determinado problema.

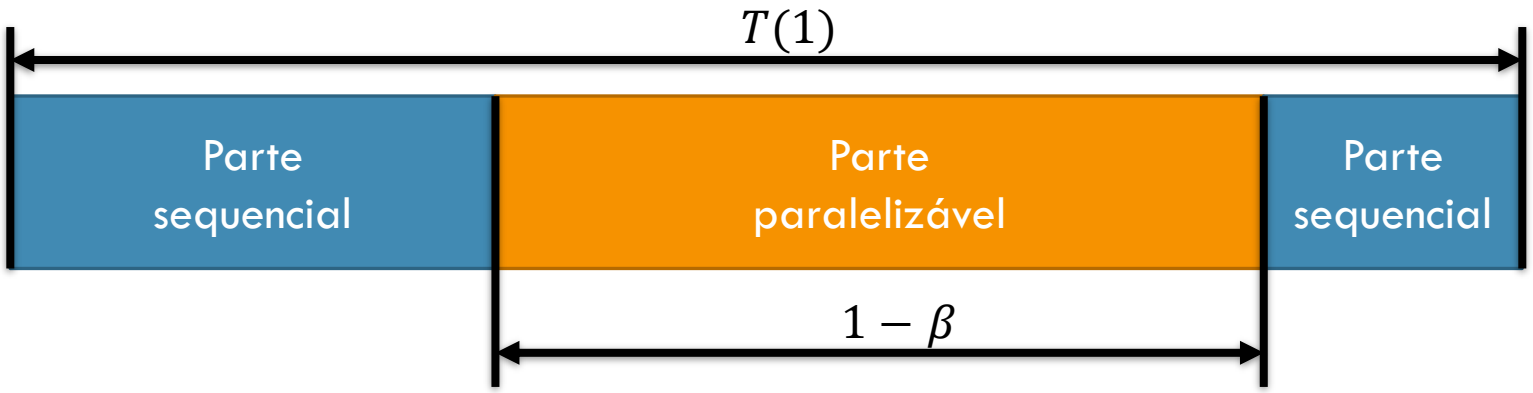
Esses recursos computacionais podem incluir:

- Um único computador com múltiplos processadores.
- Um número arbitrário de computadores ligados por rede.
- A combinação de ambos.

LEI DE AMDAHL (1967)

LEI DE AMDAHL

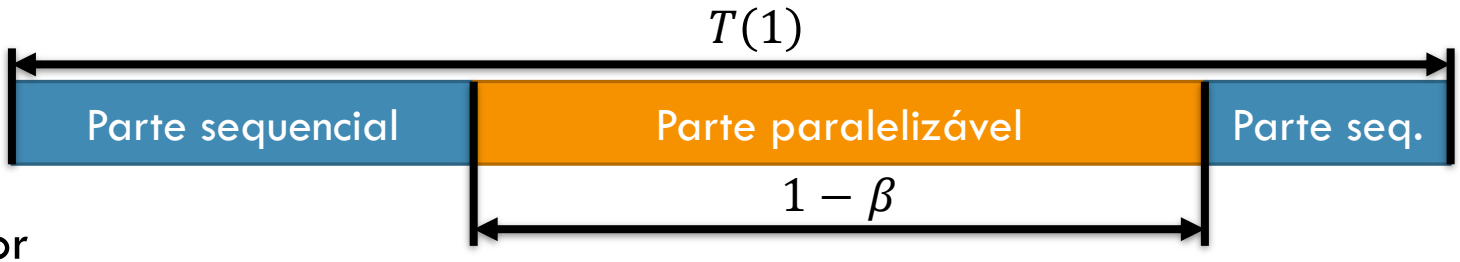
Tempo de execução em um único processador:



β = fração de código que é puramente sequencial

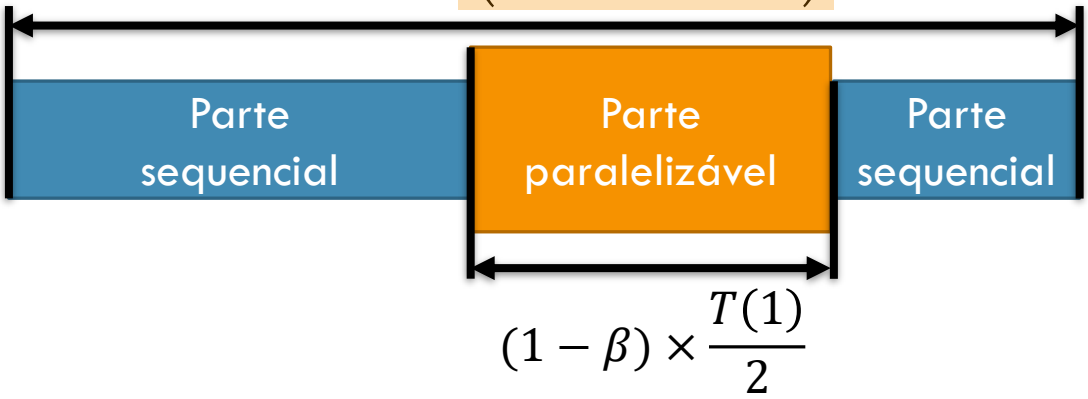
LEI DE AMDAHL

Tempo de execução em um único processador



$$T(2) = T(1) \times \beta + \left((1-\beta) \times \frac{T(1)}{2} \right)$$

Tempo de execução em 2 processadores



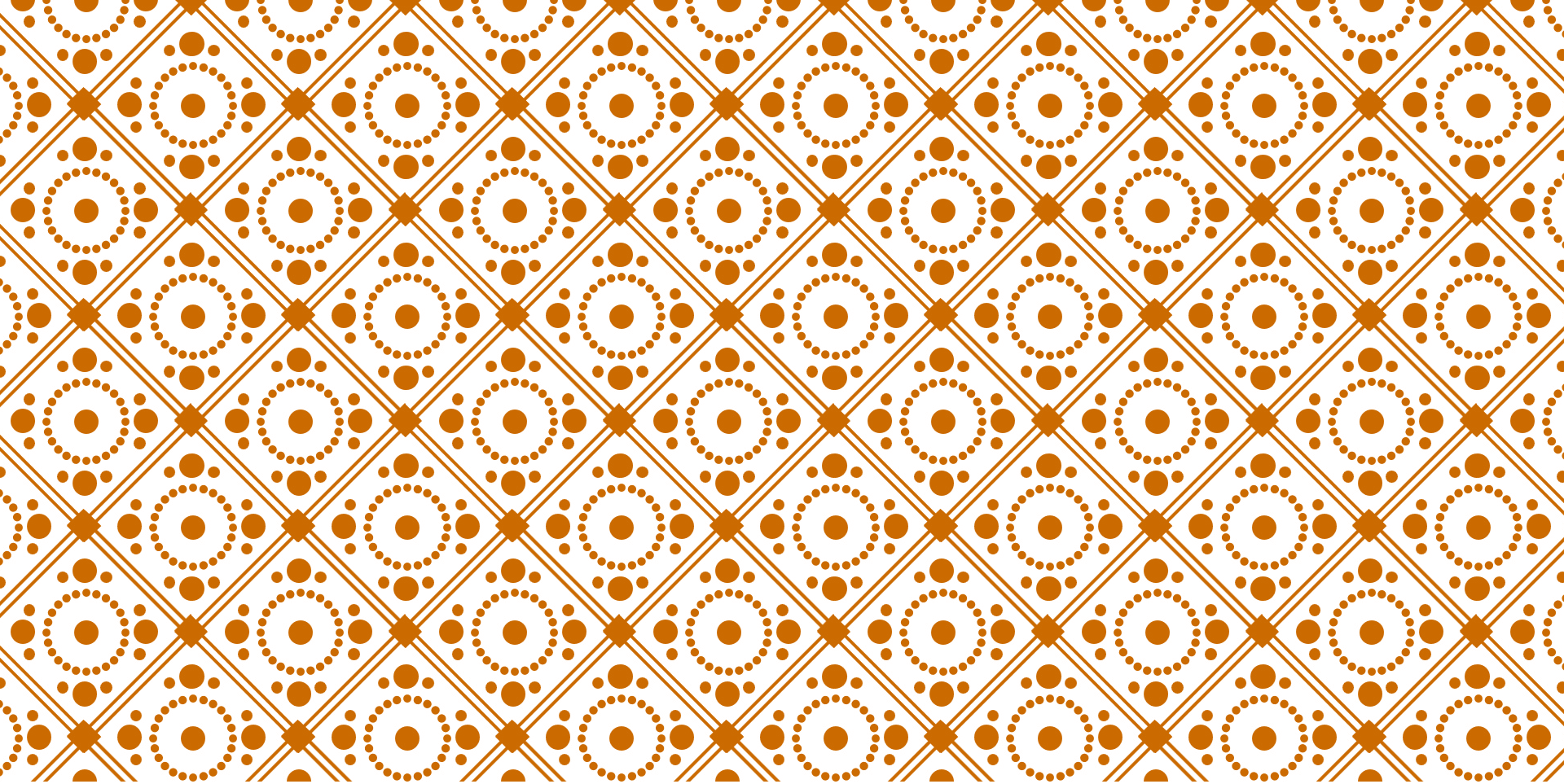
LEI DE AMDAHL

Seja $0 \leq \beta \leq 1$ a fração da computação que só pode ser realizada sequencialmente.

A lei de Amdahl diz-nos que o speedup máximo que uma aplicação paralela com p processadores pode obter é:

$$S(p) = \frac{1}{\beta + \frac{(1-\beta)}{p}}$$

A lei de Amdahl também pode ser utilizada para determinar o limite máximo de speedup que uma determinada aplicação poderá alcançar independentemente do número de processadores a utilizar (limite máximo teórico).



INTRODUÇÃO AO OPENMP

INTRODUÇÃO

OpenMP é um dos modelos de programação paralelas mais usados hoje em dia.

Esse modelo é relativamente fácil de usar, o que o torna um bom modelo para iniciar o aprendizado sobre escrita de programas paralelos.

Observações:

- Assumo que todos sabem programar em linguagem C. OpenMP também suporta Fortran e C++, mas vamos nos restringir a C.

SINTAXE BÁSICA - OPENMP

Tipos e protótipos de funções no arquivo:

#include <omp.h>

A maioria das construções OpenMP são diretivas de compilação.

#pragma omp construct [clause [clause]…]

- Exemplo:

#pragma omp parallel private(var1, var2) shared(var3, var4)

A maioria das construções se aplicam a um **bloco estruturado**.

Bloco estruturado: Um bloco com um ou mais declarações com um ponto de entrada no topo e um ponto de saída no final.

Podemos ter um **exit()** dentro de um bloco desses.

NOTAS DE COMPILAÇÃO

Linux e OS X com **gcc**:

```
gcc -fopenmp foo.c #GCC
export OMP_NUM_THREADS=40
./a.out
```

Para shell bash

Por padrão é o nº de proc. virtuais.

Também funciona no Windows!

Até mesmo no Visual Studio!

Mas vamos usar Linux ☺

FUNÇÕES

Funções da biblioteca OpenMP.

```
// Arquivo interface da biblioteca OpenMP para C/C++
#include <omp.h>

// retorna o identificador da thread.
int omp_get_thread_num();

// indica o número de threads a executar na região paralela.
void omp_set_num_threads(int num_threads);

// retorna o número de threads que estão executando no momento.
int omp_get_num_threads();

// Comando para compilação habilitando o OpenMP.
gcc –o hello hello.c –fopenmp
```

DIRETIVAS

Diretivas do OpenMP.

```
// Cria a região paralela. Define variáveis privadas e
compartilhadas entre as threads.
#pragma omp parallel private(...) shared(...)
{ // Obrigatoriamente na linha de baixo.

// Apenas a thread mais rápida executa.
#pragma omp single


}
```

EXERCÍCIO 1: HELLO WORLD

cd 1-helloWorld/ sbatch exec.batch cat XX.out

0 of 1 – hello world!

```
#include <stdio.h>

int main(){
  int myid, nthreads;



  myid = 0;

  nthreads = 1;
  printf("%d of %d – hello world!\n", myid, nthreads);

  return 0;
}
```

SOLUÇÃO 1.1: HELLO WORLD

Variáveis privadas.

cd 1-helloWorld/ sbatch exec.batch cat XX.out

```
#include <stdio.h>
#include <omp.h>
int main(){
  int myid, nthreads;

  #pragma omp parallel private(myid, nthreads)
  {
  myid = omp_get_thread_num();

  nthreads = omp_get_num_threads();
  printf("%d of %d – hello world!\n", myid, nthreads);
  }
  return 0;
}
```

```
0 of 2 – hello world!
1 of 2 – hello world!
```

SOLUÇÃO 1.2: HELLO WORLD

Variáveis privadas e compartilhadas.

cd 1-helloWorld/ sbatch exec.batch cat XX.out

```
#include <stdio.h>
#include <omp.h>
int main(){
  int myid, nthreads;

  #pragma omp parallel private(myid) shared(nthreads)
  {
  myid = omp_get_thread_num();
  #pragma omp single
  nthreads = omp_get_num_threads();
  printf("%d of %d – hello world!\n", myid, nthreads);
  }
  return 0;
}
```

```
0 of 2 – hello world!
1 of 2 – hello world!
```

SOLUÇÃO 1.3: HELLO WORLD

NUM_THREADS fora da região paralela.

cd 1-helloWorld/ sbatch exec.batch cat XX.out

```
#include <stdio.h>
#include <omp.h>
int main(){
  int myid, nthreads;

  nthreads = omp_get_num_threads();
  #pragma omp parallel private(myid) shared(nthreads)
  {
  myid = omp_get_thread_num();
  printf("%d of %d – hello world!\n", myid, nthreads);
  }
  return 0;
}
```

~~SOLUÇÃO 1.3~~: HELLO WORLD

NUM_THREADS fora da região paralela.

Não funciona.

```
#include <stdio.h>
#include <omp.h>
int main(){
  int myid, nthreads;

  nthreads = omp_get_num_threads();
  #pragma omp parallel private(myid) shared(nthreads)
  {
  myid = omp_get_thread_num();
  printf("%d of %d – hello world!\n", myid, nthreads);
  }
  return 0;
}
```

```
0 of 1 – hello world!
1 of 1 – hello world!
```

CONSTRUÇÕES DE DIVISÃO DE LAÇOS

A construção de divisão de trabalho em laços divide as iterações do laço entre as *threads* do time.

```
#pragma omp parallel private(i) shared(N)
{
 #pragma omp for
 for(i = 0; i < N; i++)
   NEAT_STUFF(i);
}
```

A variável **i** será feita privada para cada *thread* por padrão. Você poderia fazer isso explicitamente com a cláusula **private(i)**

CONSTRUÇÕES DE DIVISÃO DE LAÇOS
UM EXEMPLO MOTIVADOR

Código sequencial

```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

CONSTRUÇÕES DE DIVISÃO DE LAÇOS
UM EXEMPLO MOTIVADOR

Código sequencial

```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

Região OpenMP parallel

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  if(id == Nthrds-1) iend = N;
  for(i = istart; i < iend; i++)
    a[i] = a[i] + b[i];
}
```

CONSTRUÇÕES DE DIVISÃO DE LAÇOS
UM EXEMPLO MOTIVADOR

Código sequencial

```
for(i = 0; i < N; i++)
  a[i] = a[i] + b[i];
```

Região OpenMP parallel

```
#pragma omp parallel
{
  int id, i, Nthrds, istart, iend;
  id = omp_get_thread_num();
  Nthrds = omp_get_num_threads();
  istart = id * N / Nthrds;
  iend = (id+1) * N / Nthrds;
  if(id == Nthrds-1) iend = N;
  for(i = istart; i < iend; i++)
    a[i] = a[i] + b[i];
}
```

Região paralela OpenMP com uma construção de divisão de laço

```
#pragma omp parallel
#pragma omp for
for(i = 0; i < N; i++) a[i] = a[i] + b[i];
```

CONSTRUÇÕES PARALELA E DIVISÃO DE LAÇOS COMBINADAS

Algumas cláusulas podem ser combinadas.

```
double res[MAX]; int i;
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i < MAX; i++)
    res[i] = huge();
}
```

=

```
double res[MAX]; int i;
#pragma omp parallel for
  for(i=0; i < MAX; i++)
    res[i] = huge();
```

EXERCÍCIO 2, PARTE A: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;


  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

FUNÇÕES

Funções da biblioteca OpenMP.

```
// Arquivo interface da biblioteca OpenMP para C/C++
#include <omp.h>

// retorna o identificador da thread.
int omp_get_thread_num();

// indica o número de threads a executar na região paralela.
void omp_set_num_threads(int num_threads);

// retorna o número de threads que estão executando no momento.
int omp_get_num_threads();

// Comando para compilação habilitando o OpenMP.
gcc –o hello hello.c –fopenmp
```

DIRETIVAS

Diretivas do OpenMP.

```
// Cria a região paralela. Define variáveis privadas e
compartilhadas entre as threads.
#pragma omp parallel private(...) shared(...)
{ // Obrigatoriamente na linha de baixo.

// Apenas a thread mais rápida executa.
#pragma omp single

}

#pragma omp for
```

SOLUÇÃO 2.1, PARTE B: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

~~SOLUÇÃO 2.1~~, PARTE B: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

~~SOLUÇÃO 2.1~~, PARTE B: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
 // RACE CONDITION
    sum += v[i]; // ler sum, v[i]; somar; escrever sum;

  return sum

}
```

COMO AS THREADS INTERAGEM?

OpenMP é um modelo de *multithreading* de memória compartilhada.

- Threads se comunicam através de variáveis compartilhadas.

Compartilhamento não intencional de dados causa **condições de corrida**.

- Condições de corrida: quando a saída do programa muda quando a threads são escalonadas de forma diferente.

Apesar de este ser um aspectos mais poderosos da utilização de threads, também pode ser um dos mais problemáticos.

O problema existe quando dois ou mais *threads* tentam acessar/alterar as mesmas estruturas de dados (condições de corrida).

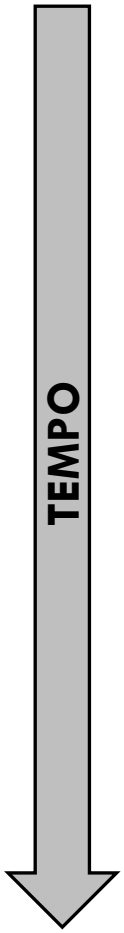
Para controlar condições de corrida:

- Usar sincronização para proteger os conflitos por dados

Sincronização é cara, por isso:

- Tentaremos mudar a forma de acesso aos dados para minimizar a necessidade de sincronizações.

CONDIÇÕES DE CORRIDA: EXEMPLO



Thread 0	Thread 1	sum
		0
Leia sum 0		0
	Leia sum 0	0
	Some 0, *5* *5*	0
Some 0, 10 10		0
	Escreva *5*, sum *5*	*5*
Escreva 10, sum 10		10 **15 !?**

CONDIÇÕES DE CORRIDA: EXEMPLO

Thread 0	Thread 1	sum
		0
	Escreva 5, sum 5	5
Escreva 10, sum 10		10

15 !?

Devemos garantir que **não importa a ordem de execução (escalonamento)**, teremos sempre um resultado consistente!

SINCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

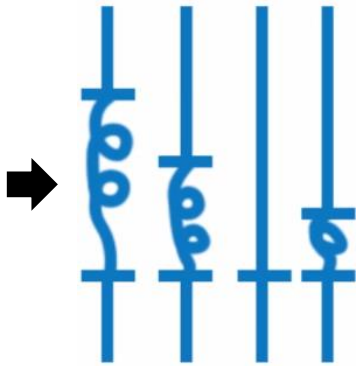
As duas formas mais comuns de sincronização são:

SINCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

As duas formas mais comuns de sincronização são:

Barreira: Cada *thread* espera na barreira até a chegada de todas as demais



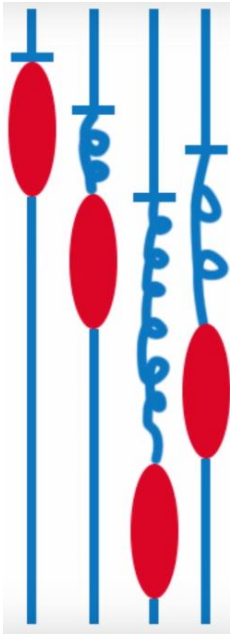
SINCRONIZAÇÃO

Assegura que uma ou mais *threads* estão em um estado bem definido em um ponto conhecido da execução.

As duas formas mais comuns de sincronização são:

Barreira: Cada *thread* espera na barreira até a chegada de todas as demais

Exclusão mútua: Define um bloco de código onde apenas uma *thread* pode executar por vez.



SINCRONIZAÇÃO: BARRIER

Barrier: Cada *thread* espera até que as demais cheguem.

```
#pragma omp parallel
{
  int id = omp_get_thread_num(); // variável privada
  A[id] = big_calc1(id);

  #pragma omp barrier

  B[id] = big_calc2(id, A);
} // Barreira implícita
```

SINCRONIZAÇÃO: CRITICAL

Exclusão mútua: Apenas uma *thread* pode entrar por vez

```
#pragma omp parallel
{
  float B; // variável privada
  int i, myid, nthreads; // variáveis privada
  myid = omp_get_thread_num();
  nthreads = omp_get_num_threads();
  for(i = myid; i < niters; i += nthreads){
    B = big_job(i); // Se for pequeno, muito overhead
    #pragma omp critical
    res += consume (B);
  }
}
```

As *threads* esperam sua vez, **apenas uma chama consume() por vez.**

SINCRONIZAÇÃO: ATOMIC

atomic prove exclusão mútua para operações específicas.

```
#pragma omp parallel
{
  double tmp, B;
  B = DOIT();
  tmp = big_ugly(B);
  #pragma omp atomic
  X += tmp;
}
```

Instruções especiais da arquitetura (se disponível)

Algumas operações aceitáveis:

```
v = x;
x = expr;
x++; ++x; x--; --x;
x op= expr;
v = x op expr;
v = x++; v = x--; v = ++x; v = --x;
```

EXERCÍCIO 2, PARTE C: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

SOLUÇÃO 2.2, PARTE C: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    #pragma omp critical
    sum += v[i];

  return sum

}
```

SOLUÇÃO 2.3, PARTE C: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    #pragma omp atomic
    sum += v[i];

  return sum

}
```

SOLUÇÃO 2.3, PARTE C: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    #pragma omp atomic
    sum += v[i];

  return sum

}
```

Qual o problema da seção crítica dentro do loop?

SOLUÇÃO 2.3, PARTE C: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    #pragma omp atomic
    sum += v[i];

  return sum

}
```

Qual o problema da seção crítica dentro do loop?

Regiões atomic – **n** vezes. **Ex.** 1 000 000 000

EXERCÍCIO 2, PARTE D: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for private(i)
  for(i = 0; i < N; i++)
    #pragma omp atomic
    sum += v[i];

  return sum

}
```

SOLUÇÃO 2.4, PARTE D: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0, sum_local;

  #pragma omp parallel private(i, sum_local)
  {
  sum_local = 0;
  #pragma omp for
  for(i = 0; i < N; i++)
    sum_local += v[i];

  #pragma omp atomic
  sum += sum_local;
  }
  return sum
}
```

SOLUÇÃO 2.4, PARTE D: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0, sum_local;

  #pragma omp parallel private(i, sum_local)
  {
  sum_local = 0;
  #pragma omp for
  for(i = 0; i < N; i++)
    sum_local += v[i];

  #pragma omp atomic
  sum += sum_local;
  }
  return sum
}
```

Regiões atomic – **nthreads** vezes. **Ex. 40** threads / vezes

EXERCÍCIO 2, PARTE E: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0, sum_local;

  #pragma omp parallel private(i, sum_local)
  {
  sum_local = 0;
  #pragma omp for
  for(i = 0; i < N; i++)
    sum_local += v[i];

  #pragma omp atomic
  sum += sum_local;
  }
  return sum
}
```

OpenMP é um modelo relativamente **fácil** de usar

REDUÇÃO

Combinação de variáveis locais de uma *thread* em uma variável única.

- Essa situação é bem comum, e chama-se **redução**.
- O suporte a tal operação é fornecido pela maioria dos ambientes de programação paralela.

DIRETIVA REDUCTION

reduction(op : list_vars)

Dentro de uma região paralela ou de divisão de trabalho:

- Será feita uma cópia local de cada variável na lista
- Será inicializada dependendo da **op** (ex. 0 para +, 1 para *).
- Atualizações acontecem na cópia local.
- Cópias locais são “reduzidas” para uma única variável original (global).

#pragma omp for reduction(* : var_mult)

EXERCÍCIO 2, PARTE E: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0, sum_local;

  #pragma omp parallel private(i, sum_local)
  {
  sum_local = 0;
  #pragma omp for
  for(i = 0; i < N; i++)
    sum_local += v[i];

  #pragma omp atomic
  sum += sum_local;
  }
  return sum
}
```

OpenMP é um modelo relativamente **fácil** de usar

SOLUÇÃO 2.5, PARTE E: VECTOR SUM

cd 2-vectorSum/ sbatch exec.batch cat XX.out

```
long long int sum(int *v, long long int N){
  long long int i, sum = 0;

  #pragma omp parallel for reduction(+ : sum)
  for(i = 0; i < N; i++)
    sum += v[i];

  return sum

}
```

VECTOR SUM

Sequencial vs. Paralelo

```
sum = 0;

for(i = 0; i < N; i++)
  sum += v[i];
```

```
sum = 0;
#pragma omp parallel for reduction(+ : sum)
for(i = 0; i < N; i++)
  sum += v[i];
```

EXERCÍCIO 3: SELECTION SORT

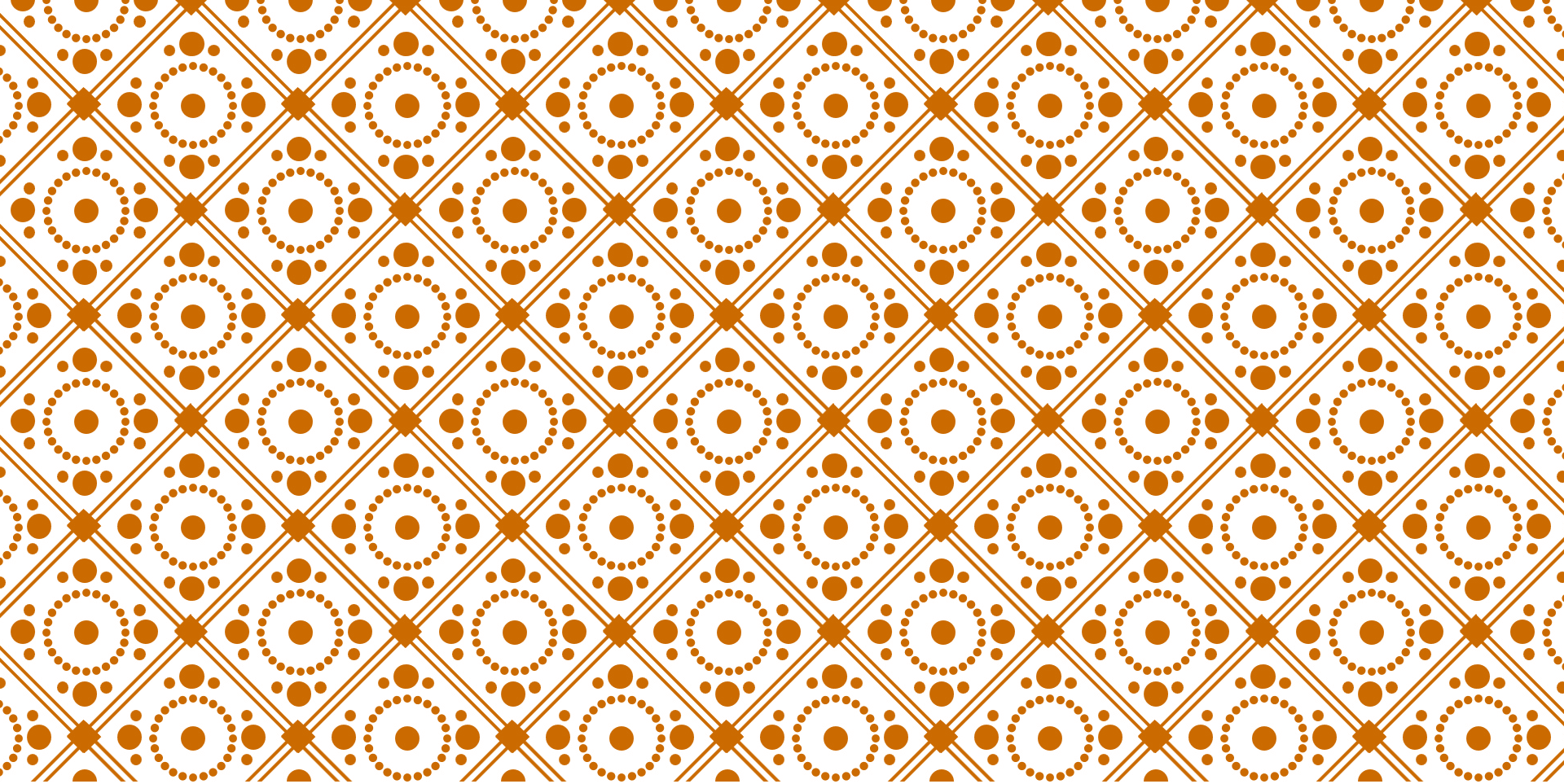
cd 3-selectionSort/ sbatch exec.batch cat XX.out

```
void selection_sort(int *v, int n){
  int i, j, min, tmp;

  for(i = 0; i < n - 1; i++){
    min = i;

    for(j = i + 1; j < n; j++)
      if(v[j] < v[min])
        min = j;

    tmp = v[i];
    v[i] = v[min];
    v[min] = tmp;
  }
}
```

SOLUÇÃO 3.1: SELECTION SORT

cd 3-selectionSort/ sbatch exec.batch cat XX.out

```
 for(i = 0; i < n - 1; i++){
  #pragma omp parallel private(j, min_local)
  { min_local = i;
    #pragma omp single  min = i
    #pragma omp for
    for(j = i + 1; j < n; j++) if(v[j] < v[min_local])
min_local = j;
   #pragma omp critical
   if(v[min_local] < v[min]) min = min_local;
  }

    tmp = v[i];
    v[i] = v[the_min];
    v[the_min] = tmp;
  }
```



INTRODUÇÃO A PROGRAMAÇÃO VETORIAL

SINGLE INTRUCTION MULTIPLE DATA (SIMD)

Técnica aplicada por unidade de execução

- Opera em mais de um elemento por iteração.
- Reduz número de instruções significativamente.

Elementos são armazenados em registradores SIMD

Scalar

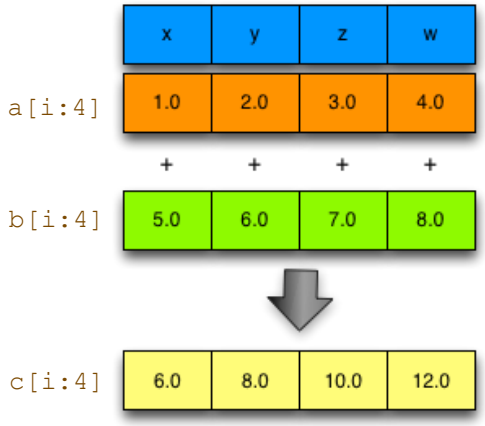
Uma instrução. Uma operação.

```
for(i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Vector

Uma instrução. Oito operações, por exemplo.

```
for(i = 0; i < N; i += 8)
  c[i:8] = a[i:8] + b[i:8];
```



SINGLE INTRUCTION MULTIPLE DATA (SIMD)

Técnica aplicada por unidade de execução

- Opera em mais de um elemento por iteração.
- Reduz número de instruções significativamente.

Elementos são armazenados em registradores SIMD

Scalar

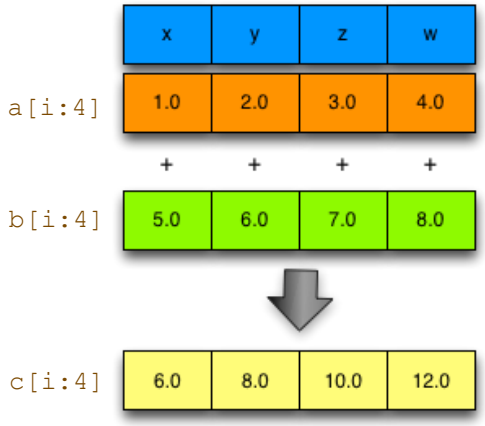
Uma instrução. Uma operação.

```
for(i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Vector

Uma instrução. Oito operações, por exemplo.

```
for(i = 0; i < N; i += 4)
  c[i:4] = a[i:4] + b[i:4];
```



Dados contíguos para desempenho ótimo
c[0] c[1] c[2] c[3] …

PROGRAMAÇÃO VETORIAL

Vetorização

```
#pragma omp simd
 for(i = 0; i < N; i++)
   c[i] = a[i] + b[i];
```

Vetorização com redução

```
#pragma omp simd reduction(+ : v)
 for(i = 0; i < N; i++)
   v += a[i] + b[i];
```

EXERCÍCIO 4, PARTE A: DOT PRODUCT SIMD

cd 4-dotProduct/ sbatch exec.batch cat XX.out

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;



  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

SOLUÇÃO 4.1, PARTE A: DOT PRODUCT SIMD

cd 4-dotProduct/ sbatch exec.batch cat XX.out

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;


  #pragma omp simd reduction(+ : dot)
  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

EXERCÍCIO 4, PARTE B: DOT PRODUCT PARALLEL SIMD

cd 4-dotProduct/ sbatch exec.batch cat XX.out

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;



  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

SOLUÇÃO 4.2, PARTE B: DOT PRODUCT PARALLEL SIMD

cd 4-dotProduct/ sbatch exec.batch cat XX.out

```
double dotproduct(double *a, int *b, long long int N){
  long long int i;
  double dot = 0.0;


  #pragma omp parallel for simd reduction(+ : dot)
  for(i = 0; i < N; i++)
    dot += a[i] * b[i];

  return dot;

}
```

EXERCÍCIO 5, PARTE A: MM - SIMD

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++)
   for(j = 0; j < N; j++)
     for(k = 0; k < N; k++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];




}
```

SOLUÇÃO 5.1, PARTE A: MM – SIMD

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++)
   for(j = 0; j < N; j++)
     #pragma omp simd
     for(k = 0; k < N; k++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];



}
```

~~SOLUÇÃO 5.1~~, PARTE A: MM – SIMD WRONG

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++)
   for(j = 0; j < N; j++)
     #pragma omp simd
     for(k = 0; k < N; k++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];



}
```

EXERCÍCIO 5, PARTE B: MM - SIMD

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++)
   for(j = 0; j < N; j++)
     #pragma omp simd
     for(k = 0; k < N; k++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];



}
```

SOLUÇÃO 5.2, PARTE B: MM - SIMD

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++)
   for(k = 0; k < N; k++)
     #pragma omp simd
     for(j = 0; j < N; j++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];



}
```

EXERCÍCIO 5, PARTE C: MM – PARALLEL SIMD

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 for(i = 0; i < N; i++)
   for(k = 0; k < N; k++)
     #pragma omp simd
     for(j = 0; j < N; j++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];



}
```

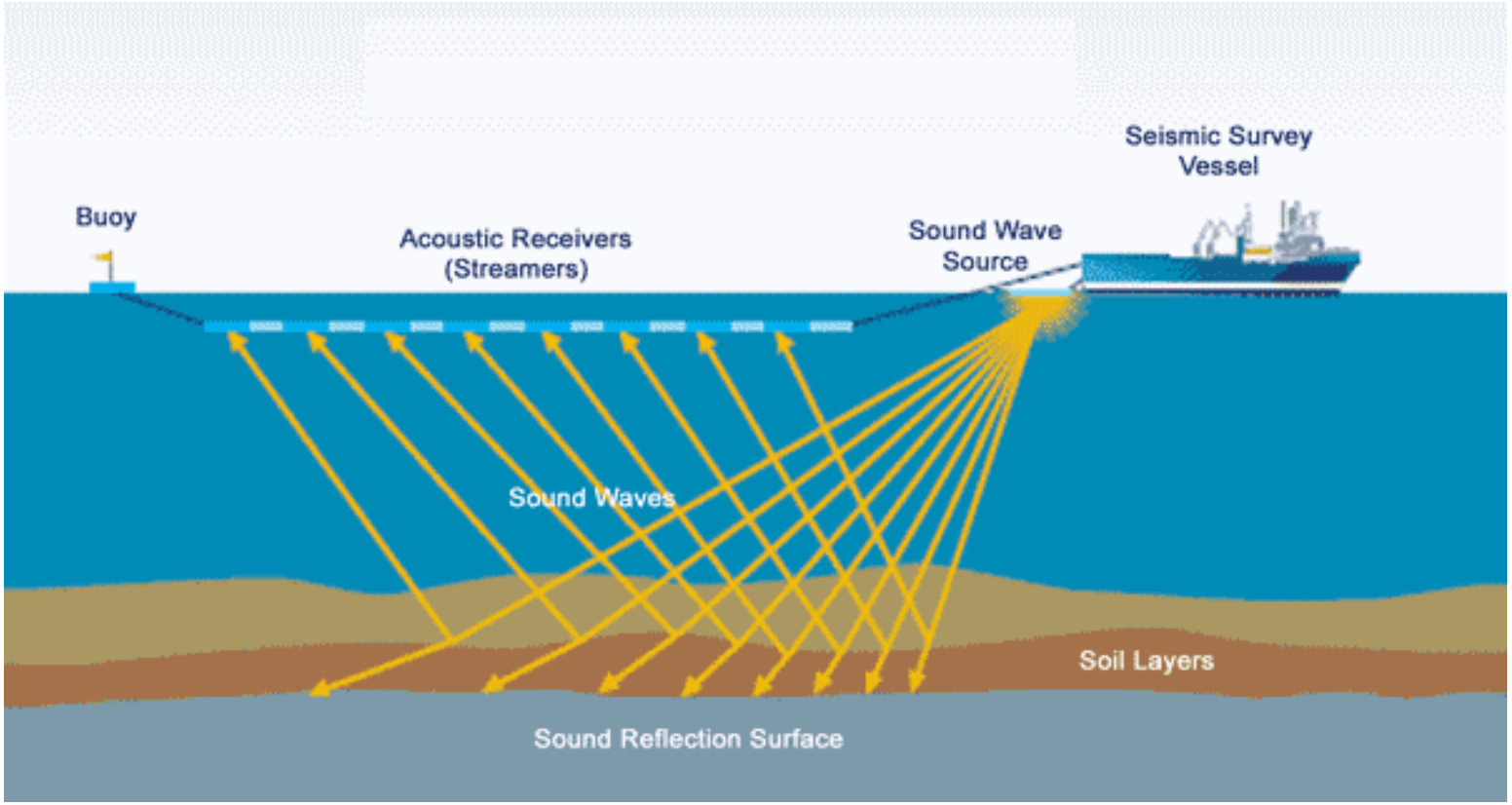
SOLUÇÃO 5.3, PARTE C: MM – PARALLEL SIMD

cd 5-matrixMultiplication/ sbatch exec.batch cat XX.out

```
void matrix_mult(double *A, *B, *C, int N){
 int i, j, k;


 #pragma omp parallel for private(i, j, k)
 for(i = 0; i < N; i++)
   for(k = 0; k < N; k++)
     #pragma omp simd
     for(j = 0; j < N; j++)
       C[i * N + j] += A[i * N + k] * B[k * N + j];


}
```

EXERCÍCIO 6: APLICAÇÃO PETRÓLEO

cd 6-petroleo/ sbatch exec.batch cat XX.out



EXERCÍCIO 6: APLICAÇÃO PETRÓLEO

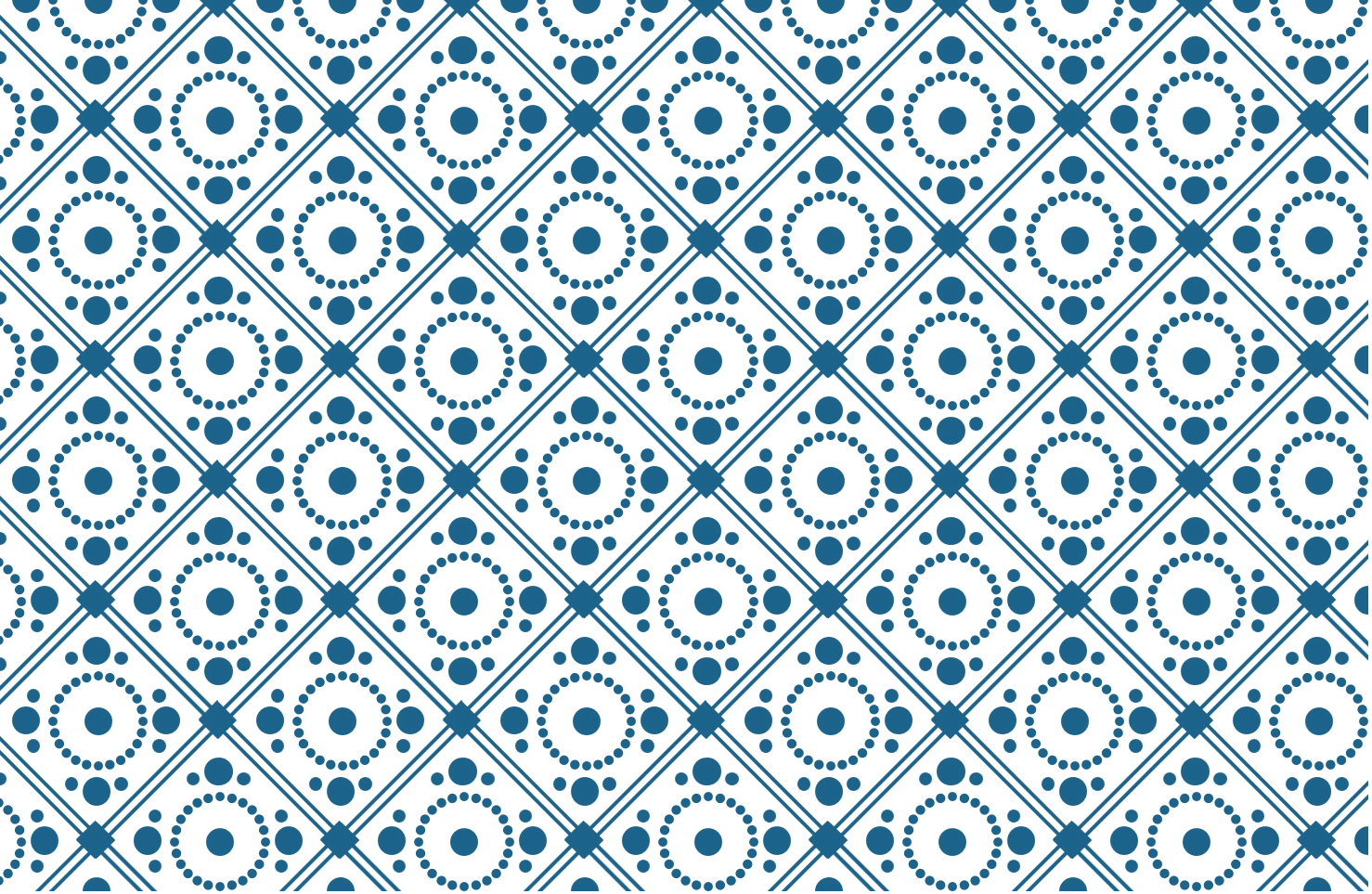
cd 6-petroleo/ sbatch exec.batch cat XX.out

```
void kernel_CPU_06_mod_3DRhoCte(...){
  ...

  for(index_X = 0; index_X < nnoi; index_X++)
    for(index_Y = 0; index_Y < nnoj; index_Y++)
      for(k = 0; k < k1 - k0; k++){

        ...
  }

  ...

}
```

SOLUÇÃO 6: APLICAÇÃO PETRÓLEO

cd 6-petroleo/ sbatch exec.batch cat XX.out

```
void kernel_CPU_06_mod_3DRhoCte(...){
  ...

  #pragma omp parallel for private(index_X, index_Y, index, k)
  for(index_Y = 0; index_Y < nnoj; index_Y++)
    for(k = 0; k < k1 - k0; k++)
      #pragma omp simd
      for(index_X = 0; index_X < nnoi; index_X++){

      ...
  }

  ...

}
```



GPPD

Grupo de Processamento Paralelo e Distribuído

UFRGS

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

LNCC

PERGUNTAS??

Escola Supercomputador

Matheus S. Serpa
msserpa@inf.ufrgs.br

MC-SD02-III