ERAD RS 2022

Acelerando kernels de Machine Learning em processadores multicore e vetorial

Matheus S. Serpa, Félix D. P. Michels Júnior, Philippe O. A. Navaux
msserpa@inf.ufrgs.br, felix.junior@inf.ufrgs.br, navaux@inf.ufrgs.br

Minicurso 6

APRESENTAÇÃO

Matheus S. Serpa
linkedin.com/in/matheusserpa/

Atividades:
- Lead Data Scientist @DigiFarmz Smart Agriculture **[We're hiring]**
- Data Science Instructor @TargetTrust

Formação:
- Bacharelado em Ciência da Computação (UNIPAMPA 2015)
- Mestrado em Computação (UFRGS 2018)

  Período sanduíche na Université de Neuchâtel - Suíça
- Doutorado em andamento em Computação (UFRGS)

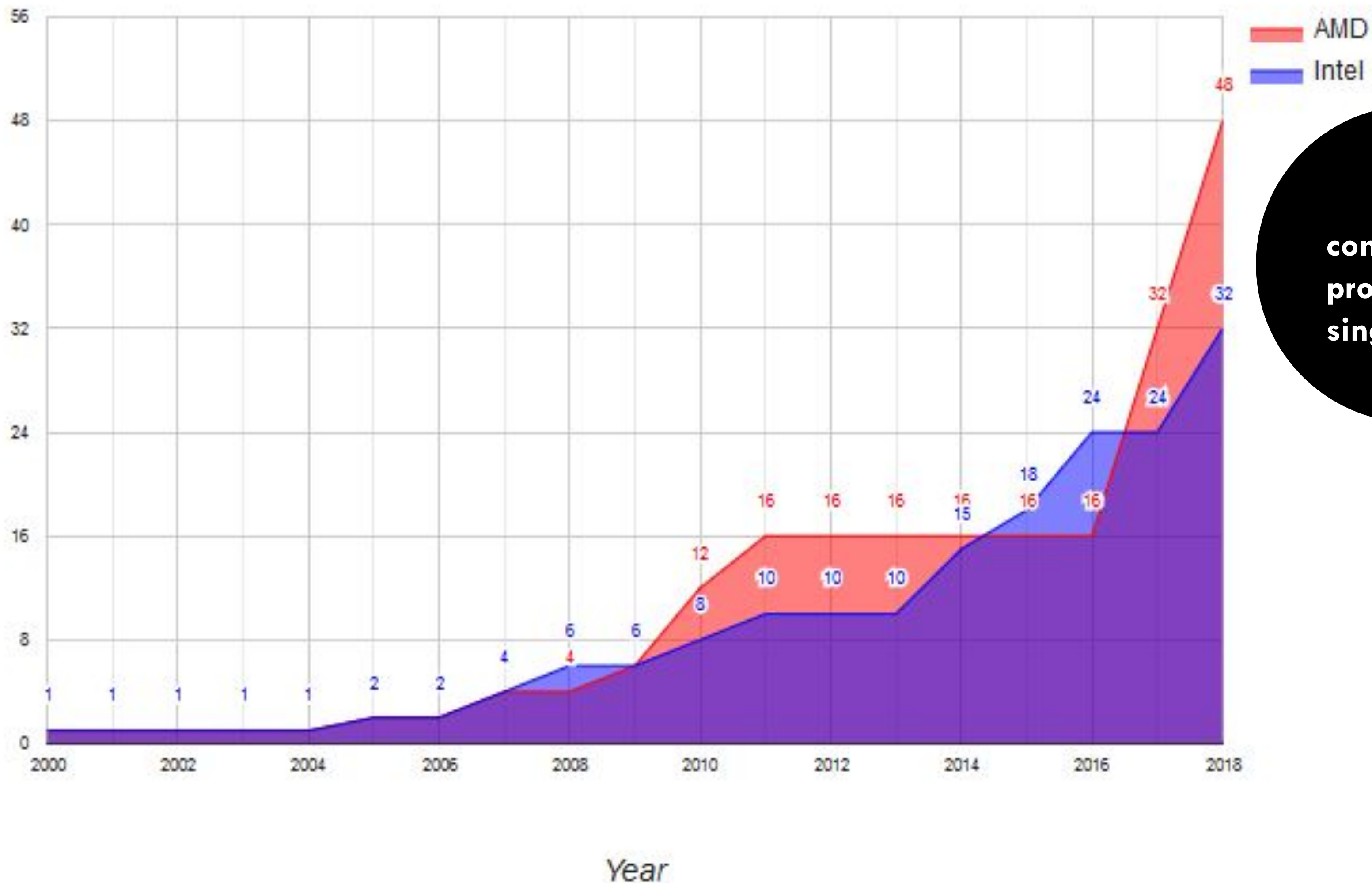
POR QUE ESTUDAR PROGRAMAÇÃO PARALELA?

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?

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.

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.

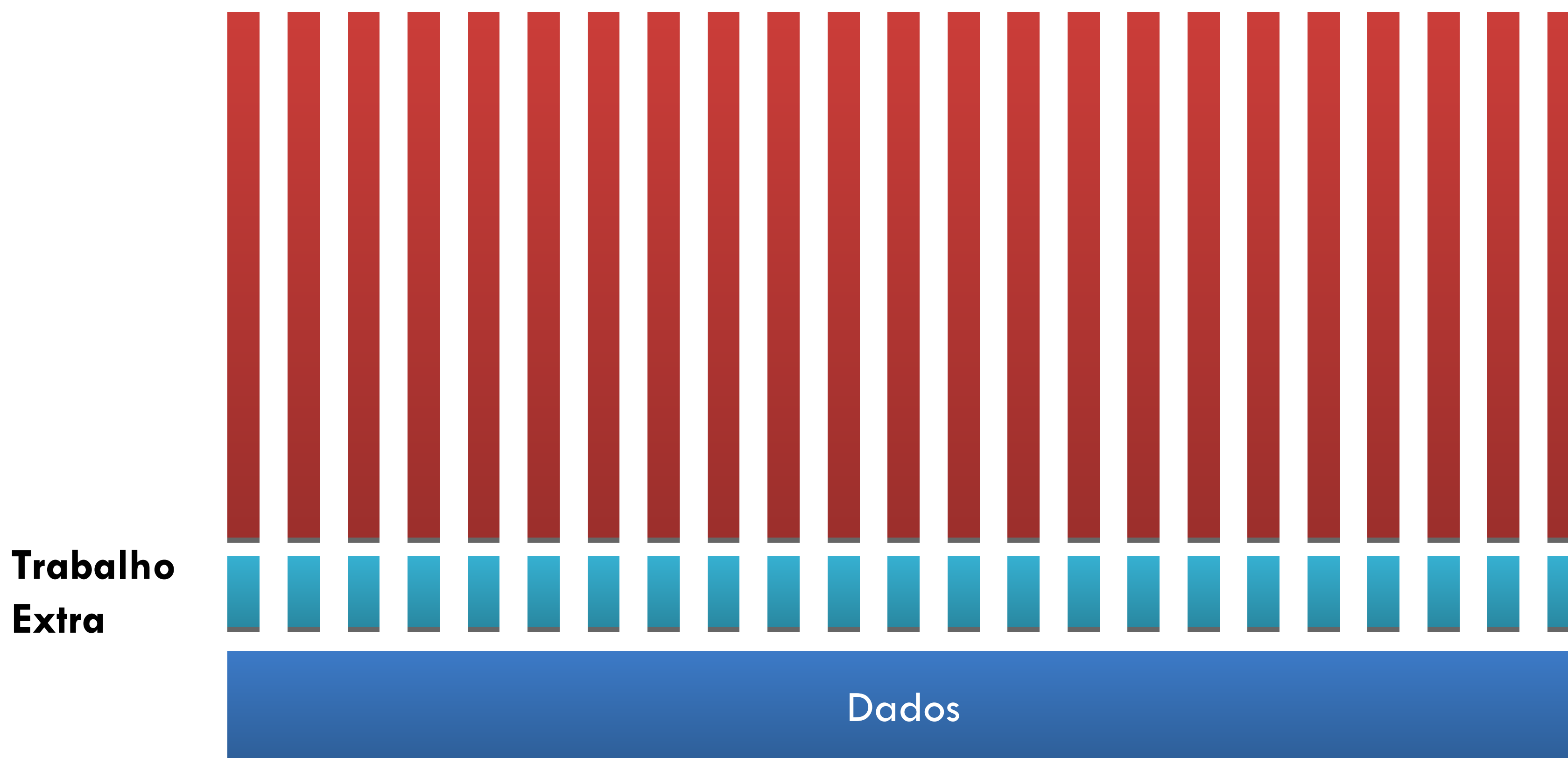
Comunicação: existe sempre um custo associado à troca de informação e enquanto as tarefas processam essa informação não contribuem para a computação.

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.

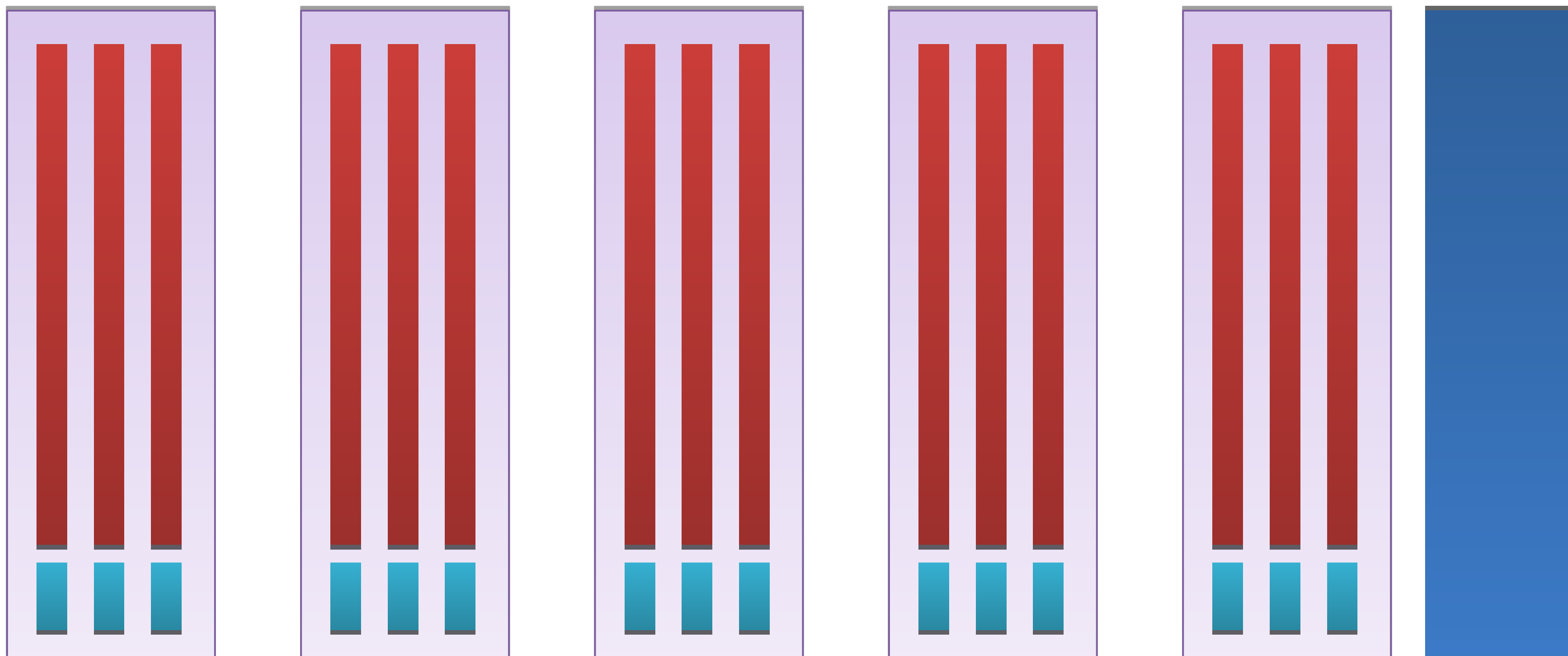
Granularidade: o número e o tamanho das tarefas é importante porque o tempo que demoram a ser executadas tem de compensar os custos da execução em paralelo (e.g. custos de criação, comunicação e sincronização).

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!



Divisão e Organização lógica do nosso algoritmo paralelo

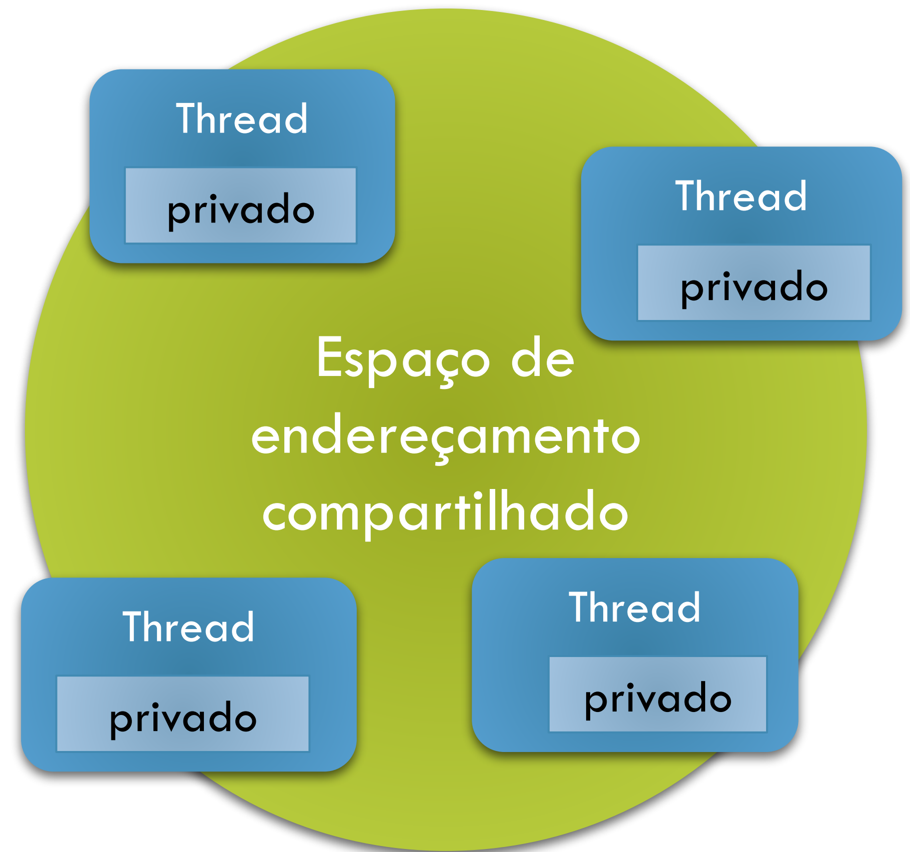
UM PROGRAMA DE MEMÓRIA COMPARTILHADA

Uma instância do programa:

Um processo e muitas threads.

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

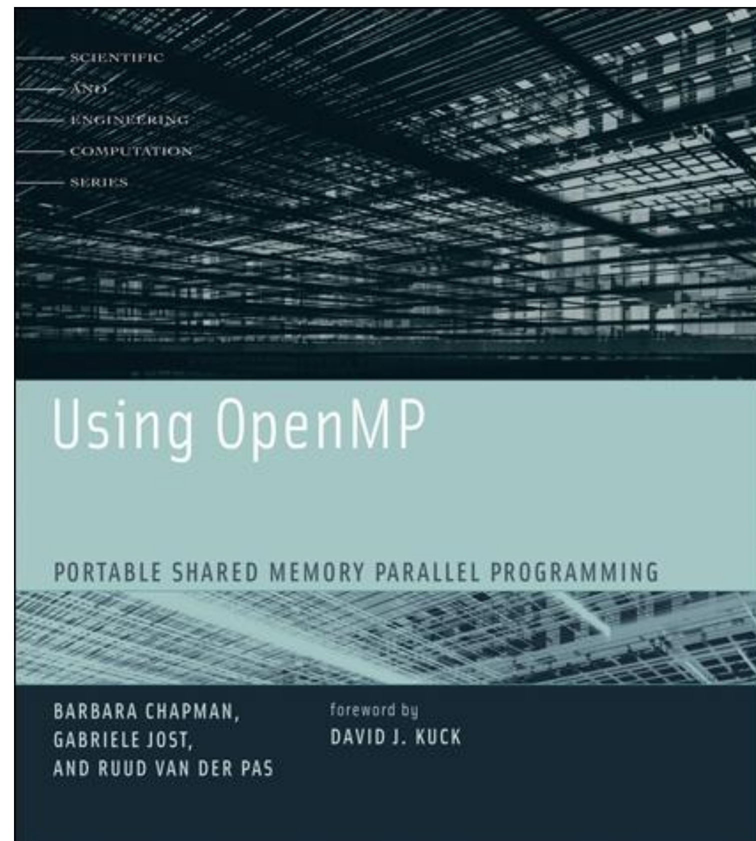
Sincronização garante a ordem correta dos resultados.



BIBLIOGRAFIA BÁSICA

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

Ano: 2007



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** or **intel icc**:

```
gcc -fopenmp foo.c #GCC
icc -qopenmp foo.c #Intel ICC
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.
icc -o hello hello.c -qopenmp
```

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

```
#include <stdio.h>

int main(){
  int myid, nthreads;



  myid = 0;

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

  return 0;
}
```

0 of 1 – hello world!

SOLUÇÃO 1.1: HELLO WORLD

Variáveis privadas.

```
#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.

```
#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.

```
#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

```
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.
icc -o hello hello.c -qopenmp
```

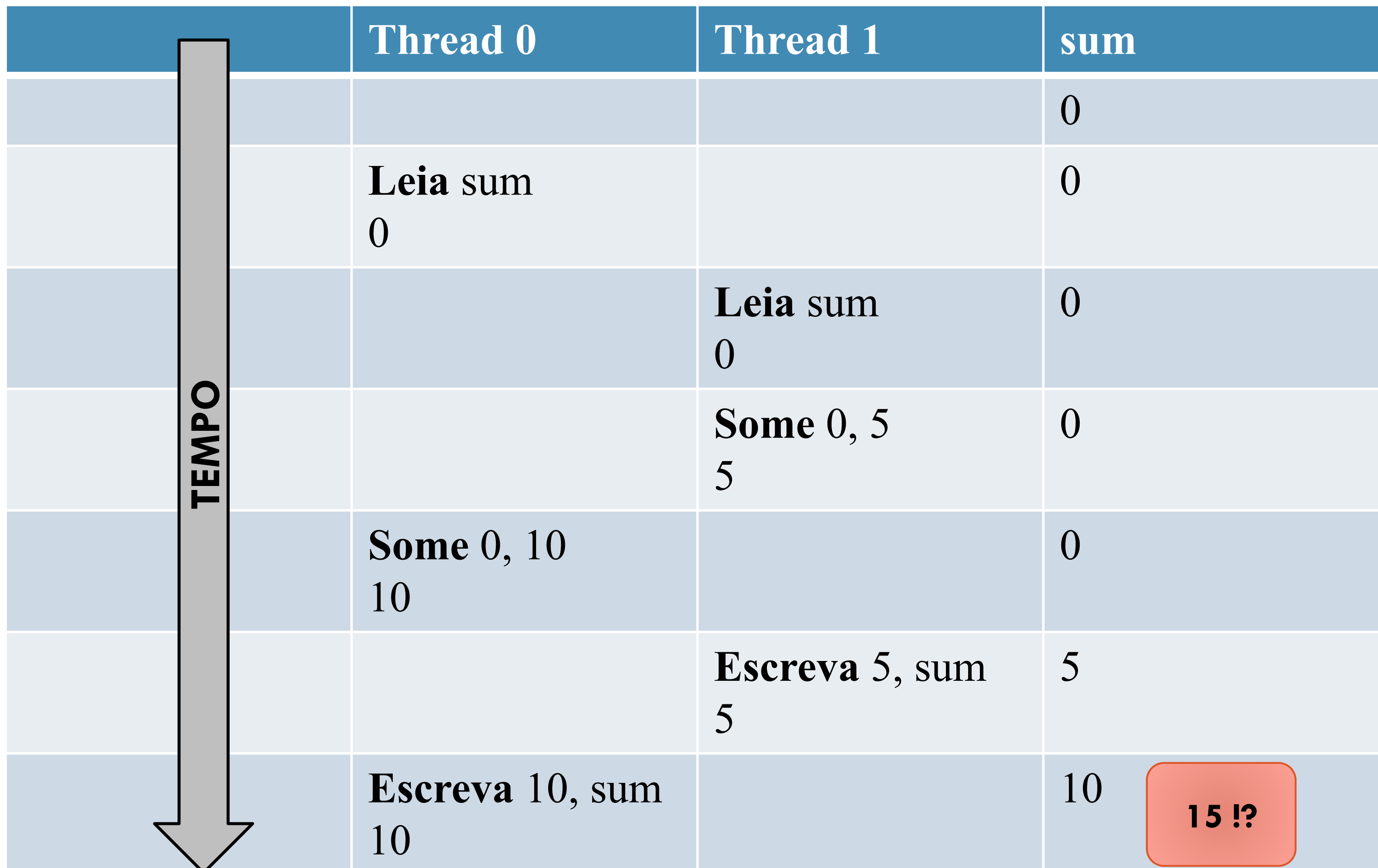
DIRETIVAS

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
```

COMO AS THREADS INTERAGEM?

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

CONDIÇÕES DE CORRIDA: EXEMPLO

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

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

15 !?

SINCRONIZAÇÃO

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

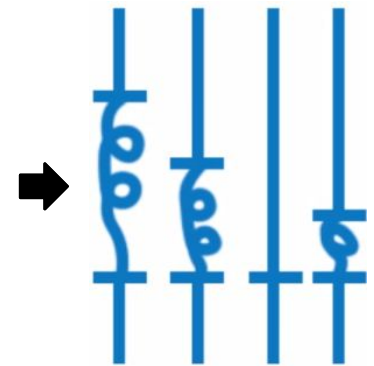
As duas formas mais comuns de sincronização são:

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



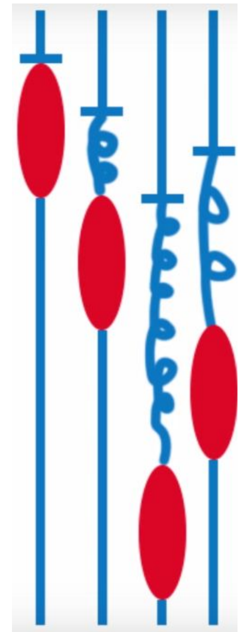
SINCRONIZAÇÃO

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

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

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

  return sum

}
```

EXERCÍCIO 2, PARTE D: VECTOR SUM

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

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

  return sum

}
```

EXERCÍCIO 2, PARTE E: VECTOR SUM

```
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

```
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

EXERCÍCIO 3: SELECTION SORT

```
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;
  }
}
```

ERAD | RS 2022

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

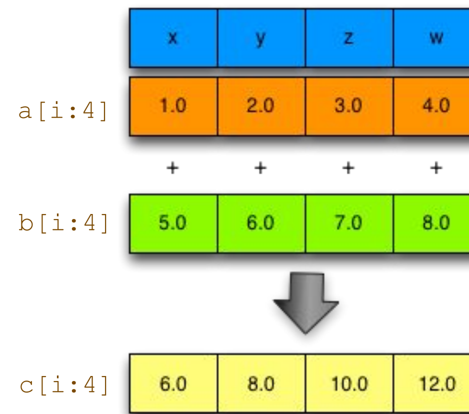
Uma instrução. Uma operação.

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

Vector

Uma instrução. Quatro operações, por exemplo.

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



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

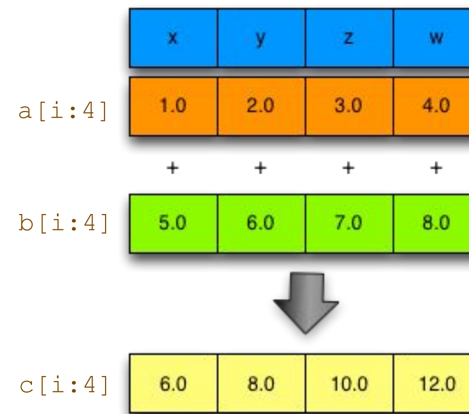
Uma instrução. Uma operação.

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

Vector

Uma instrução. Quatro 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 vector aligned
#pragma omp simd
for(i = 0; i < N; i++)
  c[i] = a[i] + b[i];
```

Vetorização com redução

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

EXERCÍCIO 4, PARTE A: MM - PARALLEL

```
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];




}
```

EXERCÍCIO 4, PARTE B: MM - SIMD

```
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];




}
```

EXERCÍCIO 4, PARTE C: MM - SIMD

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




}
```

EXERCÍCIO 4, PARTE D: MM – PARALLEL SIMD

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




}
```

Otimização de Kernels de ML

EXERCÍCIO 5, PARTE A: BACKPROPAGATION

```c
BPNN *bpnn_internal_create(n_in, n_hidden, n_out)
int n_in, n_hidden, n_out;
{
  ...

  newnet->input_units = alloc_1d_dbl(n_in + 1);
  newnet->hidden_units = alloc_1d_dbl(n_hidden + 1);
  newnet->output_units = alloc_1d_dbl(n_out + 1);

  newnet->hidden_delta = alloc_1d_dbl(n_hidden + 1);
  newnet->output_delta = alloc_1d_dbl(n_out + 1);
  newnet->target = alloc_1d_dbl(n_out + 1);

  newnet->input_weights = alloc_2d_dbl(n_in + 1, n_hidden + 1);
  newnet->hidden_weights = alloc_2d_dbl(n_hidden + 1, n_out + 1);

  newnet->input_prev_weights = alloc_2d_dbl(n_in + 1, n_hidden + 1);
  newnet->hidden_prev_weights = alloc_2d_dbl(n_hidden + 1, n_out + 1);

  return (newnet);
}
```

EXERCÍCIO 5, PARTE B: BACKPROPAGATION

```
BPNN *bpnn_read(filename){
  ...
  memcnt = 0;
  mem = (char *) malloc ((unsigned) ((n1+1) * (n2+1) * sizeof(float)));
  read(fd, mem, (n1+1) * (n2+1) * sizeof(float));
  for (i = 0; i <= n1; i++) {
    for (j = 0; j <= n2; j++) {
      fastcopy(&(new->input_weights[i][j]), &mem[memcnt], sizeof(float));
      memcnt += sizeof(float);
    }
  }
  free(mem);
  ...
}
```

EXERCÍCIO 5, PARTE C: BACKPROPAGATION

```
void bpnn_layerforward(l1, l2, conn, n1, n2){
  ...
  /*** For each unit in second layer ***/

  for (j = 1; j <= n2; j++) {
    /*** Compute weighted sum of its inputs ***/
    sum = 0.0;
    for (k = 0; k <= n1; k++) {
      sum += conn[k][j] * l1[k];
    }
    l2[j] = squash(sum);
  }
}
```

EXERCÍCIO 6: KMEANS

```
float** kmeans_clustering(...){
  ...
  do {


    for (i=0; i<npoints; i++) {
      /* find the index of nestest cluster centers */
      index = find_nearest_point(feature[i], nfeatures, clusters, nclusters);
      /* if membership changes, increase delta by 1 */
      if (membership[i] != index)
        delta += 1.0;
      /* assign the membership to object i */
      membership[i] = index;
      /* update new cluster centers : sum of all objects located within */
      partial_new_centers_len[index]++;
      for (j=0; j<nfeatures; j++)
        partial_new_centers[index][j] += feature[i][j];
    }
  } while (delta > threshold && loop++ < 500);
  ...
}
```

ERAD RS 2022

Acelerando kernels de Machine Learning em processadores multicore e vetorial

Minicurso 6

Matheus S. Serpa, Félix D. P. Michels Júnior, Philippe O. A. Navaux
msserpa@inf.ufrgs.br, felix.junior@inf.ufrgs.br, navaux@inf.ufrgs.br