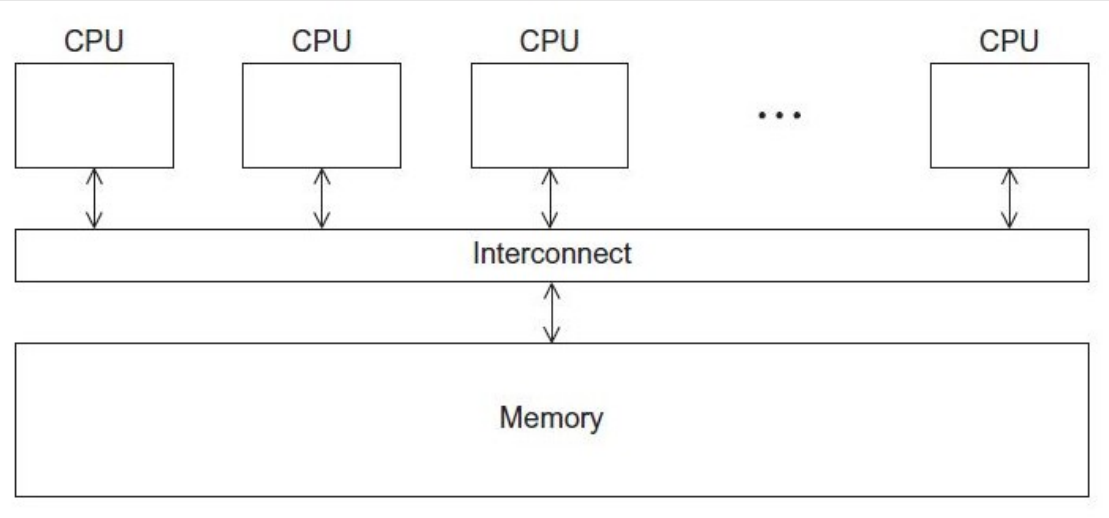
INF 310 – Programação Concorrente e Distribuída

OpenMP

Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

Introdução

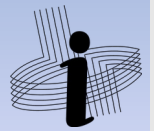
- OpenMP = *Open Multiprocessing*



Modelo de programação paralela de memória compartilhada

Introdução

- Pthreads vs OpenMP
  - Pthreads
    - Utilizada através de bibliotecas ligadas ao programa
    - Utilizada com qualquer compilador se o sistema implementar as bibliotecas
  - OpenMP
    - Utilizada através de diretivas
    - Requer suporte do compilador a tais diretivas
      - Caso não tenha suporte, tais diretivas serão simplesmente ignoradas

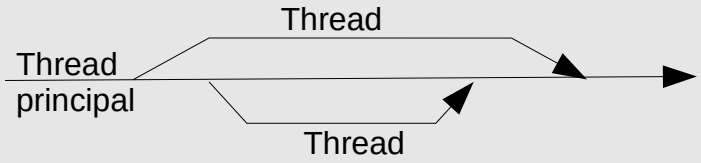


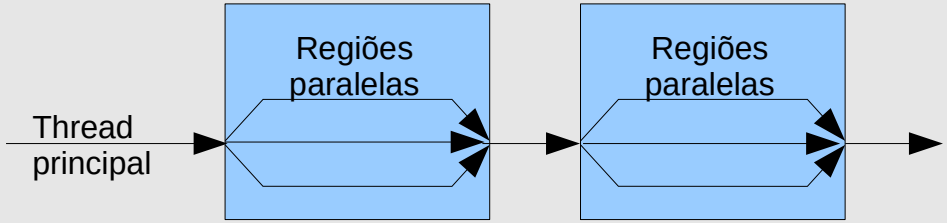
UFV

Introdução

- Pthreads vs OpenMP

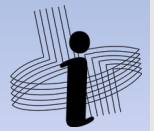
Pthreads	OpenMP
Threads podem ser independentes	Threads executam o mesmo código
Baixo nível (cada detalhe deve ser especificado para cada thread)	Alto nível (interações de baixo nível podem ser mais difíceis de programar)





Introdução

- Benefícios do OpenMP
  - Facilidade
  - Escalabilidade
  - Portabilidade
  - Permite paralelização de modo incremental



UFV

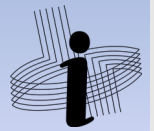
Introdução

- Biblioteca: *omp.h*
- Modelo baseado em diretivas de programação
- Em C e C++ é utilizada a instrução especial:

```
#pragma
```

- Todos as diretivas OpenMP iniciam com *pragma omp*
  - exemplo:

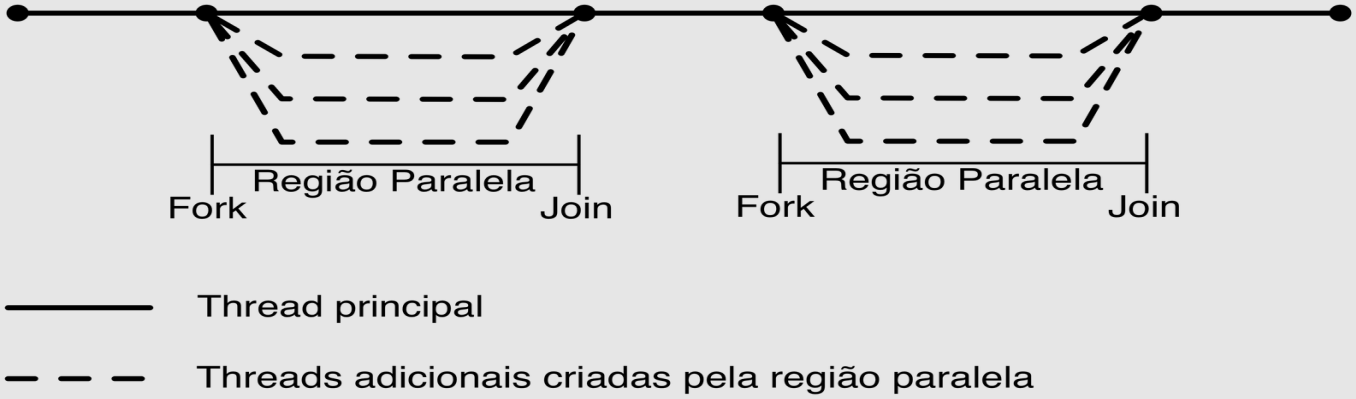
```
#pragma omp parallel
```



UFV

Introdução

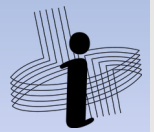
- Criação das threads e sincronização



Hello, World!

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

void Hello(void){
    int myself=omp_get_thread_num();
    int total=omp_get_num_threads();
    printf("Hello da thread %d de %d\n",myself,total);
}

int main(){

    #pragma omp parallel
    Hello();

    return 0;
}
```

Variáveis privadas de cada thread
(cada thread tem sua pilha)



UFV

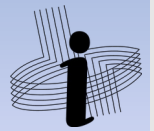
Hello, World!

- Compilando

```
$ g++ -fopenmp -o omp_hello omp_hello.c
```

- Executando

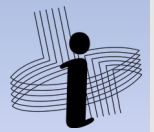
```
$ ./omp_hello
Hello da thread 0 de 4
Hello da thread 3 de 4
Hello da thread 1 de 4
Hello da thread 2 de 4
```



UFV

Hello, World!

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

void Hello(void){
    int myself=omp_get_thread_num();
    int total=omp_get_num_threads();
    printf("Hello from thread %d of %d\n",myself,total);
}

int main(int argc, char* argv[]){
    int numThreads=atoi(argv[1]);

    #pragma omp parallel num_threads(numThreads)
    Hello();

    return 0;
}
```

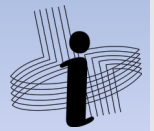


UFV

Hello, World!

- Executando

```
$ ./omp_hello 5
Hello from thread 0 of 5
Hello from thread 3 of 5
Hello from thread 1 of 5
Hello from thread 2 of 5
Hello from thread 4 of 5
```



UFV

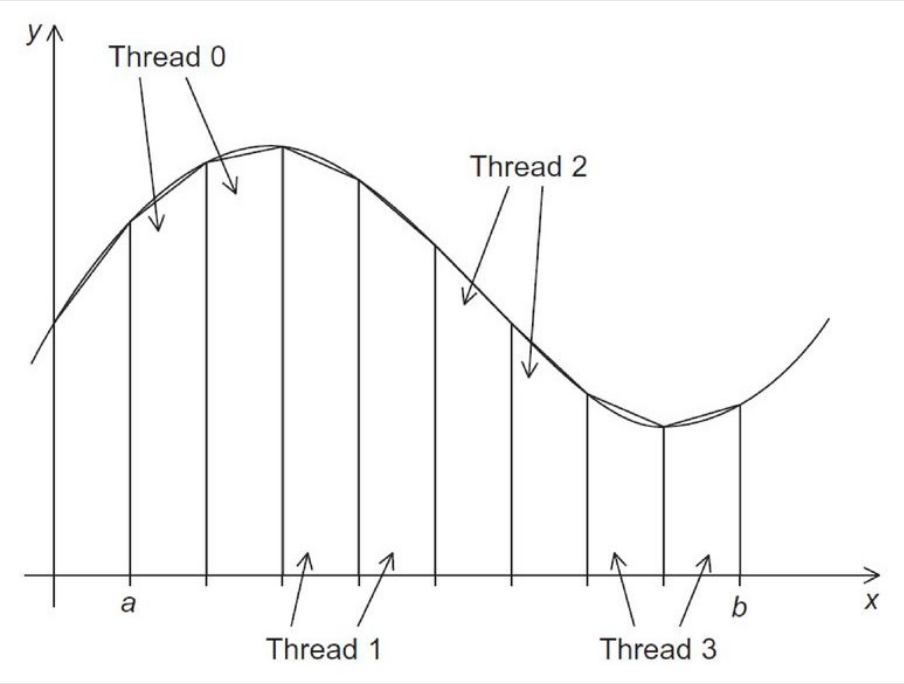
A regra do trapézio

- Cálculo da área total em 2 etapas:
  - Cálculo das áreas individuais
  - Soma da área total

```
totalArea += myArea
```

A soma da área total deve ser feita com exclusão mútua

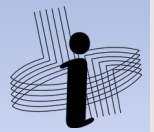
```
#pragma omp critical
totalArea += myArea
```



Atribuição de trapézios às threads

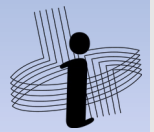
A regra do trapézio

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

double f(double x){
     return 4.0/(1+x*x); //função para valor de pi
}

void Trapezio(double a,double b,int n,double *totalArea);

int main(){
     double totalArea=0.0;
     double a=0.0;          //início
     double b=1.0;          //fim
     int n=1000;            //número de trapézios

     #pragma omp parallel
     Trapezio(a,b,n,&totalArea);

     printf("Area = %f", totalArea);
     return 0;
}
```



A regra do trapézio

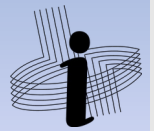
```
void Trapezio(double a,double b,int n,double *totalArea) {
    double base, x, myArea;
    int myself=omp_get_thread_num();
    int numThreads=omp_get_num_threads();

    base = (b-a)/n;                      //largura da base
    int ini=myself*n/numThreads;         //primeiro trapézio de cada thread
    int fim=(myself+1)*n/numThreads;     //último trapézio de cada thread
    double localA=a+ini*base;            //posição de início da thread
    double localB=a+fim*base;            //posição final da thread

    myArea = (f(localA)+f(localB))/2.0;
    for (int i=ini+1;i<fim;i++){
        x=a+i*base;
        myArea+=f(x);                    //soma de todas as alturas
    }
    myArea*=base;

    #pragma omp critical                 //uso de exclusão mútua para
    *totalArea+=myArea;                  //atualização da área total
}
```



UFV

Outras rotinas úteis

- omp_get_thread_num(): retorna o número identificador de uma thread
- omp_get_num_threads(): retorna o número de threads disparadas dentro de uma região paralela
- omp_set_num_threads(int): define o número de threads a serem utilizadas nas próximas regiões paralelas
- omp_get_num_procs(): retona o número de núcleos disponíveis na máquina

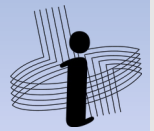


UFV

Escopo das variáveis

- Depende do momento da criação da variável
- O escopo pode ser definido explicitamente através do uso de cláusulas para a região *parallel*

	Global	**Privada**
Implícito	Variável criada antes da seção *parallel*	Criação dentro da seção *parallel*
Explícito	Cláusula *shared*	Cláusula *private*



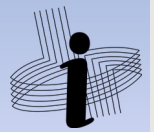
UFV

Escopo das variáveis

- Uso de clásulas
  - Comando:

```
#pragma omp parallel <cláusula>
```

  - Cláusulas:
    - private (var1, var2, ...)
    - shared (var1, var2, ...)
    - firstprivate (var1, var2, …)
    - default (none)
    - reduction (operador: var1, var2, …)



UFV

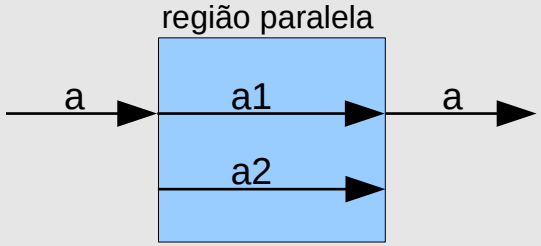
Escopo das variáveis

- private (var1, var2,...)
  - Indica quais variáveis serão privadas
  - Estas variáveis são inicializadas com valor 0 (zero)
    - Exemplo:

```
int a = 1;
#pragma omp parallel private(a) num_threads(2)
printf("paralelo = %d\n", a);
printf("sequencial = %d\n", a);
```

    - Saída:

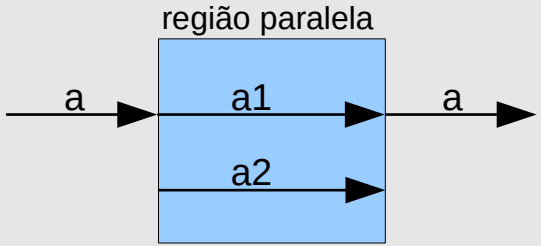
```
paralelo = 0
paralelo = 0
sequencial = 1
```



Escopo das variáveis

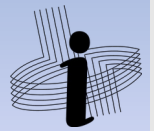
- firstprivate (var1, var2,...)
  - Indica quais variáveis serão privadas
  - Variáveis inicializadas com valor definido antes da região paralela
    - Exemplo:

```
int a = 1;
#pragma omp parallel firstprivate(a) num_threads(2)
{
    a += 1;
    printf("paralelo = %d\n", a);
}
printf("sequencial = %d\n", a);
```



    - Saída:

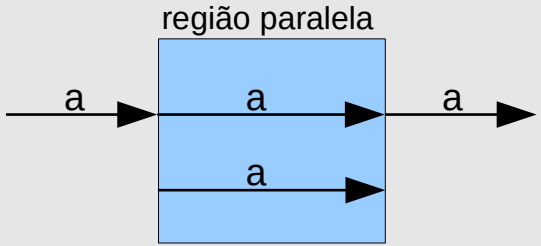
```
paralelo = 2
paralelo = 2
sequencial = 1
```



Escopo das variáveis

- shared (var1, var2,...)
  - Indica quais variáveis serão compartilhadas entre threads
  - Cada thread possui uma referência à variável original
    - Exemplo:

```
int a = 1;
#pragma omp parallel shared(a) num_threads(2)
{
    printf("paralelo = %d\n", a);
    a = 2;
}
printf("sequencial = %d\n", a);
```



    - Saída:

```
paralelo = 1
paralelo = 1
sequencial = 2
OU
paralelo = 1
paralelo = 2
sequencial = 2
```

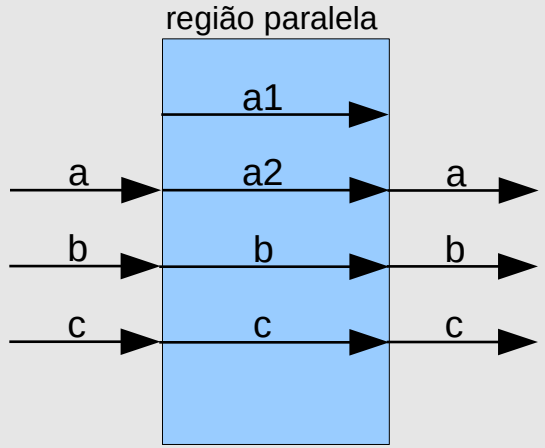
Escopo das variáveis

- default (none)
  - Suprime o escopo implícito de variáveis
  - Exceção para variáveis declaradas dentro da região paralela
    - Exemplo:

```
int a = 1, b=2, c=3;
#pragma omp parallel default(none) firstprivate(a) shared(b,c) \
        num_threads(2)
{
    a = 2;
    b = c;
    c = a;
}
printf("%d, %d, %d\n", a,b,c);
```

    - Saída:

```
1, 3, 2
OU
1, 2, 2
```

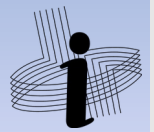


Redução

- Considere a seguinte versão da implementação da regra do trapézio

```
double LocalTrapezio(double a,double b,int n);
...
totalArea=0.0
#pragma omp parallel
{
    #pragma omp critical
    totalArea+=LocalTrapezio(a,b,n);
}
```

  - Qual o problema nessa versão?
    LocalTrapezio(...) seria chamada por cada thread com exclusão mútua



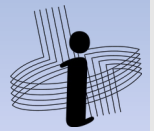
UFV

Redução

- Segunda tentativa

```
double LocalTrapezio(double a,double b,int n);
...
totalArea=0.0
#pragma omp parallel
{
    double myArea=0.0    //privado
    myArea+=LocalTrapezio(a,b,n);
    #pragma omp critical
    totalArea+=myArea
}
```

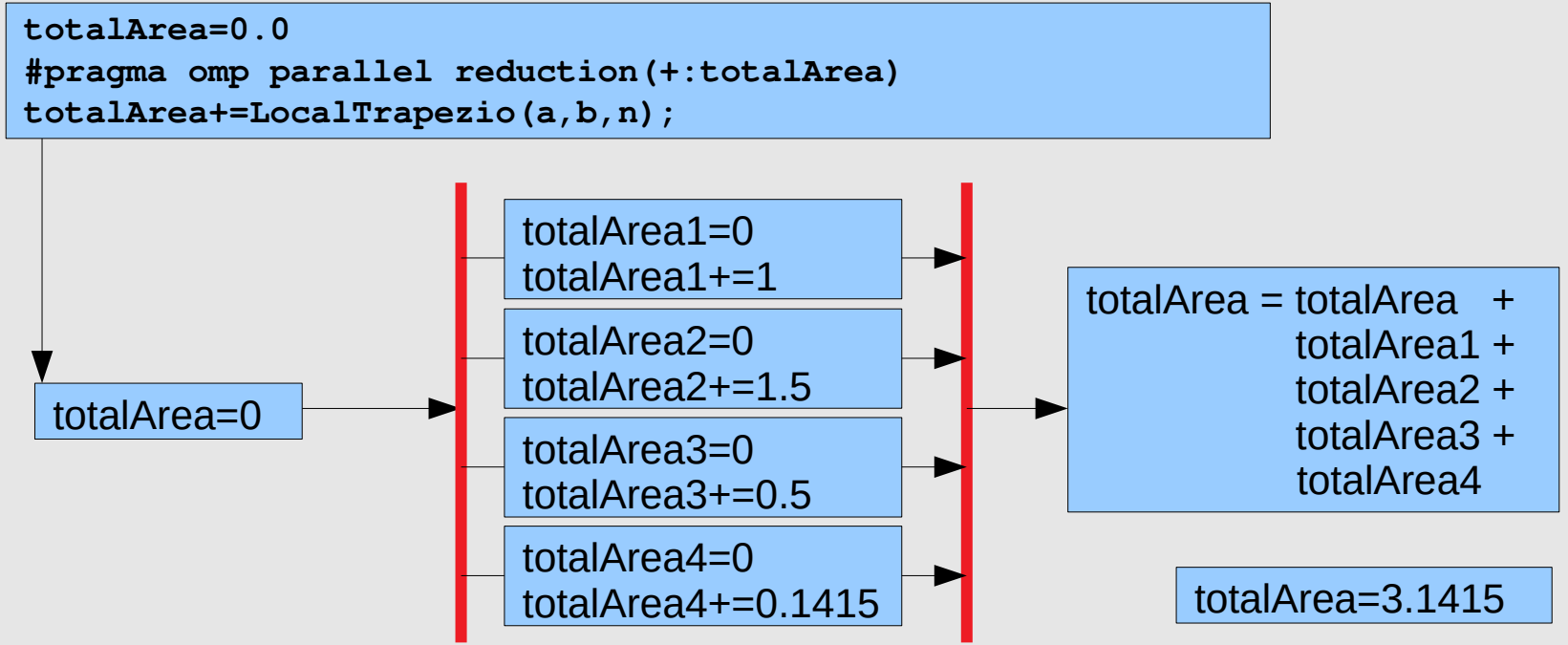
  - Uso de redução é uma alternativa mais limpa!



UFV

Redução

- Cláusula *reduction*
  - Aplica uma mesma operação binária repetidamente a uma sequência de operandos para obter um único resultado final



Redução

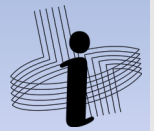
- Sintaxe

```
reduction(<operador>:<lista de variáveis>)
```

  - As variáveis privadas são iniciadas com o valor identidade (0 para soma, 1 para multiplicação...)
  - Operadores possíveis:

```
+   *   -   &   |   ^   &&   ||
```

    - Como a subtração não é comutativa nem associativa, a redução com subtração funciona internamente como uma soma



UFV

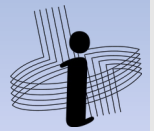
for paralelo

- OpenMP permite paralelização automática de laços do tipo *for*
  - código sequencial baseado na regra do trapézio

```
h=(b-a)/n;                  //base dos trapézios
approx=(f(a) + f(b))/2.0; //altura do primeiro trapézio
for (i=1; i<=n-1; i++)
    approx += f(a + i*h); //soma de todas as alturas
approx = h*approx;          //cálculo da área total
```

  - código paralelo com diretiva *parallel for*

```
h=(b-a)/n;
approx=(f(a) + f(b))/2.0;
#pragma omp parallel for reduction(+:approx)
for (i=1; i<=n-1; i++)   //parallel for é seguida de for
    approx += f(a + i*h);//i é sempre privada
approx = h*approx;
```

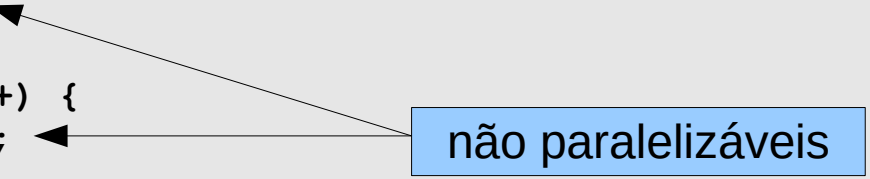


UFV

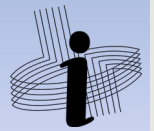
for paralelo

- Nem todos os *for* são paralelizáveis
  - Deve ser possível determinar número de iterações

```
for( ; ; ){ … } // for infinito


for(i=0; i<n; i++) {
    if (…) break;
    …
}
```



  - Exceção: é possível utilizar *exit* no corpo do *loop*

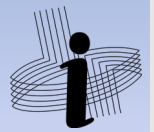


for paralelo

- Nem todos os *for* são paralelizáveis
  - O laço for deve seguir a sua forma canônica

```
                                    index++
                                    ++index
                    index < end     index--
                    index <= end    --index
for  index = start  ; index >= end  ; index += incr
                    index > end     index -= incr
                                    index = index + incr
                                    index = incr + index
                                    index = index - incr
```

  - *index* deve ser variável inteira ou ponteiro
  - *start*, *end* e *incr* devem ser do tipo inteiro
  - *index*, *start*, *end* e *incr* não podem ser alterados no loop

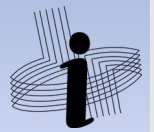


for paralelo

- Encontrando dependências entre loops

```
#pragma omp parallel for
for (i=0; i<n; i++){
    x[i]=a + i*h;
    y[i]=exp(x[i])
}
```

  - sem problema na paralização
    - as duas instruções são atribuídas à mesma thread



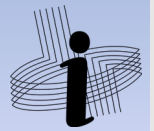
UFV

for paralelo

- Estimando o valor de Pi
  - Pi = 4*(1 - 1/3 + 1/5 – 1/7 + ...)

```
double factor = 1.0;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (k=0; k<n; k++) {
   sum += factor / (2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```

  - *factor* atualizado em uma iteração e utilizado em outra



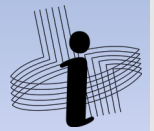
UFV

for paralelo

- Estimando o valor de Pi
  - Pi = 4*(1 - 1/3 + 1/5 – 1/7 + ...)

```
double factor = 1.0;
double sum = 0.0;
#pragma omp parallel for reduction(+:sum) private(factor)
for (k=0; k<n; k++) {
   factor = (k%2 == 0)? 1.0 : -1.0;
   sum += factor / (2*k+1);
}
pi_approx = 4.0*sum;
```

  - *factor* não pode ser compartilhada



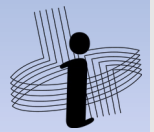
UFV

for paralelo

- *Bubble sort* (sequencial)

```
for (list_length=n; list_length>=2; list_length--)
    for (i=0; i<list_length-1; i++)
        if (a[i]>a[i+1]) {
            tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
```

  - Apresenta dependência entre *loops* tanto no *loop* interno quanto no externo

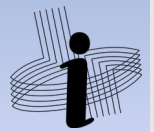


UFV

for paralelo

- *Odd-even sort*
  - Adaptação do *bubble sort* para permitir paralelismo
  - Elementos comparados dependem da fase (par ou ímpar)

Fase	0	1	2	3	4
-	9	6	8	7	5
0	6	9	7	8	5
1	6	7	9	5	8
2	6	7	5	9	8
3	6	5	7	8	9
4	5	6	7	8	9

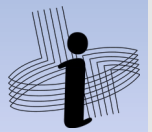


for paralelo

- *Odd-even sort*
  - Utilizando *parallel for*

```
int phase, i;

for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        #pragma omp parallel for default(none) private(i) shared(a,n)
        for (i=1; i<n; i+=2)
            if (a[i-1]>a[i]) Swap(&a[i-1], &a[i]);
    else
        #pragma omp parallel for default(none) private(i) shared(a,n)
        for (i=1; i<n-1; i+=2)
            if (a[i]>a[i+1]) Swap(&a[i], &a[i+1]);
```

As threads são criadas e destruídas a cada iteração



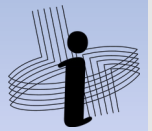
UFV

for paralelo

- *Odd-even sort*
  - Utilizando diretivas *for*

```
#pragma omp parallel default(none) private(i,phase) shared(a,n)
for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0)
        #pragma omp for
        for (i=1; i<n; i+=2)
            if (a[i-1]>a[i]) Swap(&a[i-1], &a[i]);
    else
        #pragma omp for
        for (i=1; i<n-1; i+=2)
            if (a[i]>a[i+1]) Swap(&a[i], &a[i+1]);
```

Não recria as threads a cada iteração

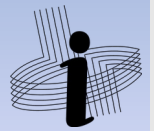


UFV

Checagem de erros

- Garantir portabilidade
- Funções exclusivas do OpenMP podem resultar em erros se o compilador não tiver suporte
- Utilizar a diretiva _OPENMP

```
#ifdef _OPENMP
    int myself = omp_get_thread_num();
    int nThreads = omp_get_num_threads();
#else
    int myself = 0;
    int nThreads = 1;
#endif
```



UFV

Diretiva *single*

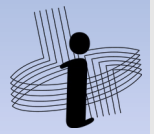
- Apenas uma das threads executa o bloco
  - Uma barreira é automaticamente utilizada no final do bloco *single*, mas não no início

```
#pragma omp parallel
{
    printf("Hello da thread %d\n", omp_get_thread_num());

    #pragma omp single
    printf("Single executado pela thread %d\n", omp_get_thread_num());
    /* barreira criada automaticamente */
}
```

  - Cláusula *nowait* pode ser utilizada para evitar que threads fiquem esperando

```
#pragma omp single nowait
func();
```

- Cláusula *master* tem função parecida à cláusula *single*
  - Apenas thread mestre executa o bloco
  - Não existe barreira ao final do bloco



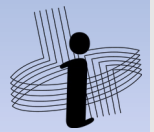
Diretiva *sections*

- Permite que threads executem códigos distintos
  - Cada *section* é executada por apenas 1 thread
  - Demais threads ficam paradas no final do bloco *sections*

```
int x,y;
#pragma omp parallel
   #pragma omp sections lastprivate(x,y)
   {
      #pragma omp section
      funcA();
      #pragma omp section
      x=funcB();
      #pragma omp section
      y=funcC();
   }
```

  - Cláusula *lastprivate*
    - Valor das variáveis estarão disponíveis após o bloco *sections*
  - Construção combinada também é possível

```
#pragma omp parallel sections
```

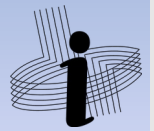


Cláusula *schedule*

- Permite controlar o modo de escalonamento das threads
- Utilizada apenas em loops
  - Supondo que o tempo de execução de f é proporcional ao valor de i

```
float soma=0;
for (int i=0; i<n; ++i)
    soma+=f(i);
```

  - Dependendo do escalonamento utilizado as primeiras threads terminam antes e precisam esperar as últimas



UFV

Cláusula *schedule*

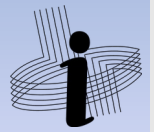
- Uso

```
#pragma omp parallel for schedule(tipo,chunk_size)
```

  - Cada *chunk* é um subconjunto contínuo e não-vazio do espaço de iteração
    - *chunk_size* é opcional e não precisa ser dado por um valor constante
  - Tipos
    - *static*: Iterações divididas em *chunks* e atribuídas estaticamente às threads no modo *round-robin*. Utilizado como padrão em MUITOS compiladores
      - tamanho padrão do *chunk* = *num_iteracoes / num_threads*
    - *dynamic*: *Chunks* são atribuídos sob demanda
      - tamanho padrão do *chunk* = 1
    - *guided*: Igual *dynamic* mas tamanho dos chunks é definido dinamicamente como *iteracoes_restantes / num_threads*. Parâmetro *chunk_size* define o tamanho mínimo
      - valor padrão do parâmetro *chunk_size* = 1
    - *runtime*: Escalonamento defindo através da variável de ambiente OMP_SCHEDULE

```
$ export OMP_SCHEDULE="static,1"
```

  - Overhead na distribuição da threads em cada tipo : *static* < *dynamic* < *guided*



Diretiva *atomic*

- Permite atualização de variáveis compartilhadas de forma eficiente sem condição de corrida
  - Forma:

```
#pragma omp atomic
<variável> <op>= <expressão>
```

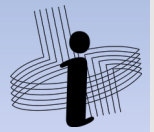
    - Operações suportadas:

```
+   *   -   /   &   |   ^   <<   >>
```

    - A expressão utilizada não pode referenciar a variável atualizada
    - Também aceita operadores ++ e --
  - Exemplo:

```
#pragma omp parallel
    #pragma omp atomic
    x += func();    //func() não é executada com exclusão mútua
    #pragma omp atomic
    x++;
    #pragma omp atomic
    x+=y++;         //condição de corrida na atualização de y
```

  - Cuidado: Utilizar *critical* e *atomic* ao mesmo tempo não garante exclusão mútua



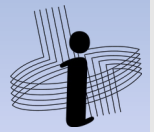
Usando travas

- Seções críticas independentes podem ser criadas através de um identificador

```
#pragma omp critical(nome)
```

  - Identificadores são atribuídos durante compilação
- Mas como definir acesso com exclusão mútua em tempo de execução?
  - ex: Acesso exclusivo a cada nó de uma lista encadeada
- Biblioteca do OpenMP implementa trava simples e aninhada

```
omp_lock_t                                  //tipo simple lock
void omp_init_lock(omp_lock_t *l);          //inicializa
void omp_set_lock(omp_lock_t *l);           //obtem a trava
void omp_unset_lock(omp_lock_t *l);         //libera a trava
void omp_destroy_lock(omp_lock_t *l);       //destrói

omp_nest_lock_t                             //tipo nested lock (aninhada)
void omp_init_nest_lock(omp_nest_lock_t *l);
void omp_set_nest_lock(omp_nest_lock_t *l);
void omp_unset_nest_lock(omp_nest_lock_t *l);
void omp_destroy_nest_lock(omp_nest_lock_t *l);
```

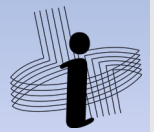


UFV

Diretiva *task*

- Permite paralelizar problemas irregulares
  - Laços sem limites conhecidos
  - Algoritmos recursivos
- Criação de *pool* de threads que executam as tarefas independentes
- Cada *task* executada de forma assíncrona por uma thread disponível
- Sincronização realizada ao final do bloco *parallel*

```
#pragma omp parallel
    #pragma omp single
    {
        node *n = lista->inicio;
        while (n){
            #pragma omp task private(n)
            calcula(n);
            n = n->próximo;
        }
    }
```



UFV