Programação Paralela com OpenMP

ELC139 - Programação Paralela

João Vicente Ferreira Lima (UFSM)

Universidade Federal de Santa Maria
jvlima@inf.ufsm.br
http://www.inf.ufsm.br/~jvlima

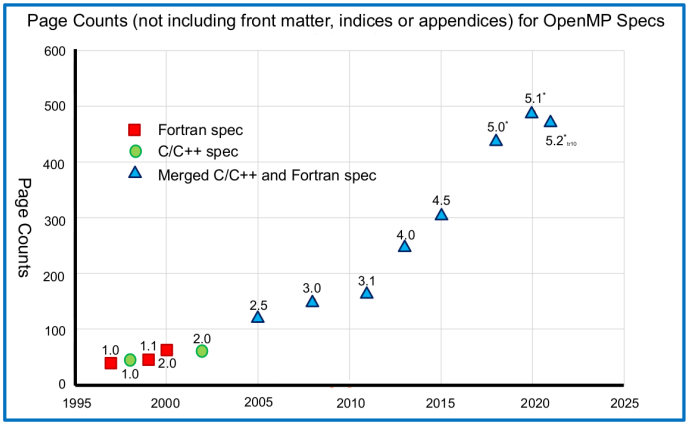
2023/1

Outline

Outline

Programação em OpenMP

- Epecificação de uma API para paralelismo em memória compartilhada



An Introduction to Parallel programming with OpenMP, Tim Mattson, 2022.

Programação em OpenMP

- Paralelismo em memória compartilhada
- API construída sobre
  - **Diretivas de compilação**
  - Métodos de biblioteca
  - Variáveis de ambiente
- As diretivas possuem **construções** e **cláusulas**
  - **Construções** - seções paralelas, divisão de dados ou tarefas, sincronização
  - **Cláusulas** - modificam ou especificam aspectos de construções

```
#pragma omp
```

Programação em OpenMP

- Paralelismo em memória compartilhada
- API construída sobre
  - **Diretivas de compilação**
  - Métodos de biblioteca
  - Variáveis de ambiente
- As diretivas possuem **construções** e **cláusulas**
  - **Construções** - seções paralelas, divisão de dados ou tarefas, sincronização
  - **Cláusulas** - modificam ou especificam aspectos de construções

```
#pragma omp
```

Programação em OpenMP

- Paralelismo em memória compartilhada
- API construída sobre
  - **Diretivas de compilação**
  - Métodos de biblioteca
  - Variáveis de ambiente
- As diretivas possuem **construções** e **cláusulas**
  - **Construções** - seções paralelas, divisão de dados ou tarefas, sincronização
  - **Cláusulas** - modificam ou especificam aspectos de construções

```
#pragma omp
```

Hello World!

A construção `parallel` executa o **bloco estruturado** em paralelo.

```
#include <omp.h>
int main(void) {
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    int nthreads = omp_get_num_threads();
    printf("Hello world from thread %d of %d\n", id,
     ↪  nthreads);
  }
}
```

Hello World!

Compilação com GCC

```
$ gcc -Wall -fopenmp -o hello hello.c
$ OMP_NUM_THREADS=4 ./hello
```

Compilação com Intel

```
$ icc -openmp -o hello hello.c
$ OMP_NUM_THREADS=4 ./hello
```

Hello World!

Saída do programa OpenMP (1)

```
Hello world from thread 0 of 4
Hello world from thread 1 of 4
Hello world from thread 2 of 4
Hello world from thread 3 of 4
```

Saída do programa OpenMP (2)

```
Hello world from thread 2 of 4
Hello world from thread 1 of 4
Hello world from thread 3 of 4
Hello world from thread 0 of 4
```

Hello World!

Saída do programa OpenMP (1)

```
Hello world from thread 0 of 4
Hello world from thread 1 of 4
Hello world from thread 2 of 4
Hello world from thread 3 of 4
```

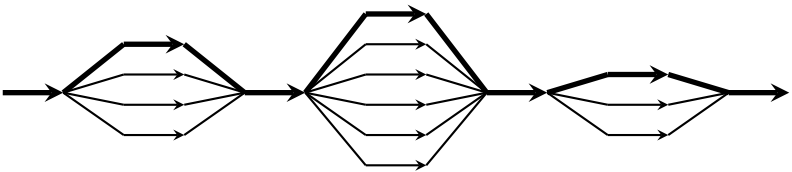
Saída do programa OpenMP (2)

```
Hello world from thread 2 of 4
Hello world from thread 1 of 4
Hello world from thread 3 of 4
Hello world from thread 0 of 4
```

Outline

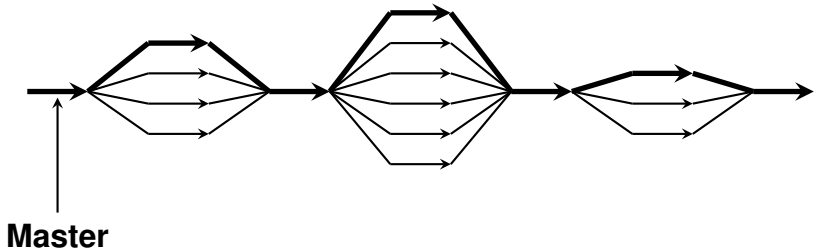
Modelo de Execução

Modelo de execução Fork/Join:



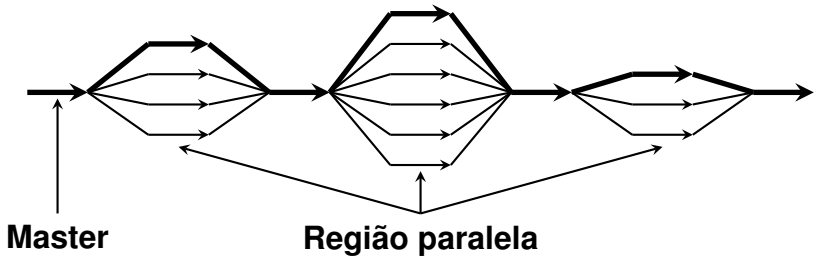
Modelo de Execução

Modelo de execução Fork/Join:



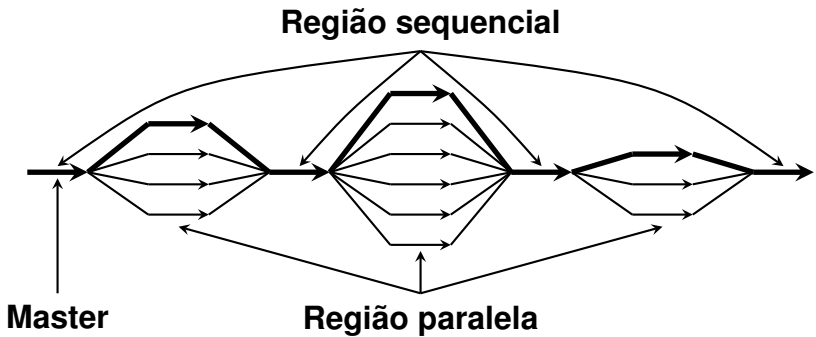
Modelo de Execução

Modelo de execução Fork/Join:



Modelo de Execução

Modelo de execução Fork/Join:



Modelo de Execução

Regiões paralelas e criação de threads.

```
omp_set_num_threads(4);
#pragma omp parallel
{
  int id = omp_get_thread_num();
  printf("Thread ID %d\n", id);
}
printf("Parte sequencial ...\n");
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  printf("Thread ID %d\n", id);
}
```

Modelo de Execução

Saída do programa OpenMP

```
Thread ID 0
Thread ID 3
Thread ID 2
Thread ID 1
Parte sequencial ...
Thread ID 1
Thread ID 0
```

Outline

Laços Paralelos

Código sequencial.

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

Laços Paralelos

OpenMP com divisão de trabalho estática.

```
#pragma omp parallel
{
  int id, i, nthreads, istart, iend;
  id = omp_get_thread_num();
  nthreads = omp_get_num_threads();

  istart = id * N / nthreads;
  iend = (id + 1) * N / nthreads;
  if( id == nthreads-1 ) iend = N;

  for(i= istart; i < iend; i++)
    a[i] = a[i] + b[i];
}
```

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
1960

Laços Paralelos

OpenMP parallel **for**.

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

Laços Paralelos

OpenMP parallel **for**.

```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                collapse (n)
                nowait
```

Laços Paralelos

Cláusula `schedule`.

```
#pragma omp for schedule(kind[,chunk])
```

- `static` - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.
- `dynamic` - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.
- `guided` - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.
- `runtime` - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.
- `auto` - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

Laços Paralelos

Cláusula `schedule`.

```
#pragma omp for schedule(kind[,chunk])
```

static - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.

dynamic - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.

guided - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.

runtime - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.

auto - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

Laços Paralelos

Cláusula `schedule`.

```
#pragma omp for schedule(kind[,chunk])
```

static - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.

dynamic - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.

guided - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.

runtime - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.

auto - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

Laços Paralelos

Cláusula `schedule`.

```
#pragma omp for schedule(kind[,chunk])
```

static - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.

dynamic - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.

guided - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.

runtime - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.

auto - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

Laços Paralelos

Cláusula `schedule`.

```
#pragma omp for schedule(kind[,chunk])
```

static - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.

dynamic - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.

guided - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.

runtime - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.

auto - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

Laços Paralelos

Cláusula `schedule`.

```
#pragma omp for schedule(kind[,chunk])
```

static - distribui blocos de iterações iguais para todas as threads e não altera essa configuração durante a execução do laço.

dynamic - cada thread remove um bloco de iterações de uma lista durante a execução do laço até que todas tenham sido executadas.

guided - as threads removem iterações dinamicamente. O tamanho do bloco de iterações inicia grande e diminui até o tamanho `chunk`.

runtime - política e bloco de iterações são definidos por funções da biblioteca ou pela variável de ambiente `OMP_SCHEDULE`.

auto - deixa a cargo da implementação do OpenMP escolher a política de escalonamento.

Laços Paralelos

Cláusula `schedule`

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

Laços Paralelos

Laços aninhados

```
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Cláusula collapse

```
#pragma omp parallel for collapse(2)
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Laços Paralelos

Laços aninhados

```
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Cláusula `collapse`

```
#pragma omp parallel for collapse(2)
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Laços Paralelos

Laços aninhados

```
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Cláusula `collapse`

```
#pragma omp parallel for collapse(2)
for(i = 0; i < N; i++) {
  for(j= 0; j < M; j++) {
    /* */
  }
}
```

Laço de $N \times M$.

Recomendado quando $N = O(nthreads)$.

Laços Paralelos

E agora ?

```
double media = 0.0f, A[N]; int i;
for(i = 0; i < N; i++) {
  media += A[i];
}
media = media / N;
```

- Estamos combinando valores em uma única variável (`media`)
- Dependência entre iterações que não pode ser eliminada facilmente
- Em programação paralela, essa é uma situação recorrente, chamada redução

Laços Paralelos

E agora ?

```
double media = 0.0f, A[N]; int i;
for(i = 0; i < N; i++) {
  media += A[i];
}
media = media / N;
```

- Estamos combinando valores em uma única variável (`media`)
- Dependência entre iterações que não pode ser eliminada facilmente
- Em programação paralela, essa é uma situação recorrente, chamada redução

Laços Paralelos

Cláusula `reduction`

`#pragma omp reduction(op : list)`

1. Cria uma cópia local (por thread) de cada variável inicializada de acordo com a operação
2. Atualização acontece na cópia local de cada thread
3. Ao final as cópias locais são reduzidas em um único valor e combinadas com o valor original

```
double media = 0.0f, A[N]; int i;
#pragma omp parallel for reduction (+:media)
for(i = 0; i < N; i++) {
  media += A[i];
}
media = media / N;
```

Laços Paralelos

Cláusula `reduction`

`#pragma omp reduction(op : list)`

1. Cria uma cópia local (por thread) de cada variável inicializada de acordo com a operação
2. Atualização acontece na cópia local de cada thread
3. Ao final as cópias locais são reduzidas em um único valor e combinadas com o valor original

```
double media = 0.0f, A[N]; int i;
#pragma omp parallel for reduction (+:media)
for(i = 0; i < N; i++) {
  media += A[i];
}
media = media / N;
```

Laços Paralelos

Cláusula `reduction`

```
#pragma omp reduction(op : list)
```

1. Cria uma cópia local (por thread) de cada variável inicializada de acordo com a operação
2. Atualização acontece na cópia local de cada thread
3. Ao final as cópias locais são reduzidas em um único valor e combinadas com o valor original

```
double media = 0.0f, A[N]; int i;
#pragma omp parallel for reduction (+:media)
for(i = 0; i < N; i++) {
  media += A[i];
}
media = media / N;
```

Exemplo: Pi

Cálculo do $\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx$

```
static long num_steps = 100000000;
double step;
int main (void) {
  int i;
  double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i= 1; i <= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  printf("\n pi with %ld steps is %lf\n ",num_steps,pi);
}
```

Exemplo: Pi

Cálculo do $\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx$

```
static long num_steps = 100000000;
double step;
int main (void) {
  int i;
  double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  for (i= 1; i <= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  printf("\n pi with %ld steps is %lf\n ",num_steps,pi);
}
```

Exemplo: Pi

Cálculo do π com OpenMP

```
#include <omp.h>
static long num_steps = 100000000; double step;
int main (void) {
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel for reduction(+:sum)
  for (i= 1; i <= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Exemplo: Pi

Cálculo do π com OpenMP

```
#include <omp.h>
static long num_steps = 100000000; double step;
int main (void) {
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel for reduction(+:sum)
  for (i= 1; i <= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Outline

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

shared - (**padrão**) compartilhada entre todas as threads.

private - cria uma nova cópia local para cada thread.

firstprivate - cria uma nova cópia local com o valor inicial da variável compartilhada.

lastprivate - atualiza o valor da variável compartilhada com o valor da última iteração sequencial.

reduction - descrita anteriormente, ela protege o conteúdo da variável por operação atômica.

threadprivate - definida na versão 4.0, cria uma cópia da variável para cada thread.

default - determina por padrão se as variáveis serão `shared` ou `none`, enquanto que `private` se aplica somente em Fortran.

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- A é compartilhado pelas threads. Valor 1
- B e C são privados em cada thread.
  - Valor inicial de B é indefinido
  - Valor inicial de C é 1
- Depois da região paralela ...
  - B e C voltam para 1
  - A é 1 ou modificado

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

informática
Universidade Federal de Santa Maria

Universidade Federal de Santa Maria
1960

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

Cláusulas de Dados

Considere o exemplo abaixo:

```
int A = 1, B = 1, C = 1;
#pragma omp parallel private(B) firstprivate(C)
```

- Quais os valores das variáveis dentro e depois da região paralela?
- `A` é compartilhado pelas threads. Valor 1
- `B` e `C` são privados em cada thread.
  - Valor inicial de `B` é indefinido
  - Valor inicial de `C` é 1
- Depois da região paralela ...
  - `B` e `C` voltam para 1
  - `A` é 1 ou modificado

Cláusulas de Dados

Cálculo do π

```
#include <omp.h>
static long num_steps = 100000000; double step;
int main (void) {
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;
  #pragma omp parallel for private(x) reduction(+:sum)
  for (i= 1; i <= num_steps; i++){
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Outline

Sincronização

- Sincronização é necessária em programação paralela
  - Coordenar a execução
  - Evitar condições de corrida (*deadlock*)
- OpenMP oferece diversar formas de sincronização
- Veremos três tipos:
  1. Barreira (`barrier`)
  2. Controle de fluxo (`master` e `single`)
  3. Exclusão mútua (`critical` e `atomic`)

Sincronização

- Sincronização é necessária em programação paralela
  - Coordenar a execução
  - Evitar condições de corrida (*deadlock*)
- OpenMP oferece diversar formas de sincronização
- Veremos três tipos:
  1. Barreira (`barrier`)
  2. Controle de fluxo (`master` e `single`)
  3. Exclusão mútua (`critical` e `atomic`)

Sincronização

- Sincronização é necessária em programação paralela
  - Coordenar a execução
  - Evitar condições de corrida (*deadlock*)
- OpenMP oferece diversar formas de sincronização
- Veremos três tipos:
  1. Barreira (`barrier`)
  2. Controle de fluxo (`master` e `single`)
  3. Exclusão mútua (`critical` e `atomic`)

Sincronização

- Sincronização é necessária em programação paralela
  - Coordenar a execução
  - Evitar condições de corrida (*deadlock*)
- OpenMP oferece diversar formas de sincronização
- Veremos três tipos:
  1. Barreira (`barrier`)
  2. Controle de fluxo (`master` e `single`)
  3. Exclusão mútua (`critical` e `atomic`)

Barreira

- Todas as threads de um grupo esperam até chegar nesse ponto

OpenMP `barrier`

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  A[id] = calculo1(id);

#pragma omp barrier
  B[id] = calculo2(id, A);
}
```

Barreira

- Todas as threads de um grupo esperam até chegar nesse ponto

OpenMP `barrier`

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  A[id] = calculo1(id);

#pragma omp barrier
  B[id] = calculo2(id, A);
}
```

Controle de Fluxo

- `master` – apenas a thread principal (master) executa
  - As demais threads ignoram o bloco e continuam a execução

```
#pragma omp parallel
{
#pragma omp master
  {
    printf("Eu sou a thread master\n");
  }
}
```

Controle de Fluxo

- `single` – apenas uma thread executa, as demais threads
  - As demais threads ignoram o bloco e esperam a execução
  - Implica em barreira implícita ao final do bloco

```
#pragma omp parallel
{
#pragma omp single nowait
  {
    int id = omp_get_thread_num();
    printf("Eu sou a thread %d\n", id);
  }
}
```

Exclusão Mútua

- `critical` – região crítica da região paralela, apenas uma thread executa
- `atomic` – exclusão mútua de uma região de memória (variável)
  - x binop = expr
  - x++ ou ++x
  - x– ou –x

```
#pragma omp parallel
{
#pragma omp critical
  valor = remove(A);

#pragma omp atomic
  total += valor;
}
```

Exclusão Mútua

- `critical` – região crítica da região paralela, apenas uma thread executa
- `atomic` – exclusão mútua de uma região de memória (variável)
  - x binop = expr
  - x++ ou ++x
  - x– ou –x

```
#pragma omp parallel
{
#pragma omp critical
  valor = remove(A);

#pragma omp atomic
  total += valor;
}
```

Exclusão Mútua

- `critical` – região crítica da região paralela, apenas uma thread executa
- `atomic` – exclusão mútua de uma região de memória (variável)
  - x binop = expr
  - x++ ou ++x
  - x– ou –x

```
#pragma omp parallel
{
#pragma omp critical
  valor = remove(A);

#pragma omp atomic
  total += valor;
}
```

Outline

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Métodos de Biblioteca

- `omp_set_num_threads(int)` - modifica o número de threads da próxima região paralela.
- `omp_get_num_threads()` - retorna o número de threads do grupo atual.
- `omp_get_thread_num()` - identificador da thread atual.
- `omp_get_max_threads()` - máx. de threads.
- `omp_in_parallel()` - retorna *true* se está em uma região paralela, *false* caso contrário.
- `omp_set_dynamic(int)` - n. de threads dinamicamente.
- `omp_get_dynamic()` - verifica se o ajuste dinâmico do número de threads está habilitado.
- `omp_num_procs()` - n. de processadores disponíveis.

Outline

Variáveis de Ambiente

- `OMP_NUM_THREADS(int)` - especifica o número de threads a serem usados nas regiões paralelas.
- `OMP_STACKSIZE(size[B|K|M|G])` - tamanho da pilha criada para cada thread pela implementação do OpenMP usada.
- `OMP_WAIT_POLICY(active | passive)` - define a política de espera em threads ociosas em barreiras e *locks*, sendo `active` para espera ativa (*busy wait*) e `passive` para espera passiva.
- `OMP_PROC_BIND(true | false)` - determina se as threads poderão mover para diferentes processadores em tempo de execução. O valor `true` define que as threads não mudam, enquanto `false` permite as migrações.

Variáveis de Ambiente

- `OMP_NUM_THREADS(int)` - especifica o número de threads a serem usados nas regiões paralelas.
- `OMP_STACKSIZE(size[B|K|M|G])` - tamanho da pilha criada para cada thread pela implementação do OpenMP usada.
- `OMP_WAIT_POLICY(active | passive)` - define a política de espera em threads ociosas em barreiras e *locks*, sendo `active` para espera ativa (*busy wait*) e `passive` para espera passiva.
- `OMP_PROC_BIND(true | false)` - determina se as threads poderão mover para diferentes processadores em tempo de execução. O valor `true` define que as threads não mudam, enquanto `false` permite as migrações.

Variáveis de Ambiente

- `OMP_NUM_THREADS(int)` - especifica o número de threads a serem usados nas regiões paralelas.
- `OMP_STACKSIZE(size[B|K|M|G])` - tamanho da pilha criada para cada thread pela implementação do OpenMP usada.
- `OMP_WAIT_POLICY(active | passive)` - define a política de espera em threads ociosas em barreiras e *locks*, sendo `active` para espera ativa (*busy wait*) e `passive` para espera passiva.
- `OMP_PROC_BIND(true | false)` - determina se as threads poderão mover para diferentes processadores em tempo de execução. O valor `true` define que as threads não mudam, enquanto `false` permite as migrações.

Variáveis de Ambiente

- `OMP_NUM_THREADS(int)` - especifica o número de threads a serem usados nas regiões paralelas.
- `OMP_STACKSIZE(size[B|K|M|G])` - tamanho da pilha criada para cada thread pela implementação do OpenMP usada.
- `OMP_WAIT_POLICY(active | passive)` - define a política de espera em threads ociosas em barreiras e *locks*, sendo `active` para espera ativa (*busy wait*) e `passive` para espera passiva.
- `OMP_PROC_BIND(true | false)` - determina se as threads poderão mover para diferentes processadores em tempo de execução. O valor `true` define que as threads não mudam, enquanto `false` permite as migrações.

https://joao-ufsm.github.io/par2023a/