INSTITUTO DE INFORMÁTICA UFRGS

UFRGS

Aula 5 – OpenMP

INF1008 – Programação Distribuída e Paralela

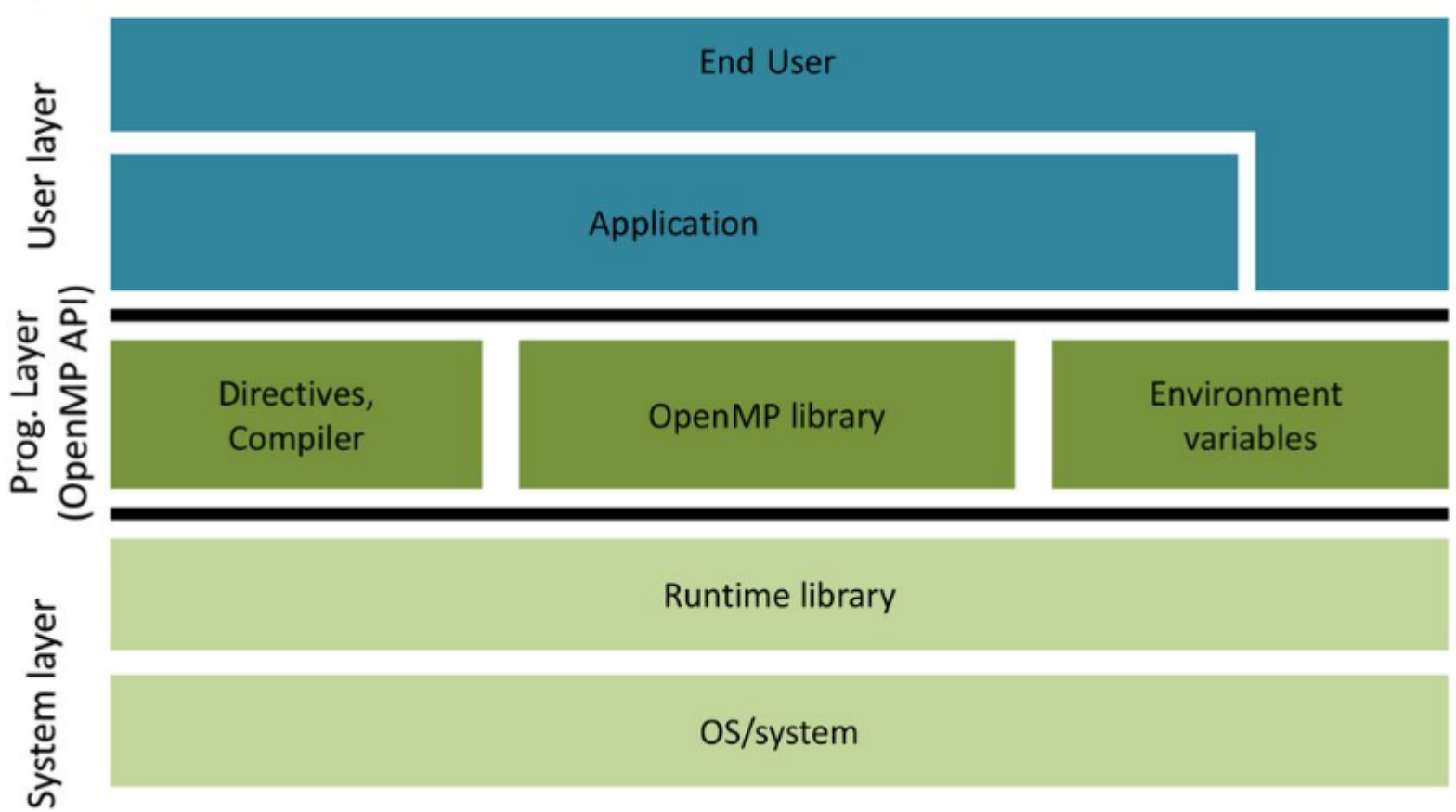
Prof. Arthur Francisco Lorenzon

05 de dezembro de 2022

Roadmap

- Introdução ao OpenMP

OpenMP


User layer
End User
Application
Prog. Layer (OpenMP API)
Directives, Compiler
OpenMP library
Environment variables
System layer
Runtime library
OS/system

OpenMP

Diretivas

- Região paralela
- Construtor de trabalho
- Tasks
- Offloading
- Afinidade
- SIMD
- Sincronização
- Atributos de compartilhamento de dados

Funções

- Número de threads
- Thread ID
- Ajuste dinâmico de threads
- Paralelismo aninhado
- Scheduling
- Limite de threads
- Nível de aninhamento
- Locking
- Timer
- ....

Variável de Ambiente

- Número de threads
- Tipo de scheduling
- Ajuste dinâmico de threads
- Paralelismo aninhado
- Stacksize
- Idle threads
- Limite de threads

OpenMP - Sintaxe

- Maioria dos construtores OpenMP são *diretivas de compilação* usando **pragmas**.
  - Para C/C++: **#pragma ...**

- Pragma x Linguagem:
  - **Pragma** não é linguagem!
  - **Pragma** é usado para fornecer informação adicional ao compilador/preprocessor em como processar o código anotado
  - **Pragma** é similar a **#include, #define...**

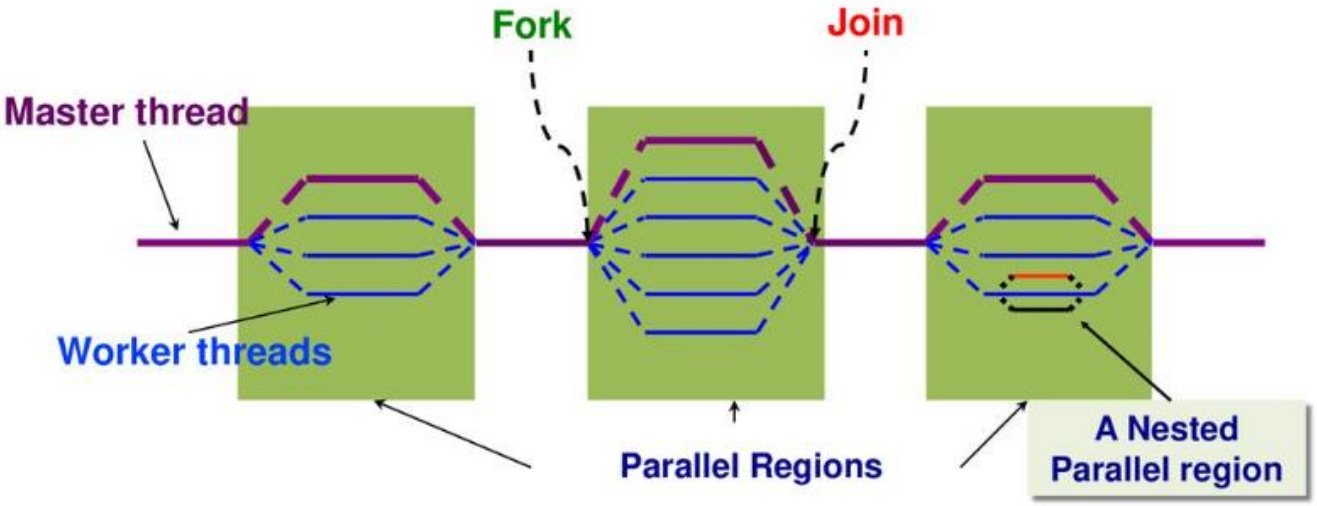
OpenMP - Sintaxe

- Nas aulas, focaremos na linguagem C e C++:
  - `#pragma omp construtor [clausula [clausula] ... ]`
- Incluir *header* da biblioteca do OpenMP:
  - `#include <omp.h>`
- *Flag* de compilação:
  - *-fopenmp*

OpenMP – *fork-join*

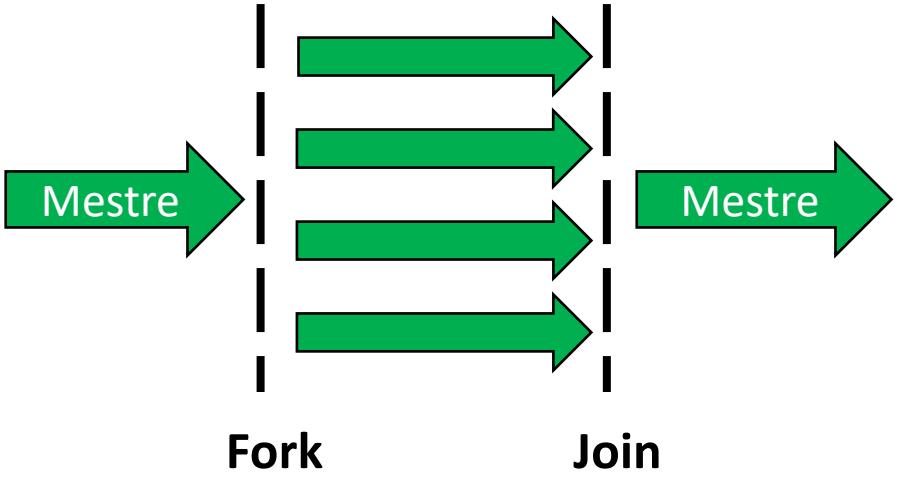
- Thread *mestre* cria múltiplas threads trabalhadoras conforme necessário.

- Região paralela é um bloco de código executado por todas as *threads* de maneira simultânea.



OpenMP – *fork-join*

- Thread *mestre* cria múltiplas threads trabalhadoras conforme necessário.
- Região paralela é um bloco de código executado por todas as *threads* de maneira simultânea.




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

int main(){

   printf("Região Sequencial - 1\n");

   #pragma omp parallel
   {
      printf("Região Paralela\n");
   }

   printf("Região Sequencial - 2\n");

   return 0;
}
```

Compilando: *gcc app.c -o app* ***-fopenmp***
Executando: ./app

OpenMP – *Definindo o número de Threads*

- Por padrão, o número de *threads* criadas é igual ao número de *hardware threads*

- Maneiras de definir o número de *threads*:
  - **Código**: `omp_set_num_threads(numeroThreads);`
  - **Código**: `#pragma omp parallel num_threads(numeroThreads);`
  - **Variável de ambiente**: `export OMP_NUM_THREADS=numeroThreads`

OpenMP – *Obtendo informações*

- Cada thread tem seu identificador no “team”:
  - *omp_get_thread_num();*
- Cada thread tem seu identificador no “team”:
  - *omp_get_num_threads();*

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

int main(){


   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      int totalThreads = omp_get_num_threads();
      printf(“ThreadId = %d, Total = %d\n”,
            id, totalThreads);
   }

   return 0;
}
```

OpenMP – *Construtores de Trabalho*

- Um construtor de trabalho divide a execução de uma região paralela entre as *threads* de um *team*
- Construtor “for”:
  - Divide o número de iterações entre as *threads*
  - Cada *thread* obtém um ou mais *chunks* (conjunto de iterações)

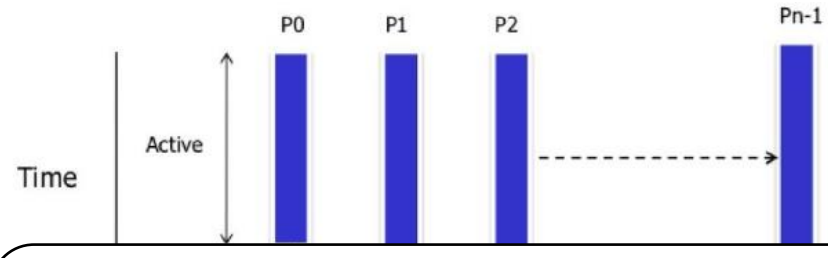
```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);

   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      #pragma omp for
      for(int i = 0; i < N; i++){
          C[i] = A[i] + B[i];
          printf("Thread %d: C[%d] = A[%d] + B[%d]", id, i, i, i);
      }
   }
   return 0;
}
```

OpenMP – *Construtores de Trabalho*

- Por padrão, existe uma barreira no final do “omp for”




```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);

   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      #pragma omp for nowait
      for(int i = 0; i < N; i++){
          C[i] = A[i] + B[i];
          printf("Thread %d: C[%d] = A[%d] + B[%d]", id, i, i, i);
      }
   }
   return 0;
}
```

Útil entre dois “*loops*” consecutivos e independentes

OpenMP – *Construtores de Trabalho*

- Cláusula *schedule*:
  - **Static**: distribui as iterações em blocos sobre as *threads* em uma maneira *round-robin*
  - **Dynamic**: quando uma thread finaliza a computação do seu bloco, recebe a próxima porção de trabalho.
  - **Guided**: mesmo que o dinâmico. Mas, o tamanho do bloco diminui exponencialmente.
  - **Auto**: o compilador ou *runtime* do OpenMP decide qual o melhor para usar.
  - **Runtime**: o esquema de escalonamento é definido de acordo com a variável de ambiente: *OMP_SCHEDULE*

OpenMP – *Construtores de Trabalho*

- Testar diferentes esquemas de *schedule* e valores de *chunk*.

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);
   int chunk = 2;

   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      #pragma omp for schedule(static, chunk)
      for(int i = 0; i < N; i++){
          C[i] = A[i] + B[i];
          printf("Thread %d: C[%d] = A[%d] + B[%d]", id, i, i, i);
      }
   }
   return 0;
}
```

OpenMP – *Construtores de Trabalho*

- ***Collapse* clause**

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);

   #pragma omp parallel for colapse(2)
   for(int i = 0; i < N; i++){
       for(int j = 0; j < N; j++){
          C[i] += A[j] + B[j];
       }
   }

   return 0;

}
```

OpenMP – *Construtores de Trabalho - Sections*

- Atribui um bloco diferente de trabalho para cada *thread*

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

int main(){

   #pragma omp parallel
   {
      #pragma omp sections
      {
         #pragma omp section
         {
            calculo_x();
         }
         #pragma omp section
         {
            calculo_y();
         }
         #pragma omp section
         {
            calculo_z();
         }
      }
   }
   return 0;
}
```

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);
   int chunk = 2;

   #pragma omp parallel
   {
      #pragma omp for schedule(static, chunk)
      for(int i = 0; i < N; i++){
         C[i] = A[i] + B[i];
      }
   }
   return 0;
}
```

Tarefa:

Implementar o código acima com sections!

OpenMP – *Construtores de Trabalho - Master*

- Denota um bloco estruturado para ser executado apenas pela *thread* mestre

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);
   int chunk = 2;

   #pragma omp parallel
   {
      #pragma omp master
      {
         printf("Iniciando região paralela\n");
      }
      #pragma omp for schedule(static, chunk)
      for(int i = 0; i < N; i++){
          C[i] = A[i] + B[i];
      }
   }
   return 0;
}
```

OpenMP – *Construtores de Trabalho - Single*

- Denota um bloco estruturado para ser executado por apenas uma *thread*

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);
   int chunk = 2;

   #pragma omp parallel
   {
      #pragma omp single
      {
         printf("Iniciando região paralela\n");
      }
      #pragma omp for schedule(static, chunk)
      for(int i = 0; i < N; i++){
          C[i] = A[i] + B[i];
      }
   }
   return 0;
}
```

OpenMP – *Cláusula Private/Shared*

- private(var) cria uma cópia local de *var* para cada *thread*
  - O valor não é inicializado;

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);
   int chunk = 2;
   int i = 0;

   #pragma omp parallel private(i)
   {
      #pragma omp for schedule(static, chunk)
      for(i = 0; i < N; i++){
          C[i] = A[i] + B[i];
      }
   }
   return 0;
}
```

OpenMP – *Cláusula Private/Shared*

- shared(var) específica que uma ou mais variáveis devem ser compartilhadas entre as *threads*

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int A[N], B[N], C[N];
   initArrays(A, B, C);
   int chunk = 2;
   int i = 0;

   #pragma omp parallel private(i) shared(A, B, C)
   {
      #pragma omp for schedule(static, chunk)
      for(i = 0; i < N; i++){
          C[i] = A[i] + B[i];
      }
   }
   return 0;
}
```

OpenMP – *Cláusula reduction*

- Específica que uma ou mais variáveis privadas para cada *thread* são objetos de uma operação de redução no final da região paralela

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int B[N], C[N];
   initArrays(B, C);
   int i = 0;

   #pragma omp parallel private(i,soma) shared(A, B, C)
   {
      #pragma omp for
      for(i = 0; i < N; i++){
          soma += A[i] + B[i];
      }
   }
   return 0;
}
```

O que acontece com a variável soma?

OpenMP – *Cláusula reduction*

- Específica que uma ou mais variáveis privadas para cada *thread* são objetos de uma operação de redução no final da região paralela

```
#include <stdio.h>
#include <omp.h>
#define N 32

int main(){
   int B[N], C[N];
   initArrays(B, C);
   int i = 0;

   #pragma omp parallel private(i) shared(A, B, C) reduction(+:soma)
   {
      #pragma omp for
      for(i = 0; i < N; i++){
          soma += A[i] + B[i];
      }
   }
   return 0;
}
```

OpenMP – Outras cláusulas

- *firstprivate:* Específica que cada *thread* deve ter sua própria instância de uma variável e que a variável deve ser inicializada com o valor da variável, pois ela existe antes do construtor paralelo
- *lastprivate*: Específica que a versão da variável do contexto delimitador é definida como a versão privada de qualquer *thread* que execute a iteração final (construtor loop) ou a última seção (*#pragma sections*)
- *default*: Específica o comportamento de variáveis sem escopo em uma região paralela.
- *copyin*: Permite que as *threads* acessem o valor da *thread* principal para uma variável *threadprivate*.
- *copyprivate:* Específica que uma ou mais variáveis devem ser compartilhadas entre todas as *threads*

Definição do Trabalho

Próxima aula...

- Paralelismo com *tasks*