*PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS*

*Instituto de Ciências Exatas e Informática*

*Departamento de Ciência da Computação - Curso de Ciência da Computação*

*Disciplina: Seminários II Segundo semestre de 2019*

*Professor: Saulo Augusto de Paula Pinto —* [***saulo@pucminas.br***](mailto:saulo@pucminas.br)

***Primeiro Roteiro***

***Introdução ao OpenMP***

**Atenção: poste no SGA um arquivo texto (editável) contendo as respostas às questões propostas neste roteiro até 23:59h do dia 30/09/2019.**

Vamos iniciar o trabalho com a API *OpenMP* criando um certo número de *threads*.

Abra seu editor predileto no Linux e copie o seguinte código:

#include <stdio.h>

#include <omp.h>

int main (){

#pragma omp parallel

{

printf("Oi mundo! Sou a thread %d \n", omp\_get\_thread\_num());

}

}

Salve seu programa em um arquivo chamado *openmp1.c*.Para compilar o código você deve informar ao compilador (o gcc), que você está usando a API OpenMP. Faça isso assim:

gcc openmp1.c -o openmp1 –fopenmp

Agora rode seu programa com

./openmp1

A diretiva

#pragma omp parallel

Cria um “construtor paralelo”, no jargão do OpenMP. Isso significa que o bloco de código que o segue é replicado para as *threads* e executado em paralelo nos vários núcleos de uma máquina.

A função *omp\_get\_thread\_num()* retorna o número identificador da *thread* “corrente” (em execução). Este identificador é atribuído a cada *thread* criada iniciando de zero.

**Questão 1**

1. Rode o programa várias vezes (umas 10 vezes!) e observe a saída produzida. Ela é sempre igual? Explique.
2. Descubra o número de núcleos (*cores*) da máquina que você está usando e relacione ao número de *threads* criadas no código. Reporte o modelo da máquina.

Você pode escolher o número de threads que deseja criar. Basta usar o procedimento *omp\_set\_num\_threads*(). Rode o seguinte código, que é igual ao que você está trabalhando, mas com a adição da chamada ao referido procedimento e observe a saída. Nele são criadas oito *threads*. Tente com outros números!

#include <stdio.h>

#include <omp.h>

int main (){

omp\_set\_num\_threads(8);

#pragma omp parallel

{

printf("Oi mundo! Sou a thread %d \n", omp\_get\_thread\_num());

}

}

Vamos exercitar bastante esta capacidade de variar o número de *threads* nas aulas porvir... Agora vamos prosseguir com mais características da API OpenMP.

O seguinte código apenas conta de zero a quatro sequencialmente:

#include <stdio.h>

#include <omp.h>

int main (){

for (int i = 0; i < 5; i++)

printf("i = %d \n", i);

}

Ao ser executado, ele produz a saída:

i = 0

i = 1

i = 2

i = 3

i = 4

O código seguinte usa o construtor paralelo. Rode-o algumas vezes e observe a saída.

#include <stdio.h>

#include <omp.h>

int main (){

#pragma omp parallel

{

for (int i = 0; i < 5; i++)

printf("i = %d \n", i);

}

}

Agora vamos alterar um pouco as coisas e ver o que acontece. Retire as duas chaves do construtor paralelo e garanta que a diretiva fique na linha imediatamente acima do “for”. Rode o código novamente algumas vezes.

Observe que o construtor paralelo replica um bloco de código. Assim, se vamos replicar um “for” ou uma chamada de métodos, não precisamos das chaves, pois um “for” em C define um bloco de código, assim como um método (função ou procedimento).

Agora, vamos tentar fixar o padrão de programação paralela “fork-join”. Para isso, vamos criar seções paralelas e sequenciais no código. Examine o código que segue cuidadosamente.

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

void openMP1() {

#pragma omp parallel

{

printf("Oi mundo! Sou a thread %d \n", omp\_get\_thread\_num());

}

}

void openMP2() {

#pragma omp parallel

{

printf("Sou a thread %d \n", omp\_get\_thread\_num());

for (int i = 0; i < 3; i++)

printf("i = %d \n", i);

}

}

int main (){

printf("Região sequencial 1. Thread master. Número de threads: %d\n", omp\_get\_num\_threads());

printf("Numero de processadores: %d\n", omp\_get\_num\_procs());

printf("Entrando na primeira região paralela...\n");

openMP1();

printf("Fim da primeira região paralela...\n");

printf("\nRegião sequencial 2. Thread master. Número de threads: %d\n", omp\_get\_num\_threads());

printf("Numero de processadores: %d\n", omp\_get\_num\_procs());

printf("Entrando na segunda região paralela...\n");

openMP2();

printf("Fim da segunda região paralela...\n");

printf("\nRegião sequencial 3. Thread master. Número de threads: %d\n", omp\_get\_num\_threads());

printf("Numero de processadores: %d\n", omp\_get\_num\_procs());

printf("Terminando o programa...\n");

}

**Questão 2**

Explique o padrão de programação “fork-join” usando o código que acabamos de estudar.

Agora vamos observar algo interessante. Vamos mudar o lugar da declaração da variável i e ver o resultado nos códigos que seguem. Rode primeiro este:

int main (){

#pragma omp parallel

for (int i = 0; i < 5; i++)

printf("i = %d \n", i);

}

Agora declare o “i” antes da diretiva #pragma e rode o código algumas vezes. Assim:

int main (){

int i;

#pragma omp parallel

for (i = 0; i < 5; i++)

printf("i = %d \n", i);

}

**Questão 3**

1. Por que faltaram alguns valores?
2. Qual a diferença entre os dois códigos, em termos de paralelismo, *threads*, memória privativa e compartilhada?

Replicar código é algo muito interessante e importante. Vamos agora dar um passo a mais que é fundamental para ganharmos tempo: em vez de replicar o código de um “for”, vamos distribuir suas iterações entre as *threads* (vamos distribuir a carga do programa). Olhe a alteração no código com a inclusão da cláusula “for” no construtor paralelo.

int main (){

#pragma omp parallel for

for (int i = 0; i < 5; i++)

printf("i = %d \n", i);

}

Rode o código e verá que todos os valores foram impressos, mas não necessariamente em ordem.

Como o *loop for* do código tem 5 iterações, uma das *threads* vai ficar com mais de uma iteração para executar (considerando que são quatro *threads* por default). Vamos detectar qual é. O código que segue identifica a *thread*.

int main (){

#pragma omp parallel for

for (int i = 0; i < 5; i++) {

printf("Sou a thread %d \n", omp\_get\_thread\_num());

printf("i = %d \n", i);

}

}

Execute o código várias vezes e observe que saídas como esta podem ocorrer:

Sou a thread 0

i = 0

Sou a thread 0

Sou a thread 3

i = 4

Sou a thread 1

i = 2

Sou a thread 2

i = 3

i = 1

Ou seja, mesmo a identificação das *threads* que são responsáveis por um comando pode ficar comprometida ou difícil no paralelismo puro e simples! Tente dizer qual *thread* foi responsável pela impressão de cada “i = “ na saída mostrada... É possível?

Mas, por *default* do OpenMP, a primeira *thread*, a zero, recebe a iteração “a mais”. Nas próximas aulas veremos como alterar isso no código.

**Questão 4**

1. Explique porque são produzidas saídas como aquela mostrada para este último código.
2. Mude o número de iterações do *loop* para 6 e descubra qual thread ficou com a iteração acrescentada.
3. Agora vá aumentando o número de iterações para 7, depois para 8, 9,... Observe e explique o padrão do OpenMP para alocar as iterações do *loop*.
4. Vá aumentando o número de iterações no *loop* e vá aumentando o número de *threads* com o procedimento *omp\_set\_num\_threads()*, como já foi feito. Crie o máximo de *threads* possível e reporte este número e as características da máquina que você rodou. Use uma variável chamada MAX\_THREADS, do tipo *int*, tanto para parâmetro de *omp\_set\_num\_threads()* quanto para o número de iterações do *loop,* e varie seu valor!