Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Sistemas de Computação
Laboratório de Sistemas Distribuídos e de Programação Concorrente

Caderno de Desafio para Programação Paralela

Caderno 01 - Introdução ao OpenMP: um primeiro contato

por
Guilherme Martins
Paulo Sérgio Lopes de Souza

São Carlos/BR, abril de 2020

Sumário

1. Desafio

Calcule a soma de todos os elementos de um vetor em *OpenMP*.

Considere o seguinte vetor: 3, 6, 5, 2, 4, 6, 3, 2.
A soma de todos os elementos deste vetor é: 3 + 6 + 5 + 2 + 4 + 6 + 3 + 2 = 31

Considere como entrada um vetor de elementos contendo, na primeira linha, a dimensão do vetor e, na segunda linha, os elementos do vetor. As linhas são separadas por uma quebra de linha simples e as colunas por um único espaço. O vetor deve ser lido por meio do redirecionamento de fluxo de entrada (***stdin***), ou seja, não é necessário usar ponteiros para arquivos.

Conteúdo do arquivo de entrada, por exemplo, ***entrada.txt***:
8
3 6 5 2 4 6 3 2

Para executar no ***bash***, por exemplo, utilize este padrão:
.\somavet < entrada.txt <enter>

Obs: na linha de comando acima, considera-se que o programa foi inserido em ***somavet.c*** e o executável chama-se ***somavet*** e está no diretório atual.

A saída deve ser impressa, utilizando o *output (**stdout**)* padrão, apenas com o elemento resposta.

Saída:
31

2. O que você precisa saber para resolver o desafio

O *OpenMP* é uma *API* (Interface de Programação de Aplicativos) para programação paralela com memória compartilhada em *C*, *C*++ e *Fortran*. A primeira versão (1.0) do *OpenMP* foi divulgada em 1997. Atualmente, o padrão encontra-se na versão 5.0, de novembro de 2018, atualizada em janeiro de 2020.

O *OpenMP* implementa o paralelismo por diretivas de compilação, ou ***#pragmas***, que são instruções providas ao compilador indicando como paralelizar um bloco de código.

O *OpenMP* utiliza paralelismo **implícito**, de forma em que o programador não tem controle direto de como será feita a paralelização. O programador indica explicitamente o que deve ser paralelizado e o *OpenMP* decide como implicitamente paralelizar o código indicado.

Este material considera o uso da linguagem *C* e do *Linux*. Para compilar um código em *OpenMP*, deve-se usar um shell, como o ***bash*** e o compilador ***gcc*** com o parâmetro ***-fopenmp*.**

gcc -o codigo codigo.c -fopenmp

Um código *.c*, para fazer o uso de primitivas do *OpenMP*, deve incluir em seu cabeçalho:

#include <omp.h>

As construções de programação do *OpenMP* começam com a diretiva:

#pragma omp

De uma forma geral, as diretivas do *OpenMP* têm a estrutura:

#pragma omp diretiva [lista de cláusulas]

O OpenMP também tem funções, todas começando com o prefixo “omp_”, e variáveis de ambiente, estas definidas no shell do SO e usadas dentro dos programas. Detalhes dessas funções e variáveis de ambiente não são detalhadas neste caderno e serão trabalhadas mais adiante em outros materiais do curso.

2.1. Diretiva parallel

Os programas *OpenMP* executam sequencialmente até encontrarem a diretiva ***parallel***. Tal instrução é responsável por criar um grupo de ***threads***. Uma forma de especificar quantas *threads* serão criadas é utilizar a cláusula ***num_threads (numero_de_threads).*** Um exemplo da utilização da diretiva *parallel* seria:

```
#pragma omp parallel num_threads (6)
 {
        //meu código paralelo
}
```

O trecho "***meu código paralelo***" acima é executado concorrentemente por 6 *threads*, sendo que são geradas 5 novas *threads* que executam concorrentemente à *thread* mestre já existente.

Exemplo de um *Hello World* em *OpenMP*:

```
#include<stdio.h>
#include<omp.h>
int main(){
  #pragma omp parallel num_threads(4)
  {
    printf("For good luck, Hello World from thread %d. \n", omp_get_thread_num());
  }
}
```

Uma possível saída deste código é (a sequência de impressão das threads varia):

```
For good luck, Hello World from thread 0.
For good luck, Hello World from thread 3.
For good luck, Hello World from thread 1.
For good luck, Hello World from thread 2.
```

O diagrama abaixo ilustra uma possível sequência de execução das *threads* deste programa:

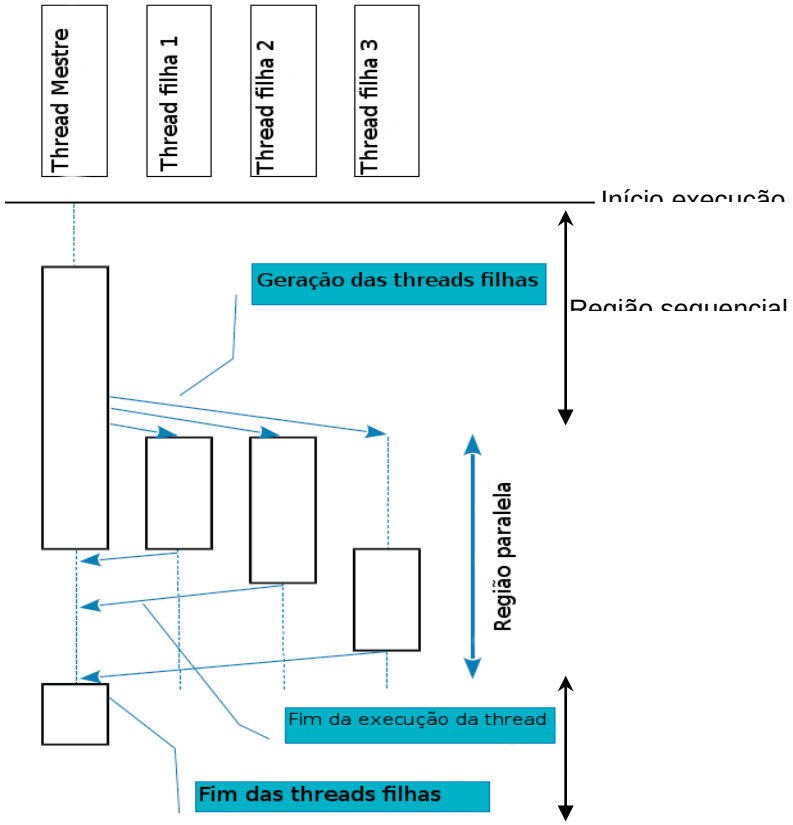

Figura 1: Diagrama para execução do "*Hello World*" com 4 *threads*. Adaptado de Barlas (2014).

2.2. Escopo de variáveis

Dentro de um bloco paralelo, as variáveis podem assumir diferentes escopos. Dois escopos principais são ***shared*** ou ***private***.

Variáveis do tipo ***shared***, ou compartilhadas, são visíveis para todas as *threads*, ou seja, se uma *thread* alterar um valor de uma variável ***shared***, este valor será refletido em todas demais *threads*. O *OpenMP* não protege regiões críticas. É responsabilidade do programador protegê-las.

Variáveis do tipo ***private***, ou privadas, são individuais a cada *thread*, ou seja, cada *thread* tem sua cópia da variável e, caso seu valor seja alterado, o mesmo não será refletido nas demais *threads*. As variáveis do tipo *private* são inicializadas com 0 (zero).

Para definir o escopo de uma variável em *OpenMP*, basta utilizar a cláusula correspondente em uma instrução ***parallel***.

Exemplo:

```
#include <stdio.h>
#include <omp.h>
int main(){
  int num_1=1, num_2=1;
  #pragma omp parallel num_threads (8) private (num_1) shared (num_2)
  {
     num_1 += num_1;
     #pragma omp critical (rc_num2)
     {
        num_2 += num_2;
     }
  }
  printf("num_1=%d\n num_2=%d\n", num_1, num_2);
  return(0);
}
```

A saída deste código será:

num_1=1 num_2=256

Neste exemplo, ambas as somas são replicadas nas 8 *threads* determinadas por ***num_threads(8)***, sendo que cada uma possui uma cópia diferente e local de ***num_1*** que é somada no escopo da região paralela do ***#pragma omp parallel***, devido à cláusula ***private(num_1)***. Como o ***printf( )*** está localizado fora da região paralela, o valor global não alterado de ***num_1*** é impresso. Os valores locais de ***num_1*** são perdidos ao se terminar a região paralela. A variável ***num_2*** é compartilhada devido à cláusula ***shared(num_2)*** e, por isso, sua soma na região paralela representa uma **região crítica** que precisa ser protegida. Há diferentes formas de se fazer esta proteção e neste exemplo usamos a diretiva ***#pragma omp critical (rc_num2)*** que garante a exclusão mútua das 8 *threads* em execução na soma de ***num_2***. O rótulo ***rc_num2*** é opcional e permite diferenciar as regiões críticas de modo que uma região crítica não impeça, desnecessariamente, uma outra de executar. O valor de ***num_2*** dobra a cada thread que passa pela região crítica e o valor impresso para esta variável no final é 256.

A *thread* mestre aguarda que as outras 07 *threads* terminem as suas execuções para poder executar o ***printf( )***. Em outras palavras, há uma sincronização implícita no final do bloco paralelo. Caso não se deseje esta sincronização, a cláusula ***nowait*** deve ser utilizada junto à diretiva ***#pragma omp parallel***.

Outros possíveis escopos de variáveis em *OpenMP* são:
firstprivate: Cada *thread* tem sua cópia da variável, porém cada cópia é inicializada com o valor que a variável possuía na região sequencial antes da região paralela.
lastprivate: Cada *thread* tem sua cópia da variável, porém o valor desta variável poderá ser repassado à porção sequencial posterior à região paralela, sendo que, no caso de laços (***loop for***), o valor repassado representa o valor que seria da última iteração sequencial do laço (*thread* que contém a última iteração); e no caso de ***sections***, o valor repassado será da última seção no código fonte.
threadprivate: Cada *thread* tem sua cópia local da variável, porém as variáveis são persistentes, ou seja, não são destruídas após o término de um bloco paralelo e podem ser reutilizadas em blocos paralelos posteriores. Usada como uma diretiva: ***#pragma omp threadprivate(var_list)***, usada no escopo de variáveis globais que serão locais e persistentes nas regiões paralelas.
copyin: Cláusula utilizada em conjunto com a diretiva ***#pragma omp threadprivate (var_list)***, inicializa as variáveis locais com o valor da *thread* mestre.

2.3. Diretiva parallel for

Loops que não possuem dependências entre suas iterações podem ser paralelizados com a diretiva *parallel for*. Sua forma básica é:

```
#pragma omp for [cláusulas]
 {
        // estrutura for
}
```

Esta diretiva divide as iterações do ***for*** em blocos e os atribui a cada uma das *threads* disponíveis, que irão realizar o processamento em paralelo. Lembre-se que o ***#pragma omp for [cláusulas]*** deve estar dentro de um bloco paralelo!

Exemplo da soma de dois vetores em *OpenMP*:

Considere a soma dos vetores *A* e *B*, armazenada no vetor *C*. O código abaixo representa uma possível implementação paralela em *OpenMP*:

```
#include <stdio.h>
#include <omp.h>
#define N 10000

int main(){
  int A[N];
  int B[N];
  int C[N];
  int i=0;
#pragma omp parallel num_threads(8) private(i)
  {
    #pragma omp for
    for(i = 0; i < N; i++)
      A[i]=B[i]=i;
  }
  #pragma omp parallel for num_threads(4) private(i)
    for(i = 0; i < N; i++){
      C[i]=A[i]+B[i];
    }
}
```

Observe que as duas formas distintas utilizadas para representar o *parallel for* são equivalentes.

2.4. Cláusula reduction

O *OpenMP* permite a realização de operações de redução por meio da cláusula *reduction*. As operações de redução podem ser utilizadas para acumular (reduzir) valores resultantes de operações de diferentes *threads* em uma única variável compartilhada. A Figura 2 abaixo ilustra a operação de redução de quatro somas.

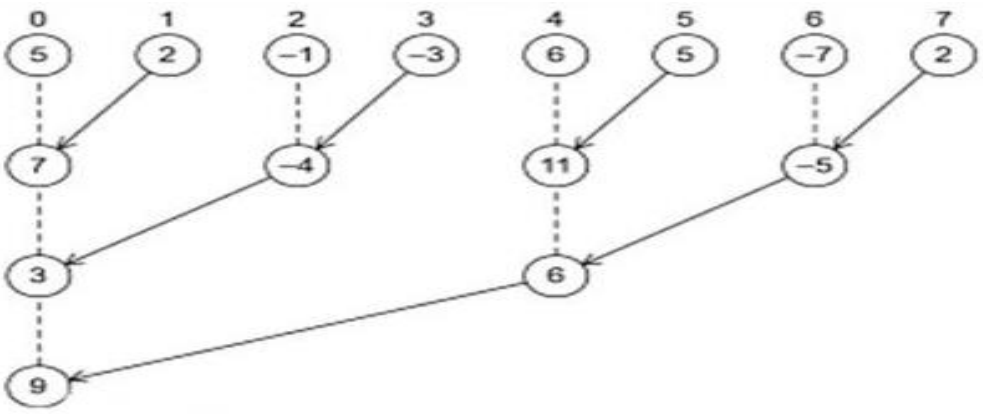

Figura 2: Redução das somas de 8 elementos. Adaptado de https://slideplayer.com/slide/7031801/

O trecho de código abaixo ilustra o cálculo do PI usando *OpenMP* e operação de redução.

```
#pragma omp parallel default(private) shared (npoints) reduction(+: sum)
num_threads(8)
{
  num_threads = omp_get_num_threads();
  sample_points_per_thread = npoints / num_threads;
  sum = 0;
  for (i = 0; i < sample_points_per_thread; i++) {
    rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
    rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
    if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
        (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        sum ++;
  }
}
```

Fonte: Grama, A., Kumar, V., Gupta, A., & Karypis, G. (2003). *Introduction to parallel computing*. Pearson Education. Capítulo 7.10.

As operações padrão do *OpenMP* para a cláusula *reduction* estão listadas abaixo:

Operações matemáticas: +, *, −, max, min

Operações lógicas: &, &&, |, ||, ^

É possível também definir novas operações para a cláusula *reduction*, o que não será abordado neste material. Para mais informações sobre este tópico, consulte http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-reduction.html

3. Um ponto de partida para a solução do desafio proposto

3.1. Implementação sequencial do desafio

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

int main(void){
   int tam,*vetor,i,soma=0;
   fscanf(stdin, "%d\n", &tam); //Lê o tamanho do vetor
   vetor=(int*)malloc(tam*sizeof(int)); //Aloca o vetor do tamanho lido
   for(i=0;i<tam;i++)
         fscanf(stdin, "%d",&(vetor[i])); //Lê os elementos do vetor

   for(int i=0;i<tam;i++)
         soma+=vetor[i]; //Faz a soma dos elementos
   printf("%d\n", soma); //Imprime a soma dos elementos
   free(vetor); //Desaloca o vetor
}
```

4. Referências Bibliográficas

Livro texto

Grama, A., Kumar, V., Gupta, A., & Karypis, G. (2003). *Introduction to parallel computing*. Pearson Education. Capítulo 7.10.

Bibliografia Complementar

Barlas, G. (2014). *Multicore and GPU Programming: An integrated approach*. Elsevier. Capítulo 4.

Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier. Capítulo 5.

Rauber, T., & Rünger, G. (2013). *Parallel Programming*. Springer. Second edition. Capítulo 6.