Programação de Arquiteturas com Memória Compartilhada Utilizando OpenMP

MCZA020-13 - Programação Paralela

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

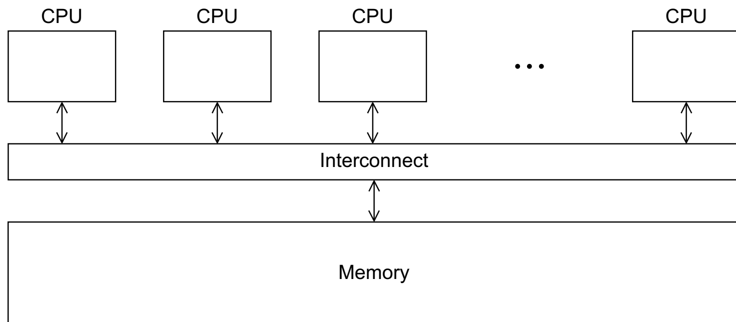
- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em: https://www.cs.usfca.edu/~peter/ipp/
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: http://pngimg.com

OpenMP

- Escrevendo programas usando OpenMP
- Usar OpenMP para paralelizar laços seriais com pequenas mudanças no código fonte
- Explorar paralelismo de tarefas
- Sincronização explícita de threads
- Problemas típicos em programação para máquinas com memória compartilhada

- Uma API para programação paralela em memória compartilhada.
- MP = multiprocessing
- Projetada para sistemas no quais todas as threads ou processos podem, potencialmente, ter acesso à toda memória disponível.
- O sistema é visto como uma coleção de núcleos ou CPUs, no qual todos eles têm acesso à memória principal.

Um sistema com memória compartilhada



Pragmas

- Instruções especiais para pre-processamento.
- Tipicamente adicionadas ao sistema para permitir comportamentos que não são parte do especificação básica de C.
- Compiladores que não suportam pragmas ignoram-nos.

```
#pragma
```

Hello World!

UFABC

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void) {
6     int my_rank = omp_get_thread_num();
7     int thread_count = omp_get_num_threads();
8     printf("Hello from thread %d of %d\n", my_rank,
    ↪   thread_count);
9  }
10
11 int main(int argc, char* argv[]) {
12    int thread_count = strtol(argv[1], NULL, 10);
13
14 #pragma omp parallel num_threads(thread_count)
15    Hello();
16
17    return 0;
18 }
```

Compilando e executando

UFABC

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4
```

compilando

executando com 4 threads

resultados possíveis

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

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

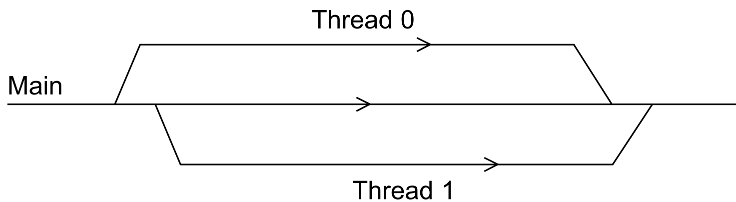
```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

OpenMP pragmas


```
1 #pragma omp parallel
```

- Diretiva paralela mais básica.
- O número de threads que executam o bloco que segue o pragma é determinado pelo sistema de runtime.

Um processo de duas threads fazendo fork e join



Cláusula

UFABC

- Uma cláusula (*clause*) é texto que modifica uma diretiva.
- A cláusula `num_threads` pode ser adicionada à uma diretiva paralela.
- Permite o programador especificar o número de threads que devem executar no bloco que segue o pragma.

```
1  #pragma omp parallel num_threads ( thread_count )
```

Alguns comentários

UFABC

- Alguns sistemas podem limitar o número de threads que podem ser executadas.
- O padrão OpenMP não garante que serão iniciadas `thread_count` threads.
- A maioria dos sistemas pode iniciar centenas ou, até mesmo, milhares de threads.
- A não ser que desejemos iniciar um número muito grande de threads, quase sempre conseguiremos o número de threads desejado.

- Em OpenMP, o conjunto de threads formado pela thread original e pelas novas threads é chamado team.
- A thread original é chamada master, e as threads adicionais são chamadas slaves.

Caso o compilador não tenha suporte ao OpenMP

UFABC

```c
#include <omp.h>

//Em vez do acima escrever:

#ifdef _OPENMP
#include <omp.h>
#endif
```

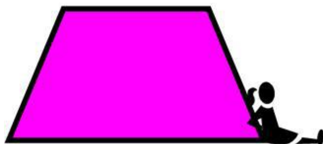
Caso o compilador não tenha suporte ao OpenMP

UFABC

- E no código cercar as chamadas ao OpenMP com `ifdef`

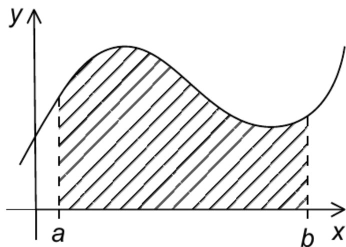
```
#ifdef _OPENMP
int my_rank = omp_get_thread_num ( );
int thread_count = omp_get_num_threads ( );
#else
int my_rank = 0;
int thread_count = 1;
#endif
```

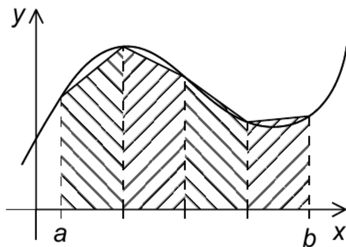
A aproximação trapezoidal



A aproximação trapezoidal

A aproximação trapezoidal





```
1  // Entradas: a, b e n
2  h = (b -a) / n;
3  approx = (f(a) + f(b))/2.0;
4  for (i = 1; i <= n-1; i++) {
5      x_i = a + i * h;
6      approx += f (x_i);
7  }
8  approx = h * approx;
```

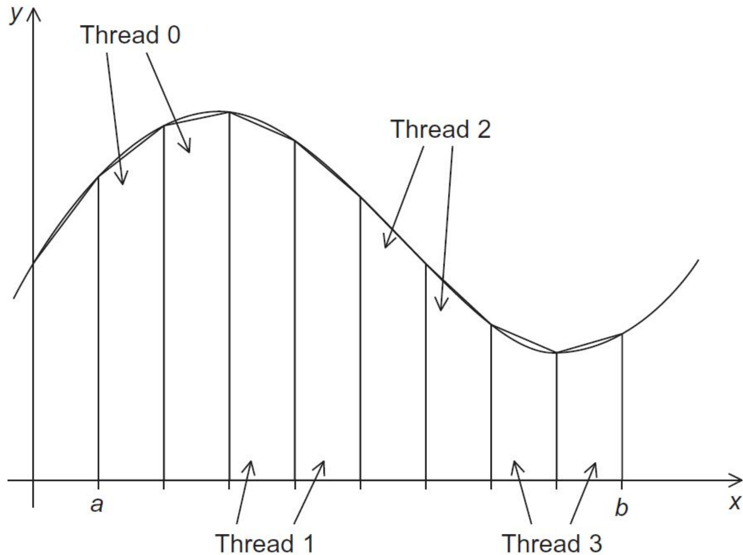
Uma primeira versão com OpenMP

UFABC

- Identificamos 2 tipos de tarefas
  - Computação das áreas dos trapézios individualmente
  - Somar a área de cada um dos trapézios
- Não há comunicação entre as tarefas da primeira tarefa, mas todas se comunicam com a segunda tarefa
- Assumimos que haverá bem mais trapézios do que núcleos de processamento
- Logo, agregamos tarefas através da junção de trapézios vizinhos e deixamos a cargo de uma thread (e portanto de um núcleo) o cálculo de cada bloco de trapézios.

Atribuindo trapézios às threads

UFABC



Condição de corrida

UFABC

Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

Resultados imprevisíveis eram obtidos de execuções quando threads tentavam executar simultaneamente o código:

```
1  global_result += my_result;
```

Exclusão mútua em OpenMP

UFABC

```
1 #pragma omp critical
```

- Apenas um thread pode executar o bloco logo após o pragma por vez
- Atenção é uma seção crítica global! Note que não há identificação alguma!

UFABC

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

int main(int argc, char* argv[]) {
   double  global_result = 0.0, a, b;
   int     n, thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}
```

UFABC

```c
void Trap(double a, double b, int n, double* glb_res_p){
   double  h, x, my_result, local_a, local_b;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   h = (b-a)/n;
   int local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (int i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *glb_res_p += my_result;
}
```

Escopo das variáveis

- Em linguagens de programação, o escopo de uma variável é definido pelas partes do programa nas quais as variáveis podem ser usadas.
- Em OpenMP, o escopo de uma variável se refere ao conjunto de threads que podem acessar a variável em um bloco paralelo.

Escopo em OpenMP

UFABC

- Uma variável que pode ser acessada por todas as threads de um *team* possui um escopo shared.
- Uma variável que é acessada por apenas uma thread tem escopo private.
- O escopo das variáveis declaradas antes de um bloco paralelo é shared.

A cláusula de redução

UFABC

- Nós fomos obrigados a declarar uma função com o protótipo:

```
void Trap(double a, double b, int n, double* glb_res_p);
```

Quando na verdade eu gostaria de ter feito:

```
void Trap(double a, double b, int n);
// E no meio do meu código

...
global_result = Trap(a, b, n);
...
```

Se usarmos uma função como:

```c
double Local_trap(double a, double b, int n);
```

Poderíamos escrever:

```c
global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
#   pragma omp critical
    global_result += Local_trap(a, b, n);
}
```

Mas isso é ruim! Estamos forçando a execução sequencial!

A cláusula de redução

UFABC

- Podemos resolver a execução sequencial declarando uma variável privada fora do bloco crítico dentro do bloco paralelo

```
1  global_result = 0.0;
2  # pragma omp parallel num_threads(thread_count)
3  {
4      double my_result = 0.0; //private
5      my_result += Local_trap(a, b, n);
6  #   pragma omp critical
7      global_result += my_result;
8  }
```

- Mas é quase igual o que tínhamos originalmente pra começo de conversa!
- Dá pra melhorar...

Operadores de redução

- Um operador de redução (*reduction operator*) é um operador binário (tal como adição e multiplicação).
- Uma redução é uma computação que repetidamente aplica o mesmo operador de redução a uma sequência de operandos visando obter um único resultado.
- Todos os resultados intermediários da operação devem ser armazenadas na mesma variável: a variável de redução.

Cláusula de redução

UFABC

- Uma cláusula de redução pode ser adicionada a uma diretiva paralela.

```
reduction(<operator>: <variable list>)
```

- Alguns operadores disponíveis: `+`, `*`, `&`, `|`, `^`, `&&`, `||`
- Assim, podemos ”consertar”o código anterior:

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count) \
    reduction(+: global_result)
global_result += Local_trap(a, b, n);
```

DIRETIVA **PARALLEL FOR**

- Dispara um time de threads para executar o bloco lógico que segue.
- O bloco lógico que segue a diretiva precisa ser um laço for.
- Além disso, a diretiva `parallel for` paraleliza o laço dividindo as iterações entre os threads.

Exemplo

UFABC

```
1      // Entradas: a, b e n
2      h = (b -a) / n;
3      approx = (f(a) + f(b))/2.0;
4   #  pragma omp parallel for num_threads(thread_count) \
5          reduction (+: approx)
6      for (i = 1; i <= n-1; i++) {
7          x_i = a + i * h;
8          approx += f (x_i);
9      }
10     approx = h * approx;
```

Tipos de sentenças for paralelizáveis pelo OpenMP

UFABC

```
                                     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
```

Cuidado...

UFABC

- A variável `index` precisa ser to tipo inteiro ou apontador (e.x., não pode ser `float`).
- As expressões `start`, `end`, and `incr` precisam ter tipos compatíveis. Por exemplo, se `index` é um apontador, então `incr` precisa ser do tipo inteiro.
- As expressões `start`, `end`, and `incr` não podem mudar durante a execução do laço.
- Durante a execução do laço, a variável `index` somente pode ser modificada pela "expressão de incrementar" dentro da sentença for.

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

note 2 threads

```
fibo[ 0 ] = fibo[ 1 ] = 1;
# pragma omp parallel for num_threads(2)
for (i = 2; i < n; i++)
    fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

Que tá pegando?

UFABC

- Compiladores OpenMP não checam dependências entre iterações do laço que está sendo paralelizado com a diretiva parallel for.
- Um laço cujos resultados de uma ou mais iterações dependem de outras iterações não pode, no geral, ser corretamente paralelizado por OpenMP.

Estimando π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \dots + (-1)^n \frac{1}{2n+1} + \dots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i ++) {
    sum += factor/(2*i +1);
    factor = -factor;
}
pi = 4.0 * sum;
```

Estimando π - OpenMP Versão 1

UFABC

```
1      double factor = 1.0;
2      double sum = 0.0;
3  # pragma omp parallel for num_threads(thread_count) \
4          reduction(+: sum)
5      for (i = 0; i < n; i ++) {
6          sum += factor/(2*i +1);
7          factor = -factor;
8      }
9      pi = 4.0 * sum;
```

- Nesta versão temos um *loop carried dependency*: o valor de `fator` depende da iteração anterior do laço.
- Não será paralelizado corretamente.

Estimando π - OpenMP Versão 2

UFABC

```
1     double sum = 0.0;
2  # pragma omp parallel for num_threads(thread_count) \
3        reduction(+: sum) private(factor)
4     for (i = 0; i < n; i ++) {
5         factor = (i % 2 == 0) ? 1 : -1;
6         sum += factor/(2*i +1);
7     }
8     pi = 4.0 * sum;
```

- Assim retiramos a dependência. Não era uma dependência real de dados, por isto conseguimos alterar o código.
- Note que a variável `factor` é definida como `private`, assim, as execuções das diferentes threads não interferem umas com as outras.

A cláusula `default`

UFABC

- Deixa o programador definir o escopo de cada variável em um bloco.

```
default (none)
```

- Com esta cláusula o compilador vai requerer que definamos o escopo de cada variável usada em um bloco e que foi declarada for a do bloco.
- Recebe também os valores `shared` (padrão) ou `private`

A cláusula `default`

```
1     double sum = 0.0;
2  # pragma omp parallel for num_threads(thread_count) \
3        default(none) reduction(+: sum) \
4        private(i, factor) shared(n)
5     for (i = 0; i < n; i ++) {
6         factor = (i % 2 == 0) ? 1 : -1;
7         sum += factor/(2*i +1);
8     }
9     pi = 4.0 * sum;
```