OpenMP

OpenMP

Open specifications for Multi-Processing via collaborative work between interested parties from hardware and software industry, government and academia.

http:\\www.openmp.org

OpenMP

OpenMP é uma interface de programação (API - *Application Program Interface*), portável, baseada no modelo de programação paralela de memória compartilhada para arquitetura de múltiplos processadores.

OpenMP

Portável

- ▷ A API é definida para C/C++ e Fortran
- ▷ Implementadas em várias plataformas incluindo Unix/Linux e Windows.

OpenMP

Componentes:

- Bibliotecas de funções
- Diretivas de compilação
- Variáveis de ambiente

OpenMP

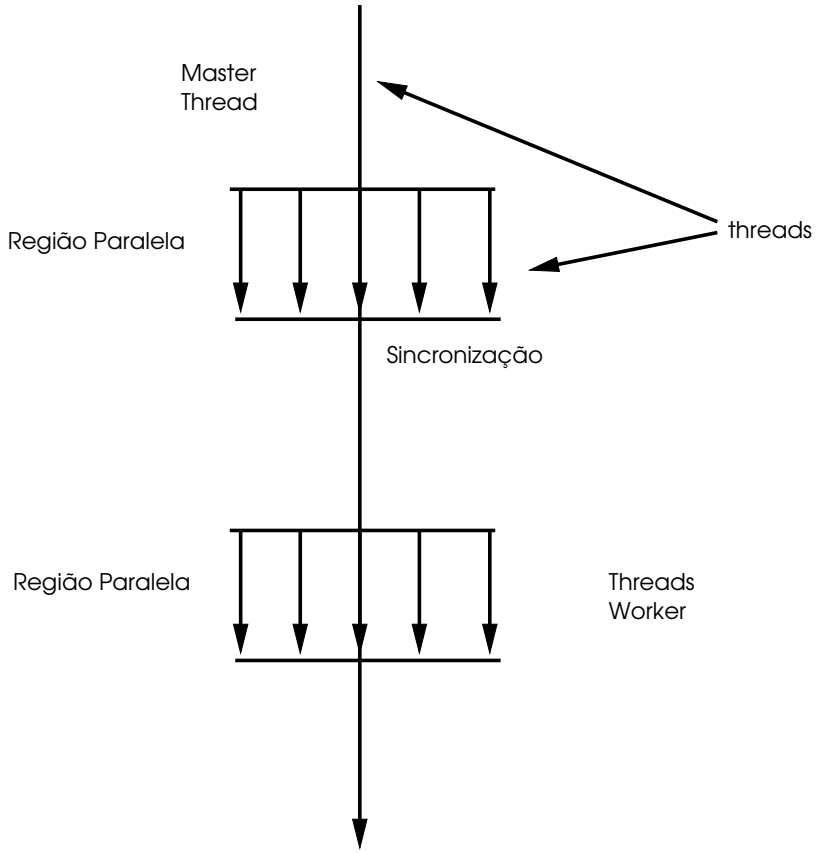
Paralelismo baseado em Threads, com memória compartilhada

Carga de trabalho dividida em várias threads

- ▷ Variáveis podem ser compartilhadas (**shared**) ou privadas (**private**)
- ▷ Comunicação é feita através das variáveis compartilhadas

Modelo de execução

Modelo Fork-Join



Região Paralela

Uma região paralela é um bloco de código que será executado por múltiplas threads

- ▷ Cada thread executa o mesmo código

```
código sequencial

#pragma omp parallel [clause[,] ...]
{

} /* fim da região paralela*/

código sequencial
```

Hello world

```
#include <omp.h>
main(){
   int nthreads, tid;
   #pragma omp parallel private(nthreads, tid)
   {
      tid = omp_get_thread_num();
      printf("Hello world from thread = %d\n", tid);

      if (tid == 0) {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
   } /* fim da região paralela*/
}
```

Hello world

Compilando com gcc no linux e OS X:

```
> gcc -fopenmp hello.c -o hello
> ./hello
```

Variáveis

Variáveis

Private

- Variáveis locais em cada thread

```
int nthreads, tid;
#pragma omp parallel private(nthreads, tid)
{
   tid = omp_get_thread_num();
   ...
}
```

Variáveis

Shared

- Variáveis compartilhada entre as threads

```
int nthreads, tid, a, b;
a = 10; b = 20;

#pragma omp parallel private(nthreads, tid) shared(a,b)
{
   tid = omp_get_thread_num();
   ...
}
```

Variáveis compartilhadas

Sincronização de processos

▷ Acesso concorrente aos dados podem resultar em inconsistência (race condition);

▷ Os processos concorrentes necessitarão de mecanismos de sincronização de processos.

Variáveis

First private

▷ Define uma lista de variáveis com o atributo PRIVATE, mas sendo inicializadas automaticamente, de acordo com o valor que possuíam no thread master antes de uma região paralela.

```
int nthreads, tid;
float pi = 3.1415;

#pragma omp parallel private(nthreads, tid) firstprivate(pi)
{
   tid = omp_get_thread_num();
   ...
}
```

For paralelo

- O for paralelo é um loop independente para cada thread
- O paralelismo ocorre na distribuição das iterações do loop para cada thread
- Loops infinitos e do while não são paralelizáveis no OpenMP

For paralelo

Estrutura básica

```
#pragma omp parallel
{
   #pragma omp for
   // for (...) {}
}
```

For paralelo

```
#include<omp.h>
#include<stdio.h>
int main(int argc, char *argv[]){
   int i, id;
   #pragma omp parallel private(id)
   {
      id = omp_get_thread_num();
      #pragma omp for
      for(i = 0; i < 8; i++)
         printf("%d - %d \n",id, i);
   }
   return 0;
}
```

For paralelo

```
> gcc ex02.c -o ex02 -fopenmp
> ./ex02
0 - 0
0 - 1
3 - 6
3 - 7
2 - 4
2 - 5
1 - 2
1 - 3
```

Cláusula Schedule

- O escalonamento das threads podem ser:
  - static, dynamic, guided e runtime

Cláusula Schedule

▷ static: as iterações são agrupadas (chunks), estaticamente distribuídos às threads

▷ dynamic: as iterações são agrupados em chunks, dinamicamente distribuídos, quando uma thread finaliza, recebe automaticamente o outro chunk.

Cláusula Schedule

▷ guided: semelhante ao dynamic, mas os chunks iniciam-se grandes e se tornam pequenos ao longo da execução

▷ runtime: a decisão é tomada em tempo de execução a partir da variável `OMP_SCHEDULE`

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main() {
   int i, chunk;
   float a[N], b[N], c[N];
   for (i=0; i < N; i++)
      a[i] = b[i] = i * 1.0;
   chunk = CHUNKSIZE;
   #pragma omp parallel shared(a,b,c,chunk) private(i) {
      #pragma omp for schedule(dynamic,chunk) nowait
      for (i=0; i < N; i++)
         c[i] = a[i] + b[i];
   } // fim da região paralela
}
```

Diretivas Sections

- Cria seções paralelas
- Cada seção será executada em uma thread diferente

Diretivas Sections

```
#include<omp.h>

main(){
   int x;
   #pragma omp sections {

       #pragma omp section
       { foo(x); }

       #pragma omp section
       { bar(x); }
   }

}
```

Exercício: Integração numérica

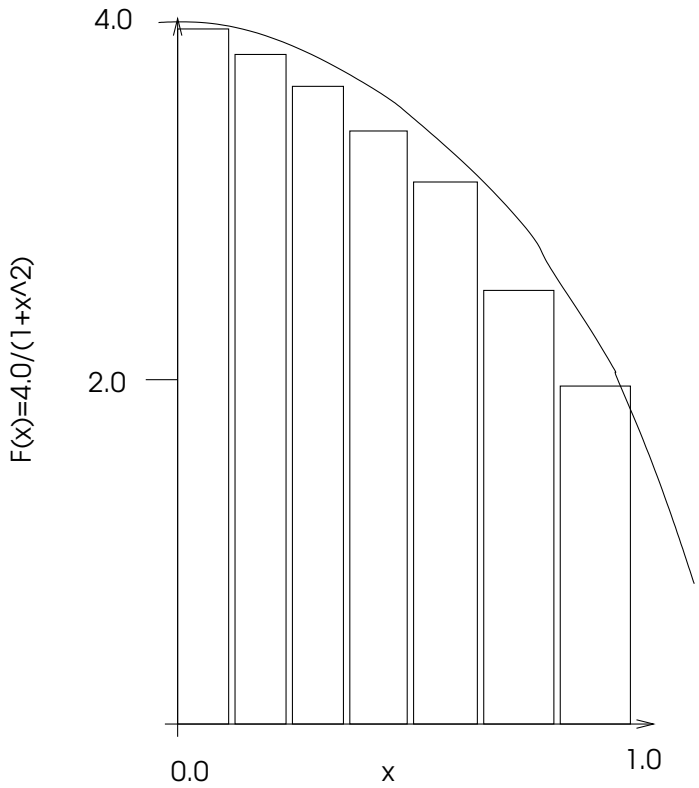
Matematicamente, sabemos que

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

Podemos aproximar a integral como a soma de retângulos

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \Pi$$

Onde cada retangulo tem largura Δx e altura $F(x_i)$ no meio do intervalo i.



$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \Pi$$

Exercício: Integração numérica

Programa serial Pi

```
static long num_steps = 100000;
double step;
void main ()
{  int i; double x, pi, sum = 0.0;
   step = 1.0 / (double) num_steps;
   for (i = 0; i < num_steps; i++){
      x = (i + 0.5) * step;
      sum = sum + 4.0 / (1.0 + x * x);
   }
   pi = step * sum;
}
```

Exercício: Integração numérica

Escreva uma versão paralela para o problema da integração numérica

Constructs de Sincronização

Constructs de Sincronização

A sincronização é usado para impor restrição de ordem e proteger acesso aos dados compartilhados

Tipos de Sincronização:

- critical, atomic, barrier, ordered

Sincronização: critical

Exclusão mútua: apenas uma thread por vez pode entrar na região crítica

Sincronização: critical

```
int cnt = 0;
int f = 7;
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i < 20; i++)
   {
      if (b[i] == 0) {
         #pragma omp critical
         cnt ++;
      }
      a[i] = b[i] + f * (i + 1);
   }
}
```

Sincronização: atomic

Exclusão mútua: apenas aplicada para atualizar dado em memória

Sincronização: atomic

```
int cnt = 0;
int f = 7;
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i < 20; i++)
   {
      if (b[i] == 0) {
         #pragma omp atomic
         cnt ++;
      }
      a[i] = b[i] + f * (i + 1);
   }
}
```

Sincronização: Barrier

Quando esta diretiva é alcançada por uma thread, este espera até os restantes cheguem ao mesmo ponto.

Exercícios

1) Escreva um programa que compute a multiplicação entre matrizes e vetores $b = Ax$. Utilize as diretivas do OpenMP para execução paralela.

2) Escreva um programa com OpenMP que some os elementos de um vetor de números inteiros. Utilize um for paralelo.

3) Escreva um programa com OpenMP que some os elementos de um vetor de números inteiros. Utilize um a diretiva reduction.

Exercícios

4) Escreva um programa com OpenMP que encontre o maior elemento de um vetor de números inteiros. Utilize um a diretiva reduction.

5) Escreva um programa com OpenMP que encontre o maior elemento de um vetor de números inteiros. Neste exercício não é permitido usar reduction e nem for paralelo, utilize apenas a região paralela.

Exercícios

6) Suponha que você tenha o seguinte código serial:

```
for (i = 0; i < N; i++)
   a[i] = b[i] + c[i];
for (i = 0; i < N; i++)
   d[i] = a[i] + b[i];
```

Escreva a versão paralela deste trecho utilizando as diretivas do OpenMP

Fim