Programação em Computação Paralela e Distribuída

Laboratório, ERBASE 2009

OpenMP

Domingo Giménez

Universidad de Murcia, Spain

Grupo de Computación Paralela

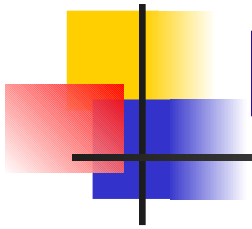
http://dis.um.es/~domingo

Murilo Boratto

UFBa

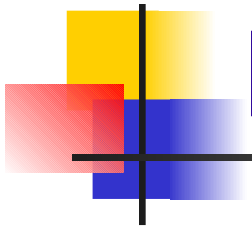
Leandro Coelho

UNEB



Noções básicas

- Ferramenta de programação para memória commum
- Modelo de programação *fork-join*, com geração de múltiplas threads
- Inicialmente executa-se um thread até que aparece o primeiro construtor paralelo, se criam threads escravos e o que os põe em marcha é o maestro
- Ao final do construtor se sincronizan os threads e continua a execução o maestro

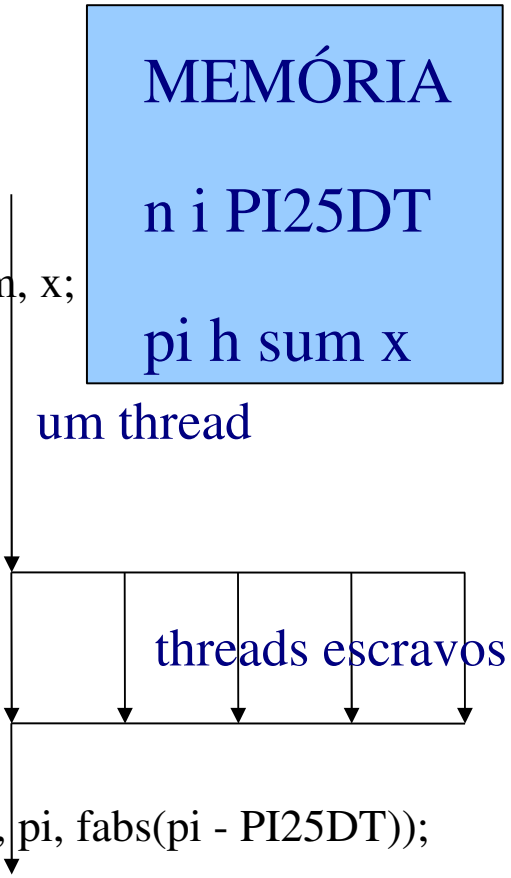


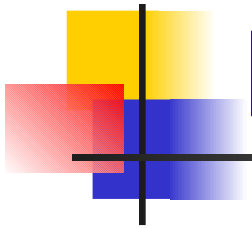
Noções básicas

- Contém:
  - **Construtores**: indicam como distribuir o trabalho, gestionar threads, sincronizar...
  - **Funções**: para estabelecer, obter e comprovar valores
  - **Variaveis de ambiente**: indicam a forma da execução

Exemplo

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
double f(double x) {    return sqrt(1 - pow(x, 2.)); }
int main(int argc, char *argv[]) {
    int  n, i;
    double PI25DT = 3.14159265358979323846264338, pi, h, sum, x;
    printf("Numero de intervalos a usar: ");
    scanf("%d",&n);
    h   = 1.0 / (double) n;
    sum = 0.0;
    #pragma omp parallel for reduction(+:pi) private(x, i)
    for (i = 1; i <= n; i ++) {
        x = h * ((double)i - 0.5);
        pi += f(x); }
    pi = 4. * h * pi;
    printf("\npi es aproximadamente %.16f, el error es %.16f\n", pi, fabs(pi - PI25DT));
}
```





Exemplo

compilação:

gcc -O3 -o codigo3-16 codigo3-16.c -fopenmp

icc -O3 -o codigo3-16 codigo3-16.c -openmp

execução:

$> export OMP_NUM_THREADS=1

$> time ./codigo3-16 <in10000000

Numero de intervalos a usar:

pi es ...

real 0m0.240s

user 0m0.240s

$> export OMP_NUM_THREADS=4

$> time ./codigo3-16 <in10000000

Numero de intervalos a usar:

pi es ...

real 0m0.094s

user 0m0.240s

Construtores (*pragmas*)

Formato:

API C

#pragma omp nome-directiva [cláusulas]

API Fortran

!$ omp nome-directiva [cláusulas]

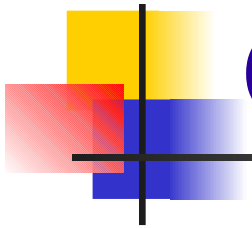
também: c$omp ou *$omp

Construtor parallel

#pragma omp parallel [cláusulas]

bloco

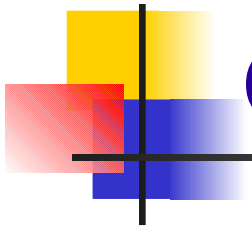
- Cria-se um grupo de threads e o que os põe em marcha actua de maestro
- Se aparece a cláusula **if** avalia-se sua expressão, e se dá valor diferente de zero criam-se os threads, se é zero faz-se em secuencial
- O número de threads se obtém por variables de ambiente ou funções da livraria
- Há barreira implícita ao final da região



Construtor parallel

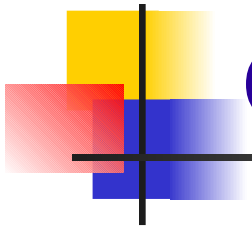
- Quando dentro de uma região há outro construtor paralelo, cada escravo criaria outro grupo de threads escravos dos que seria o maestro
- Há uma série de cláusulas (**private**, **firstprivate**, **default**, **shared**, **copyin** e **reduction**) para indicar a forma em que se acede às variables

Cláusulas de alcance de dados

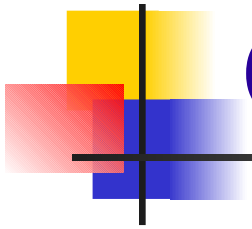


- private(lista): as variables da lista são privadas aos threads, não se inicializan dantes de entrar e não se guarda seu valor ao sair
- firstprivate(lista): as variables da lista são privadas aos threads, se inicializan ao entrar com o valor que tivesse a variable correspondente
- lastprivate(lista): as variables da lista são privadas aos threads, ao sair combinam com o valor da última iteración ou secção

Cláusulas de alcance de dados



- shared(lista): as variables da lista são compartilhadas por todos os threads
- default(shared|none): indica como são as variables por defeito
- reduction(operador:lista): as variables da lista se obtêm pela aplicação do operador. Há uma série de possíveis operadores
- copyin(lista): para atribuir o valor da variable no master as variables locais aos threads

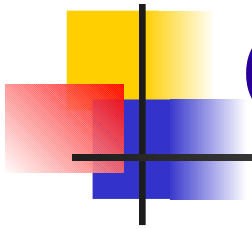


Construtor for

#pragma omp for [cláusulas]

bloco for

- As repetiçãos do bucle executam-se em paralelo por threads que já existem (construtor parallel)
- A inicialização é uma asignación
- O incremento é uma soma ou uma resta
- A avaliação da condição é a comparação de uma variable inteira sem signo com um valor, com um comparador maior ou menor (pode incluir igual)

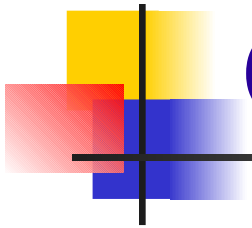


Construtor for

#pragma omp for [cláusulas]

bloco for

- Há barreira ao final, ou pode-se utilizar a claúsula **nowait** para que não esperem os threads
- Cláusulas de partição de variables: **private**, **firstprivate**, **lastprivate** e **reduction**



Construtor for

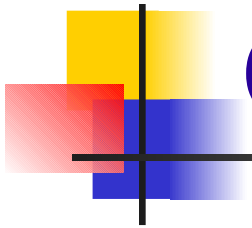
schedule: indica a forma como se dividem os repetiçãos do for entre os threads

- **schedule**(**static**,tamanho): as iteraciones dividem-se com tamanho indicado, e a asignación faz-se estaticamente aos threads. Se não se indica o tamanho se divide a partes iguais entre os thread
- **schedule**(**dynamic**,tamanho): dividem-se com o tamanho indicado, e atribuem-se aos threads quando acabam seu trabalho
- **schedule**(**guided**,tamanho): atribuem-se dinamicamente aos threads com tamanhos decrecientes
- **schedule**(**runtime**): deixa a decisão para o tempo de execução, e obtêm-se da variable do ambiente OMP_SCHEDULE

Construtor sections

- A cada secção executa-se por um thread.
- Há barreira ao final se não se utiliza a cláusula **nowait**
- Cláusulas de partição de variables: **private**, **firstprivate**, **lastprivate** e **reduction**

```
#pragma omp sections [cláusulas]
{
   [#pragma omp section]
      bloque
   [#pragma omp section
      bloque
      ...
   ]
}
```



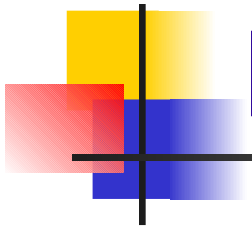
Construtores combinados

#pragma omp parallel for [cláusulas]

#pragma omp parallel sections [cláusulas]

Formas abreviadas do construtor **parallel** seguido por um único construtor **for** ou **sections**

admite suas cláusulas mas não **nowait**



Exemplos

os códigos **3-13** e **3-14** são exemplos de **for** e **sections**

compilar, executar, entender e tirar as dúvidas

o código **3-15** é um exemplo da multiplicación matriz por vector

modificá-lo para fazer uma multiplicación matriz por matriz

e comprovar o ganho de tempo usando paralelismo

Constructores de sincronização

#pragma omp single [cláusulas]

- O bloco executa-se por um único thread na equipa
- Há barreira ao final se não se utiliza a cláusula **nowait**
- Cláusulas de partição de variables: **private** e **firstprivate**

#pragma omp master

- O bloco executa-o o thread mestre
- Não há sincronización ao entrar nem sair

Constructores de sincronização

#pragma omp critical [nome]

- Assegura exclusão mútua na execução do bloco
- O nome pode-se usar para identificar secções críticas diferentes

#pragma omp atomic

expressão

- A expressão deve ser *x binop=exp*, *x*++, ++*x*, *x*-- ou --*x*, onde *x* é uma expressão com valor escalar, e *binop* é um operador binario
- Assegura a execução da expressão de forma atómica

Constructores de sincronização

#pragma omp barrier

- Sincroniza todos os threads na equipa

#pragma omp flush [lista]

- Assegura que as variáveis ficam actualizadas para todos os threads
- Se não há lista se actualizam todas as variáveis compartilhadas
- Faz-se flush implícito ao acabar **barrier**, ao entrar ou sair de **critical** ou **ordered**, ao sair de **parallel**, **for**, **sections** ou **single**

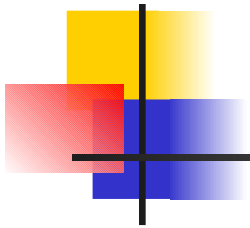
Constructores de sincronização

#pragma omp ordered

- O bloco executa-se no ordem em que executar-se-ia em secuencial

#pragma omp threadprivate (lista)

- Usado para declarar variables privadas aos threads



Funções

- incluir a biblioteca:

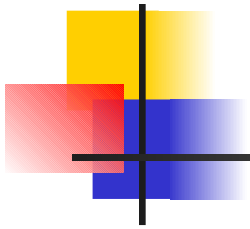
```
#include <omp.h>
```

```
void omp_set_num_threads(int num_threads);
```

- Põe o número de threads a usar na seguinte região paralela

```
int omp_get_num_threads(void);
```

- Obtém o número de threads que se estão a usar numa região paralela



Funções

int omp_get_max_threads(void);

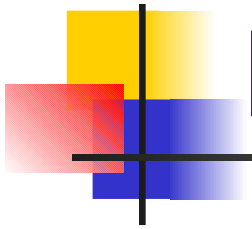
- Obtém a máxima quantidade possível de threads

int omp_get_thread_num(void);

- Devolve o número do thread.

int omp_get_num_procs(void);

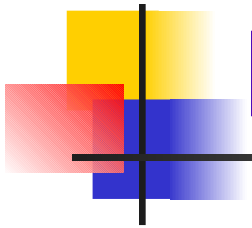
- Devolve o máximo número de processadores que se podem usar



Funções

int omp_in_parallel(void);

- Devolve um valor diferente de zero se executa-se dentro de uma região paralela
- Ajustamento dinâmico: omp_set_dynamic e omp_get_dynamic
- Paralelismo anidado: omp_set_nested e omp_get_nested



Funções

- Manejamento de chaves:

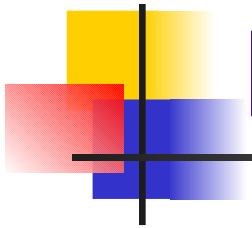
void omp_init_lock(omp_lock_t *lock); inicializa a chave como não bloqueada

void omp_init_destroy(omp_lock_t *lock); destrói a chave

void omp_set_lock(omp_lock_t *lock); pede a chave

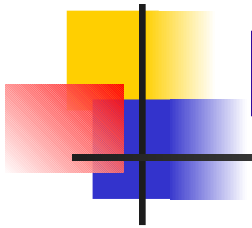
void omp_unset_lock(omp_lock_t *lock); solta a chave

int omp_teste_lock(omp_lock_t *lock); pede a chave mas não se bloqueia



Funções

- OMP_SCHEDULE: indica o tipo de scheduling para **for** e **parallel for**
- OMP_NUM_THREADS: põe o número de threads a usar, ainda que pode-se mudar com a função de livraria
- OMP_DYNAMIC: autoriza ou desautoriza o ajuste dinâmico do número de threads
- OMP_NESTED: autoriza ou desautoriza o anidamiento. Por defeito não está autorizado



Exemplos

os códigos **3-11** e **3-12** são exemplos do típico programa "Hello world“

contêm várias das funções que vimos

Provar com outras funções

Provar com alguns dos exemplos o funcionamento das variáveis do ambiente