==========

Paralelização em OpenMP

Identifique os loops “pesados”;
Distribua-os:

Versão sequencial

```
double res[10000];
for (i=0 ; i < 10000 ; i++)
       calculo_pesado(&res[i]);
```

Versão paralela

```
double res[10000];
#pragma omp parallel for
  for (i=0 ; i < 10000 ; i++)
       calculo_pesado(&res[i]);
```

OMP for

```
for(int x=0; x < width; x++)
{
       for(int y=0; y < height; y++)
       {
              finalImage[x][y]=RenderPixel(x,y, &sceneData);
       }
}
```

```
#pragma omp parallel for
for(int x=0; x < width; x++)
{
       for(int y=0; y < height; y++)
       {
              finalImage[x][y]=RenderPixel(x,y, &sceneData);
       }
}
```

Esta diretiva com a cláusula omp for diz ao compilador para paralelizar automaticamente o loop for com OpenMP. Se um usuário estiver usando um processador quad-core, o desempenho de seu programa pode ser esperado para ser aumentado em 300% com a adição de apenas uma linha de código, `#pragma omp parallel for`

==========

5 Categorias de Diretivas

Define regiões paralelas: `parallel`
Compartilhamento dos dados: `shared, private,. . .`
Distribuição de trabalho: `for`
Sincronizações: `atomic, critical, barrier,. . .`
Funções em runtime: `omp_set_num_threads(), omp_set_lock(),. . .`
Variáveis de ambiente: `OMP_SCHEDULE, OMP_NUM_THREADS, . . .` (são executadas no prompt de commando)

==========

Definição de região paralela

Criação de threads

```
OMP PARALLEL double A[10000];
omp_set_num_threads(4);
#pragma omp parallel
  { int th_id = omp_get_thread_num();
      calculo_pesado(th_id, A);
  }
  printf("Terminado");
```

Observações:

- ✓ o início da execução das threads é sinalizado;
- ✓ as threads são sincronizadas;
- ✓ o vetor A é compartilhado;
- ✓ Usou-se funções OpenMP, além das diretivas.

==========

Compartilhamento dos dados

Variáveis compartilhadas:
- variáveis estáticas;
- variáveis globais.

Variáveis privadas a cada thread:
- variáveis locais a um bloco;
- variáveis alocadas na pilha de um procedimento chamado por uma seção paralela.

==========

Alterar o compartilhamento de dados

Existem cláusulas para especificar, variável por variável, o que compartilhar.

As cláusulas completam as diretivas `parallel, sections, for`.

`shared( toto )` especifica que a variável 'toto' é compartilhada;

`private( titi )` especifica que a variável 'titi' é privada: cria uma cópia privada em cada thread;

`default(private)` e `default(shared)` existem também.

==========

Distribuição de trabalho OMP FOR

`for` pode ser anotado para ser distribuído.

```
#define N 1000;
int i;
#pragma omp parallel
  #pragma omp for
   for (i=0 ; i < 10000 ; i++)
     {
       calculo_pesado();
     }
     printf("Terminado");
```

- ✓ As iterações são distribuídas entre as threads.
- ✓ Tem uma barreira implícita no final do laço.
- ✓ `omp for` pode ser complementado pela diretiva `schedule` para especificar como fazer a distribuição da carga do `for (i=0 ; i < 10000 ; i++)`.

==========

omp for schedule

Distribuição das iterações por bloco entre as threads.

`for schedule(static [,chunk])` - distribuição estática das iterações por bloco (de tamanho '`chunk`') entre as threads.

`for schedule(dynamic [,chunk])` - distribuição "dinâmica" (cíclica) das iterações por bloco entre as threads.

`for schedule(guided [,chunk])` - distribuição estática das iterações por bloco entre as threads; o tamanho do bloco diminui a medida que o cálculo anda;.

`for schedule(runtime)` : o escalonamento dos loops é deixado para ser determinado na execução (OMP_SCHEDULE).

Veja o exemplo de uso do SCHEDULE em : **Multiplicação de matrizes em** Apostila Introdução ao OpenMP (AULA)

==========

OMP PARALLEL

```
#pragma omp parallel
  {
      int th_id = omp_get_thread_num() ;
      int nb_th = omp_get_num_threads();
      int inicio = th_id * 10000 / nb_th;
      int fim = (th_id+1)*10000 / nb_th;
      for (i=inicio ;  i < fim ; i++)
          a[i] := a[i]+b[i];
      printf("Terminado");
  }
```

OMP FOR

```
#pragma omp parallel
  #pragma omp for schedule(static)
   for (i=0 ;  i < 10000 ; i++)
          a[i] := a[i]+b[i];
   printf("Terminado");
```

==========

PRIVATE

```
int soma = 0 ;
#pragma omp parallel for schedule(static) private(soma)
  for (i=0 ; I < 10000 ; i++)
        soma += a[i];
        printf("Terminado – soma = %d", soma);
```

2 problemas: **inicialização** + **valor final**!

==========

Exemplos comparativos

PRIVATE

```
int soma = 0;
#pragma omp parallel for schedule(static)  private(soma)
     for (i=0 ; I < 10000 ; i++)
        soma += a[i];
        printf("Terminado – soma = %d", soma);
```

Aqui, dois problemas: i**nicialização** e **valor final** !

LASTPRIVATE

```
int soma = 0 ;
#pragma omp parallel for schedule(static)
```

```
  #pragma omp firstprivate(soma) lastprivate(soma)
       for (i=0 ; I < 10000 ; i++)
          soma += a[i];
       printf("Terminado");
```

Resolveu o problema da **inicialização** e do **valor final** !

==========

Redução

Mais uma cláusula: `reduction(op : list);`
usada para operações tipo "all-to-one":

- exemplo: op = '+'
- cada thread terá uma cópia da(s) variável(is) definidas em 'list' com a devida inicialização;
- ela efetuará a soma local com sua cópia;
- ao sair da seção paralela, as somas locais serão automaticamente adicionadas na variável

Exemplo de redução

Redução

```
#include <omp.h>
#define NUM_THREADS 4
void main( )
{
  int i, tmp, res = 0;
  #pragma omp  parallel  for reduction(+:res)  private(tmp)
     for (i=0 ; i< 10000 ; i++)
     {
       tmp = Calculo( );
       res += tmp ;
     }
     printf("O resultado vale %d´´, res) ; } Obs: os índices de
     laços sempre são privados.
}
```

Obs: Os índices de loops sempre são privados.

==========

Distribuição de trabalho (2)

Pode-se usar `omp section` **quando não se usam loops:**

OMP SECTIONS

```
#pragma omp parallel
```

```
  #pragma omp sections
  {
    Calculo1( );
  #pragma omp section
    Calculo2( );
  #pragma omp section
    Calculo3( );
}
```

As seções são distribuídas entre as threads.

==========

Sincronizações

Existem várias instruções para **sincronizar os acessos à memória compartilhada**:

Seção crítica

- #pragma omp critical {. . . }
- Apenas uma thread pode executar a seção crítica num dado momento.

Atomicidade:

- versão "light" da seção crítica.
- funciona apenas para a próxima instrução de acesso à memória.

Barreira:

- #pragma omp barrier
- barreiras implícitas nos fins das seções paralelas! master e ordered.

Master e Ordered

`#pragma omp ordered`: impõe a ordem de execução sequencial.

`#pragma omp master`: apenas a thread master executa o bloco.

==========

Funções de biblioteca para o run-time

- Não são diretivas!
- Funções para setar/consultar parâmetros durante a execucão:
  - **número de threads**: `omp_set_num_threads`, `omp_get_num_threads`;
  - **número de processadores**: `omp_num_procs( )`.
  - **locks**: existe um tipo `omp_lock_t` e primitivas: `omp_init_lock( )`, `omp_set_lock( )`, etc...

==========

Alternativa: variáveis de ambiente

Não são diretivas — não aparecem no código! São configuradas na linha de comando.

Variáveis para setar/consultar parâmetros antes da execução:

- número de threads: `OMP_NUM_THREADS`

- tipo de escalonamento (runtime): `OMP_SCHEDULE`

==========

Diversos

O número de threads especificado pelo usuário é indicativo.

- O *runtime* pode, na verdade, mapear as tarefas para um número menor de threads.
- Isso pode tipicamente acontecer em laços aninhados.

==========

Bibliografia

- OpenMP home-page: http://www.openmp.org/presentations
- Parallel Programming in OpenMP. R. Chandra et al., Morgan Kaufmann, 2001.

==========