

# OpenMP

Nicolas Maillard  
nicolas@inf.ufrgs.br

Instituto de Informática  
Universidade Federal do Rio Grande do Sul

2006

# Da arquitetura à Programação

- o curso apresentou a tecnologia usada na arquitetura Multicore;
  - nível do **Hardware**.
- foi apresentada e discutida a noção de *Thread*;
  - nível do **Sistema Operacional**.
- agora vem a parte de programação de aplicativos.
  - nível do **Software**.

# O problema da programação paralela

- É um problema antigo. . .
  - Começou com sistemas distribuídos nos anos 70.
- É um problema complicado.
  - Heterogeneidade das máquinas;
  - Dinamicidade das máquinas;
  - Memória Compartilhada ou Distribuída?
- Não existe nem um MODELO de programa paralelo
  - Modelo sequencial: modelo de Von Neumann.
  - ModeloS paraleloS:
    - Processos comunicantes;
    - Memória compartilhada;
    - Objetos Distribuídos;
    - Paralelismo funcional;
    - Paralelismo de dados;
    - Compilação.

- Single Program Multiple Data
  - Um programa só é executado por todos os processadores.
- Norma de especificação alto-nível para máquina **com memória compartilhada**.
  - = camada mais abstrata em cima de *threads*.
- Parallelismo de **laços**.
  - O usuário explicita o paralelismo dos laços.
  - Parallelismo tipo “Fork-Join”.
- Uso de anotações (diretivas) para acrescentar uma linguagem usual.
  - **pragma openmp** Whatever you want( )
  - Uma precompilação permite tornar o código seqüencial.
  - Funciona com C, C++, Fortran.

# Disponibilidade de OpenMP

- Consórcio de fabricantes de HW/SW:
  - (Compaq), HP, IBM, Intel, SGI, SUN.
  - PGI, KAI, PSR, Absoft
  - DOE, NAG, ASCI, ...
- Compiladores *OpenMP compliant*
  - PGI
  - Intel
  - gcc 4.2.0 (SVN)

# Paralelização em OpenMP

- Identifique seus laços “pesados”;
- Distribua-os:

## Versão seqüencial

```
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]);
```

# Comunicação entre threads

- Via a memória **compartilhada**.
  - Possibilidade de definir as variáveis compartilhadas.
- Necessidade de **sincronizar** os acessos.
  - isso implica em **sobrecusto**;
  - open-MP esconde isso ao programador;
  - necessidade de projetar o algoritmo (distribuição dos dados, volume de acessos remotos. . . ).

# 5 categorias de diretivas

- Regiões paralelas;
  - `omp parallel`
- Distribuição de trabalho;
  - `omp for`
- Compartilhamento dos dados;
  - `omp shared, private, ...`
- Sincronizações;
  - `omp atomic, critical, barrier, ...`
- *runtime* funções e variáveis de ambiente.
  - `omp_set_num_threads()`, `omp_set_lock()`, ...
  - `OMP_SCHEDULE`, `OMP_NUM_THREADS`, ...

# Definição de regiões paralelas

- 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");
```

- a chave abrindo sinaliza o início da execução das threads;
- a chave fechando sincroniza as threads;
- A é compartilhado.
- Obs: usou-se funções OPen-MP além das diretivas.

- **for** pode ser anotado para ser distribuído.

## OMP FOR

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

- As iterações são distribuídas entre as threads.
- Tem uma barreira no final do laço.
- omp for é complementado pela diretiva **schedule** para especificar como fazer a distribuição.

- **omp for schedule(static [, step])** : distribuição estática das iterações por bloco (de tamanho 'step') entre as threads;
- **omp for schedule(dynamic [, step])** : distribuição “dinâmica” (cíclica) das iterações por bloco entre as threads;
- **omp for schedule(guided [, step])** : distribuição estática das iterações por bloco entre as threads; o tamanho do bloco diminui a medida que o cálculo anda;
- **omp for schedule(runtime)** : o escalonamento dos laços é deixado para ser determinado à execução (OMP\_SCHEDULE).

# Exemplo comparativo

## 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");
```

# Distribuição de trabalho (2)

- Pode-se usar **omp section** quando não se usam laços:

## OMP SECTIONS

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

- As seções são distribuídas entre as threads.

# 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 **omp parallel**, **omp sections** ou **omp 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.

## 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!

## PRIVATE

```
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 fim!

# Exemplos

## 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!

## PRIVATE

```
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 fim!

- 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 global.

# 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.

Existe várias instruções para sincronizar os acessos à memória compartilhada:

- **seção crítica**
  - `#pragma omp critical {...}`
  - Apenas uma thread pode executar a SC num dado momento.
- **atomicidade**
  - versão “light” da SC.
  - funciona apenas para a próxima instrução de acesso à memória.
- **barreira:**
  - `#pragma omp barrier`
  - barreiras implícitas nos fins das seções paralelas!
- **master e ordered.**

# Funções de biblioteca para o run-time

- Não são diretivas!
- Funções para setar/consultar parâmetros durante a execução:
  - 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 — mas não aparecem no código!
- Variáveis para setar/consultar parâmetros antes a execução:
  - número de threads: `OMP_NUM_THREADS`,
  - tipo de escalonamento (runtime): `OMP_SCHEDULE`,

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