Programação Paralela com OpenMP (Parte 2)

Ricardo Rocha e Eduardo R. B. Marques

Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Porto

Computação Paralela 2018/2019

OpenMP – Parte 2

Conteúdo:

- Ciclos paralelos (`omp for`) – desempenho e correcção:
  - Escalonamento de trabalho com a cláusula `schedule`.
  - Análise de dependências entre iterações de ciclos ("loop-carried dependencies") e reconversão de código.
- Encadeamento de código:
  - Directivas orfãs.
  - Encadeamento de directivas.
  - Âmbitos de variáveis em chamadas a funções.

omp for – a cláusula schedule

```
#pragma omp for schedule(type [,c])
```

Para um ciclo paralelo **omp for**, a cláusula **schedule** define a forma de divisão de trabalho / escalonamento de iterações entre as várias threads:

- O parâmetro **type** define o tipo de escalonamento.
- O parâmetro opcional **c** ("chunk size") define a unidade de trabalho para alocação em termos do número de iterações contíguas a executar por uma só thread.

omp for – **escalonamento** static

```
#pragma omp for schedule(static [,c])
```

- **static,c**: as unidades de trabalho são formadas por **c** iterações contíguas, alocadas estaticamente a cada thread de forma alternada.
- Por omissão **c = n / t** onde **n** é o número de iterações e **t** o número de threads.
- Na omissão da cláusula **schedule**, tipicamente o ambiente de execução emprega **schedule(static)** por omissão.

Q: O que esperar do seguinte fragmento para **c = 4, 3, 2, 1** ?

```
#pragma omp parallel for schedule(static,c) num_threads(3) ordered
for (int i=0; i < 12; i++) {
  #pragma omp ordered
  printf("[%d>%d]", i, omp_get_thread_num());
}
```

omp for – escalonamento static (2)

```
#pragma omp parallel for schedule(static,c) num_threads(3) ordered
for (int i=0; i < 12; i++) {
  #pragma omp ordered
  printf("[%d>%d]", i, omp_get_thread_num());
}
```

```
$./test_static.bin
c=4
 [0>0] [1>0] [2>0] [3>0] [4>1] [5>1] [6>1] [7>1] [8>2] [9>2] [10>2] [11>2]
c=3
 [0>0] [1>0] [2>0] [3>1] [4>1] [5>1] [6>2] [7>2] [8>2] [9>0] [10>0] [11>0]
c=2
 [0>0] [1>0] [2>1] [3>1] [4>2] [5>2] [6>0] [7>0] [8>1] [9>1] [10>2] [11>2]
c=1
 [0>0] [1>1] [2>2] [3>0] [4>1] [5>2] [6>0] [7>1] [8>2] [9>0] [10>1] [11>2]
```

omp for – **escalonamento** dynamic

```
#pragma omp for schedule(dynamic [,c])
```

- `dynamic, C`: unidades de trabalho formadas por `C` iterações contíguas, alocadas **dinamicamente** a cada thread à medida que estes terminam de executar o conjunto anterior.
- Por omissão `C = 1`.
- O esquema é adequado a computações irregulares, onde o esforço computacional por iteração pode ser altamente variável.
- Por outro lado, o ambiente de execução tem de gerir a alocação dinamicamente o que acarreta um custo computacional extra no face ao esquema `static`

`static` vs `dynamic` – exemplo

- Tomemos como exemplo a avaliação da função de Ackermann, definida para inteiros não-negativos m e n

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(m-1,1) & \text{se } m > 0 \wedge n = 0 \\ A(m-1,A(m,n-1)) & \text{se } m > 0 \wedge n > 0 \end{cases}$$

- Código "naive" (sem uso de "memoization" por exemplo):

```
uint64_t ack(uint64_t m, uint64_t n, uint64_t* c) {
  (*c)++; // call counter
  if (m == 0) return n + 1;
  if (n == 0) return ack(m - 1, 1, c);
  return ack(m - 1, ack(m, n - 1, c), c);
}
```

- Valor (e tempo de computação) da função cresce rapidamente mesmo para valores pequenos de m e n, ex: $A(4,3) = 2^{2^{65536}} - 3$.

`static` vs `dynamic` – exemplo (2)

- Experiência: ciclo `omp for` para cálculo em I iterações de $A(3, vn[i])$ $i \in 0, ..., I-1$ onde $vn[i]$ é um número pseudo-aleatório entre 0 e 13.
- Cálculo de $A(3, vn[i])$ pode ter tempo de execução bastante variável, dependendo do valor de $vn[i]$.
- Vamos comparar desempenho de

```
#pragma omp for schedule(static,1) ...
for (int i=0; i < iterations; i++) ... // compute A(3, vn[i])
```

versus

```
#pragma omp for schedule(dynamic,1) ...
for (int i=0; i < iterations; i++) ... // compute A(3, vn[i])
```

`static` vs `dynamic` – exemplo (3)

- Escalonamento `static` – nº igual de iterações p/thread, carga de trabalho des-balanceada!

```
$ ./ackermann.bin s 4 100
schedule: s threads: 4 iterations: 100
...
t: 0 iterations: 25 time:  2.475 s calls:  876654009
t: 1 iterations: 25 time: 13.685 s calls: 4706110486
t: 2 iterations: 25 time: 19.324 s calls: 6571480150
t: 3 iterations: 25 time: 11.558 s calls: 3817663575
total time: 19.326 seconds, 15971908220 calls
```

- Escalonamento `dynamic` – nº variável de iterações p/thread, carga mais balanceada!

```
$ ./ackermann.bin d 4 100
schedule: d threads: 4 iterations: 100
...
t: 0 iterations: 29 time: 13.043 s calls: 4429196999
t: 1 iterations: 20 time: 11.451 s calls: 3773131333
t: 2 iterations: 18 time: 11.455 s calls: 3694777738
t: 3 iterations: 33 time: 12.143 s calls: 4074802150
total time: 13.044 seconds, 15971908220 calls
```

omp for – **escalonamento** guided

```
#pragma omp for schedule(guided [,c])
```

- **guided, c**: escalonamento dinâmico em conjuntos de iterações proporcionais ao número de threads e número de iterações por executar, com um mínimo de **c** iterações p/alocação a thread. Por omissão **c=1**.
- Iterações alocadas por thread decrescem exponencialmente de uma forma dependente da implementação, até ao valor mínimo de **c**. O esquema é portanto adaptativo: os conjuntos de iterações começam por ser maiores no início e menores no fim. Tenta-se diminuir assim a probabilidade de um thread ficar sem trabalho prematuramente / des-balanceamento de carga.
- Lógica interna de alocação pode ser ter também menos onerosa do que **dynamic**, porque requer menos passos.

omp for – **escalonamento** runtime

```
#pragma omp for schedule(runtime)
```

- **runtime**: o escalonamento é definido em tempo de execução em função do valor da variável de ambiente OMP_SCHEDULE, definida como (dynamic|guided|dynamic)[,c]
- Exemplos de definição:
  - OMP_SCHEDULE=static
  - OMP_SCHEDULE=dynamic,3
  - OMP_SCHEDULE=guided,5

Dependências nos dados

- Se $S1$ e $S2$ são instruções que acedem à mesma posição de memória, e pelo menos uma delas escreve nessa posição, diz-se que existe uma **dependência nos dados** entre $S1$ e $S2$.
- A correcção de um programa paralelo poderá estar em causa quando:
  - $S1$ e $S2$ executam de forma concorrente (temos uma condição de corrida).
  - $S1$ e $S2$ executam numa ordem inversa à de uma execução sequencial.
- **Problema básico**: se $S1$ precede $S2$ num programa sequencial e as instruções são dependentes, então $S1$ deve preceder $S2$ num programa paralelo equivalente. Como vamos ver isso pode inibir grandemente o grau de paralelismo de um programa.

Dependência dos dados – definições

Sendo $S1$ e $S2$ duas instruções escrevemos $S1 \Rightarrow S2$ se $S1$ **precede** $S2$ durante a execução.
Para $S1 \Rightarrow S2$ definimos os seguintes tipos de dependências:

- **Dependência de fluxo** ("flow dependence") – $S1 \xrightarrow{\mathrm{F}} S2$ – se $S1$ escreve posições de memória acedidas por $S2$, sem outras escritas intervenientes.
- **Anti-dependência** ("anti-dependence"): $S1 \xrightarrow{\mathrm{A}} S2$ se $S1$ lê posições de memória re-escritas por $S2$, sem outras escritas intervenientes.
- **Dependência de output** ("output dependence"): $S1 \xrightarrow{\mathrm{O}} S2$ se $S1$ escreve posições de memória também escritas por $S2$, sem outras escritas intervenientes.
- De forma geral escrevemos $S1 \longrightarrow S2$ se algum dos tipos de dependência acima estiverem definidos.

Dependência dos dados – exemplo

```
// Sequential program fragment: S1 => S2 => S3 => S4
a = b - c; // (S1) reads b and c , writes a
c = a + 1; // (S2) reads a, writes c
a = a + c; // (S3) reads a and c, writes a
d = a - 2; // (S4) reads a, writes d
```

- Dependências de fluxo: $S1 \xrightarrow{\mathrm{F}} S2$ (a) $S1 \xrightarrow{\mathrm{F}} S3$ (a) $S2 \xrightarrow{\mathrm{F}} S3$ (c) $S3 \xrightarrow{\mathrm{F}} S4$ (a)
- Anti-dependências: $S1 \xrightarrow{\mathrm{A}} S2$ (c) $S2 \xrightarrow{\mathrm{A}} S3$ (a).
- Dependências de output: $S1 \xrightarrow{\mathrm{O}} S3$ (a).
- **Não** se considera $S1 \xrightarrow{\mathrm{F}} S4$ (a) porque existe uma escrita interveniente em a por $S3$.

Dependências "loop-carried"

```
#pragma omp for // WRONG!
for (i = 1; i < N; i++)
  a[i] = a[i] + a[i - 1]; // S1: has loop-carried dependencies
#pragma omp for ordered // OK but of little use !
for (i = 1; i < N; i ++)
  #pragma ordered
  a[i] = a[i] + a[i - 1];
```

- Dependências entre instruções em iterações diferentes do ciclo são designadas por **"loop-carried"**.
- Seria incorrecto a primeira paralelização do ciclo acima. Denotando $S1(i)$ pela instrução $S1$ executada na iteração i, temos $S1(i) \xrightarrow{\mathrm{F}} S1(i+1)$. É difícil paralelisar este ciclo ...
- O código estará correcto se usarmos a cláusula `ordered`, mas isso significa sequencializar as iterações.

Dependências "loop-carried" (2)

```
#pragma omp for // OK!
for (i = 1; i < N; i+=2)
  a[i] = a[i] + a[i - 1]; // S1: has no loop-carried dependencies
```

- Neste caso ciclo escreve em `a[1,3,5,...]` sem dependências entre iterações.
- A paralelisação é válida.

Dependências "loop-carried" (3)

```
// Standard matrix multiplication
#pragma omp for // OK!
for (i = 0; i < N; i++) // no loop-carried dependencies for i
  for (j = 0; j < N; j++) // no loop-carried dependencies for j
    for (k = 0; k < N; k++) // loop-carried dependencies for k
      c[i][j] = c[i][j] + a[i][k] * b[k][j]; // S
```

- As dependências "loop-carried" são entre iterações do ciclo mais interno (sobre `k`) envolvendo `c[i][j]`:

$$S(i,j,k) \overset{\mathrm{A,F,O}}{\longrightarrow} S(i,j,k+1).$$

- A paralelisação é válida, pois threads diferentes executarão iterações com gamas de valores diferentes para `i`.
- Quando temos vários ciclos imbricados, normalmente queremos paralelisar o ciclo exterior. É válida a existência de dependências nos dados entre sub-iterações dos ciclos interiores para a mesma iteração do ciclo exterior.

Remoção de dependências "loop-carried"

```
for (i = 0; i < N - 1; i++) {
  x = d[i] + i;          // S1
  a[i] = a[i + 1] + x; // S2
}
```

Dependências:

- $S1(i) \xrightarrow{\text{F}} S2(i)$ [x] (mesma iteração; não é "loop-carried"), $S1(i) \xrightarrow{\text{O}} S1(i+1)$ [x] e $S2(i) \xrightarrow{\text{A}} S1(i+1)$ [x]
- $S2(i) \xrightarrow{\text{A}} S2(i+1)$ [a]

Como remover as dependências "loop-carried"?

- x é um valor temporário, exemplo de falsa partilha de dados. As dependências são fáceis de eliminar de várias formas. **Q:** Ideias?
- Para resolver a anti-dependência no acesso a a poderíamos tentar ter os valores "antigos" de a à entrada do ciclo. **Q:** Ideias?

Remoção de dependências "loop-carried" (2)

```
for (i = 0; i < N - 1; i++) {
  x = d[i] + i;          // S1
  a[i] = a[i + 1] + x; // S2
}
```

Versão paralela:

```
#pragma omp for
for (i = 0; i < N - 1; i++) aux[i] = a[i + 1]; // S0
#pragma omp for
for (i = 0; i < N - 1; i++) {
  int x = d[i] + i;  // S1' - x now local to each iteration
  a[i] = aux[i] + x; // S2'
}
```

Usa-se buffer auxiliar (**aux**) e **x** é declarada como local ao ciclo (**private** também resultaria). $S0(i) \xrightarrow{\text{F}} S2'(i)$ [**aux[i]**] e $S0(i) \xrightarrow{\text{A}} S2'(i+1)$ [**a[i+1]**]. não são dependências "loop-carried", já que relacionam instruções de 2 ciclos diferentes executados em sequência.

Remoção de dependências "loop-carried" (3)

```
x = 0;
for (i = 0; i < N; i++)
  x = x + a[i]; // S1
```

Temos $S1(i) \xrightarrow{\mathrm{O,A,F}} S1(i+1)$. O código é paralelizável de forma simples com empregando uma redução num ciclo `omp for`:

```
x = 0;
#pragma omp for reduction(+: x)
for (i = 0; i < N; i++)
  x = x + a[i]; // x -> reduction
```

Em linha com o mecanismo de redução, `x` torna-se privada para as iterações de cada thread. Dependências "loop-carried" mantêm-se mas são locais a cada thread, antes do passo final de redução.

Removendo dependências "loop-carried" (4)

Um outro exemplo:

```
for (int i = 1; i < m; i++)
  for (int j = 0; j < n; j++)
     a[i][j] = 2 * a[i-1][j]; // S1
```

- Dependências: $S1(i,j) \xrightarrow{\mathrm{A,F}} S1(i+1,j)$.
- É válido e nesse caso eficiente inserirmos `#pragma omp for` em associação ao ciclo mais externo? E para o ciclo mais interno?

Removendo dependências "loop-carried" (5)

```
for (int i = 1; i < m; i++)
  #pragma omp for // correct, but efficient?
  for (int j = 0; j < n; j++)
     a[i][j] = 2 * a[i-1][j]; // S1
```

- Dependências: $S1(i,j) \xrightarrow{\text{A,F}} S1(i+1,j)$.
- As dependências "loop-carried" são ao nível do ciclo mais externo, portanto aplicar `#pragma omp for` a este seria incorrecto.
- Podemos fazê-lo `#pragma omp for` para o ciclo mais interno, mas ficamos com `m - 2` passos "fork/join" o que poderá terá grande impacto no desempenho do programa.
- **Q:** É possível fazer alguma coisa?

Removendo dependências "loop-carried" (6)

```
#pragma omp for // correct, but efficient?
for (int j = 0; j < n; j++)
  for (int i = 1; i < m; i++)
    a[i][j] = 2 * a[i-1][j]; // S1
```

- Dependências: $S1(i,j) \xrightarrow{\mathrm{A,F}} S1(i+1,j)$.
- Se invertermos os ciclos as dependências mantêm-se mas locais a cada thread, e passamos a precisar de apenas um passo "fork/join".
- No entanto localidade de acesso à memória é baixa, que deverá comprometer também o desempenho: em vez de acessos contíguos à memória temos uma distância ("stride") de n posições de memória em cada iteração.

Encadeamento de código e directivas órfãs

```
void someFunc() {
  #pragma omp for // orphaned directive
  for ( )
    ...
}
int main() {
   #pragma omp parallel
   {
     someFunc(); // part of parallel region
   }
   someFunc(); // directive will be ignored
}
```

- **Directiva órfã**: directiva OpenMP fora do âmbito directo de uma directiva `omp parallel`.
- Uma directiva orfã é considerada se executada no contexto de uma região paralela activa, e ignorada se no contexto de uma região sequencial.

Encadeamento de directivas

No caso de existirem vários níveis de encadeamento de directivas (órfãs ou não-órfãs) aplicam-se uma série de restrições:

- Uma secção `parallel` encadeada com (dentro de) outra secção `parallel` dá origem a um novo "team of threads", **se** a implementação suportar paralelismo imbricado ("nested parallelism") e este estiver ativa (`omp_set_nested()`). Caso contrário, a secção encadeada é executado por apenas uma thread.
- Directivas `for`, `sections`, `barrier`, `single`, `master`, `critical`, `ordered` não podem ser encadeadas entre si.

Âmbito de variáveis em chamadas a funções

Quando uma região paralela contém chamadas a uma função, no âmbito da função chamada:

- Variáveis globais são consideradas `shared`.
- Variáveis locais são consideradas `private`, excepto se declaradas com o modificador `static`.

É necessário cuidado para evitar partilha indevida de variáveis. Vamos ver um exemplo a seguir.

Âmbito de variáveis em chamadas a funções (2)

```
int w;

f(int a[], int n) {
  int i, j;
  #pragma omp parallel for private(w)
  for (i = 0; i < n; i++) {
    int q = w;
    w += i;
    for (j = 1; j <= 5; j++)
      g(&a[i], &q, j);
  }
}

g(int *x, int *y, int z) {
  static int s = 0;
  int k;
  s++;
  for (k = 0; k < z; k++)
    *x = *y + w;
}
```

Âmbito de variáveis em chamadas a funções(3)

Fun	Var	Âmbito	Explicação	OK?
f()	`a[]`	shared	declarada fora do construtor `parallel`	
	`n`	shared	declarada fora do construtor `parallel`	
	`i`	private	iterador do ciclo do construtor `for`	
	`j`	shared	declarada fora do construtor `parallel`	Não
	`q`	private	declarada dentro do construtor `parallel`	
	`w`	private	cláusula private	Não
g()	`x`	private	argumento da função	
	`*x`	shared	referência à variável `a`	
	`y`	private	argumento da função	
	`*y`	private	referência à variável `q`	
	`z`	private	argumento da função	
	`s`	shared	variável static	Não
	`k`	private	variável local à função	
	`w`	shared	variável global	Não