CAP-387(2016) – Tópicos Especiais em Computação Aplicada: Construção de Aplicações Massivamente Paralelas

Aula 23: Outros Detalhes de OpenMP

Celso L. Mendes, Stephan Stephany

LAC / INPE

Emails: celso.mendes@inpe.br, stephan.stephany@inpe.br

Escopo de Variáveis

- Programa serial
  - O escopo de uma variável é o conjunto das partes do programa onde a variável pode ser usada.
    - Var. local: escopo = função onde a variável é declarada
    - Var. global: escopo = todo o arquivo
- Programa em OpenMP
  - O escopo de uma variável indica qual/quais thread(s) pode(m) acessar a variável num bloco paralelo
    - Escopo=*shared*: todos os threads podem acessar a variável
    - Escopo=*private*: apenas um thread pode acessar a variável

Escopo de Variáveis (cont.)

- Escopos default:
  - Variáveis externas ao bloco paralelo: default = *shared*
  - Variáveis internas ao bloco paralelo: default = *private*
  - Mas… como já visto, o escopo default pode ser mudado pelo programador

Exemplo:

```
int id;
. . .
#pragma omp parallel private(id)
{
        id = omp_get_thread_num();
        …
}
```

Acesso Compartilhado - Problema

- Perigos do acesso a variáveis compartilhadas
  - Em alguns casos, pode ser necessário disciplinar o acesso

Exemplo:

```
int global_sum;                          <var.shared>
. . .
#pragma omp parallel
{
        int local_sum;                   <var.private>
        … (cálculo de local_sum) …
        global_sum += local_sum;         Erro! Acessos simultaneos
        …
}
```

Acesso Compartilhado Seguro

- Disciplina de acesso: opção *omp critical*
  - Idéia: forçar o acesso de um thread de cada vez a *global_sum*
  - Todos os threads executam o comando, porém um de cada vez
    - Notar a diferença em relação à opção *omp master* (apenas um thread executaria)

No exemplo anterior:

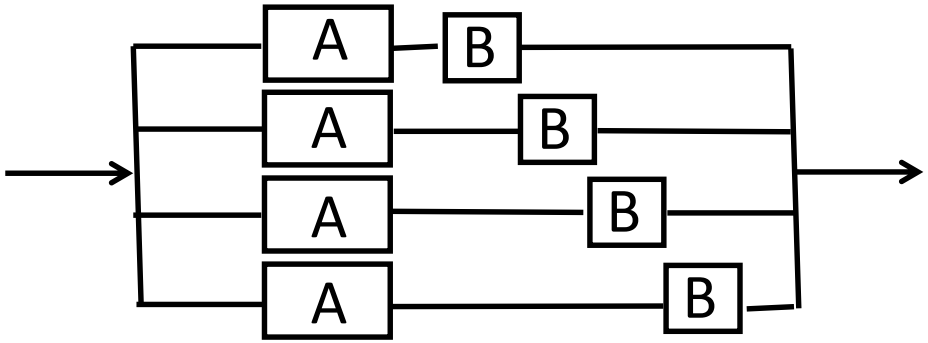
```
int global_sum;                          <var.shared>
. . .
#pragma omp parallel
{
        int local_sum;                   <var.private>
        … (cálculo de local_sum) …
#pragma omp critical
        global_sum += local_sum;         Atualizações seguras (mutuamente exclusivas)
        …
}
```

Overhead do Acesso Seguro

- Opção *omp critical* força serialização – possível ineficiência
  - Deve ser usada com cuidado, apenas quando de fato necessária

Num exemplo genérico:

```
#pragma omp parallel
{
        <bloco A>
#pragma omp critical
        <bloco B>
}
```



- Idealmente, T_A » T_B
  - ➢ Seção crítica deve ser a menor possível!
- Se $T_A \leq T_B$ então o paralelismo é pobre

Alternativa: Operação de Redução

- Combinação de valores é comum em programas reais

Exemplo:

```
int sum=0;                                  <var.shared>
#pragma omp parallel for
for (i=0; i<n; i++) sum += A[i] ;           Erro! Acessos simultaneos a sum

#pragma omp parallel for
for (i=0; i<n; i++)
#pragma omp critical
        sum += A[i];                        Correto! Porém, ineficiente: serializado!
```

Opção OMP Reduction

- Combinação de valores explicitada pelo programador

No exemplo anterior:

int sum=0; <var.shared>

#pragma omp parallel for reduction(+: sum)

for (i=0; i<n; i++) sum += A[i]; Acessos a *sum* são disciplinados

Forma geral: *reduction(operador: variável)*

tal que *operador* é um operador associativo e comutativo

- Em C: *operador* = {+, *, -, &, |, ^, &&, ||}

(OBS: funcionamento correto de "-" depende do compilador)

Restrições à Paralelização de Loops

- Forma Geral:

  #pragma omp parallel for

  for (index=start; index<end; index+=incr) …

- Restrições:
  - OpenMP só paraleliza loops onde o número de iterações é conhecido antes da execução do loop!
  - *index* deve ser um inteiro ou um ponteiro
  - *start, end* e *incr* devem ter tipos compatíveis e não podem mudar durante a execução do loop
  - Loops com *while* ou *do-while* não são paralelizáveis

Restrições à Paralelização de Loops

- Outros tipos de loops não-paralelizáveis:
  - Loops infinitos

    for (; ;) { … }
  - Loops com possível término antecipado

    for (…) { if (…) break; …}

    func(…) { … ; for (…) { if (…) return; … }
- Excessão: loop que pode ser paralelizado:

  for (…) { if (…) exit(); … }

Loops com Dependências

- Dependências *dentro* da mesma iteração não impedem paralelização

```
#pragma omp parallel for
 for (i=0; i<n; i++)  {
        x[i] = a + i*h ;
        y[i] = exp( x[i] );
}
```

- Dependências *entre* iterações são problemáticas

```
 for (i=2; i<n; i++)   fib[i] = fib[i-1] + fib[i-2] ;
```

Caso *omp parallel for* seja aplicado, execução irá gerar resultado errado!

Escalonamento de Loops

- Escalonamento: forma de distribuição das iterações do loop pelos threads existentes
- Forma Geral: *#pragma omp parallel for schedule(<tipo>,<chunk>)*
- Escalonamento ideal: Depende do tipo de loop

sum = 0.0;

for (i=0; i<n; i++) sum += f(i);

→ Duração de cada iteração depende de *f(i)*

Escalonamento Estático

- Forma Geral: *#pragma omp parallel for schedule(static,<chunk>)*
- Exemplo: *n*=12, Número de threads=3

Caso *<chunk>* = 1 :

Thread 0: processa iterações i=0,3,6,9
Thread 1: processa iterações i=1,4,7,10
Thread 2: processa iterações i=2,5,8,11

Caso *<chunk>* = 2 :

Thread 0: processa iterações i=0,1,6,7
Thread 1: processa iterações i=2,3,8,9
Thread 2: processa iterações i=4,5,10,11

Escalonamento Estático (cont.)

Caso *<chunk>* = 4 :

Thread 0: processa iterações i=0,1,2,3
Thread 1: processa iterações i=4,5,6,7
Thread 2: processa iterações i=8,9,10,11

- Este caso é equivalente à distribuição default
  <chunk> = (Número de iterações) / (Número de threads)
- Também conhecida como distribuição em blocos
- Ruim em alguns casos, por exemplo se *T(f(i))* aumenta com *i*
  - T(i=0)=a, T(i=1)=a+b, T(i=2)=a+2b, T(i=3)=a+3b, …
  - Thread-2 terá muito maior carga que Thread-0 !
    - Distribuição com <chunk>=1 traria mais equilíbrio
    - Distribuição com <chunk>=1 é conhecida como cíclica

Escalonamento Dinâmico

- Forma Geral: *#pragma omp parallel for schedule(dynamic,<chunk>)*
  - Cada thread executa um <*chunk*> de iterações
  - Ao terminar cada *chunk*, o thread requisita outro *chunk* ao runtime-system de OpenMP
  - <*chunk*> pode ser omitido no corpo da diretiva
    - Neste caso, assume-se <*chunk*>=1
  - Como as iterações são distribuídas durante a execução, há um pequeno overhead devido à distribuição

Escalonamento Guiado

- Forma Geral: *#pragma omp parallel for schedule(guided,<chunk>)*
  - Similar ao dinâmico: cada thread executa um *bloco* de iterações
  - Ao terminar um *bloco*, o thread requisita outro *bloco* ao runtime-system de OpenMP
  - Tamanho do novo *bloco* é então reduzido, até atingir *<chunk>*
  - Cada *bloco* tem tamanho proporcional ao número de iterações restantes dividido pelo número de threads
  - Se *<chunk>* é omitido no corpo da diretiva, assume-se 1

Exemplo: n=10.000, Número de threads=2, *<chunk>*=1

Tamanhos dos blocos: 5.000, 2.500, 1.250, 625, 312,156, ..., 1

Escalonamento em Tempo Real

- Forma Geral: *#pragma omp parallel for schedule(runtime)*
  - Determinado pelo valor atual da var. ambiental OMP_SCHEDULE
    Exemplo:

    export OMP_SCHEDULE = "static,1"

    → Loop vai ser distribuído de forma cíclica pelos threads
  - Var. ambiental pode ser configurada antes da execução
    - Mas também pode ser alterada pelo próprio programa!
    - Escalonamento real é ditado pelo valor de $OMP_SCHEDULE ao iniciar a execução do loop

Escalonamento Automático

- Forma Geral: *#pragma omp parallel for schedule(auto)*
  - Escalonamento é totalmente determinado pelo compilador de OpenMP
  - Potencialmente, diferentes resultados de desempenho com compiladores distintos

Escalonamentos – Qual Usar?

- Caso Comum: Todas as iterações têm durações aproximadamente iguais
  - Distribuição padrão basta (ou seja, *static*, por blocos)
- Caso Especial: Iterações têm duração crescente (ou decrescente) com o índice do loop
  - Distribuição cíclica é melhor (ou seja, "*static,1*")
- Caso Geral: Iterações com durações imprevisíveis
  - Distribuições *dynamic* ou *guided* podem ser mais recomendáveis
  - Observar que ambas causam algum overhead