Processamento Paralelo

AULA 7

Multithreaded Programming with OpenMP – Aula 4

Professor: Luiz Augusto Laranjeira
luiz.laranjeira@gmail.com

Material originalmente produzido pelo Prof. Jairo Panetta e adaptado para a FGA pelo Prof. Laranjeira.

Introdução à Programação Paralela no Modelo Fork-Join utilizando o Padrão OpenMP 2.0

Elementos de OpenMP

1. Diretivas
   - Sintaxe
   - Região Paralela
   - Cooperação de Trabalho
   - Combinação de Região Paralela e Cooperação de Trabalho
   - Sincronização
   - Ambiente
2. Funções
   - Ambiente
   - Sincronismo
   - Tempo
3. Variáveis de Ambiente
   - Ambiente

Diretivas de Sincronização

- Finalidade: sincronizar a execução das threads
- Obrigatoriamente no escopo dinâmico de uma região paralela
- Construções:
  - barrier
  - ordered
  - critical (mais tarde)
  - flush (mais tarde)
  - ...

Barrier

- Sincroniza todas as threads

```
#pragma omp barrier
```

- Semântica:
  - Cada thread que atinge a barreira aguarda até que todas as threads do time atinjam a barreira
- Restrições:
  - Todas as threads de um time encontram a barreira ou nenhuma thread a encontra
  - Comandos de cooperação de trabalho e barreiras devem ser encontrados na mesma ordem por todas as threads de um time
- Uso incorreto causa deadlock!!!

Exemplo

- Paralelize o laço seguinte usando OpenMP

```
for (i=1; i<10; i++) {
   b[i] = func(i);
   a[i] = b[i-1] + b[i];
}
```

- Tentativa 1:

```
#pragma omp parallel for private (i)
for (i=1; i<10; i++) {
   b[i] = func(i);
   a[i] = b[i-1] + b[i];
}
```

Errado pois iterações do laço não são independentes

Exemplo

- Tentativa 2:

```
#pragma omp parallel for private(i),  schedule(static,1)
for (i=1; i<11; i++) {
   b[i] = func(i);
#pragma omp barrier
   a[i] = b[i-1] + b[i];
}
```

Errado:

Não há garantia que todas as threads encontrem a barreira em todas as iterações (suponha 3 threads; quantas iterações por thread? 4 O que ocorre na quarta iteração?)

- Tentativa 3:

```
#pragma omp parallel for private(i)
for (i=1; i<11; i++)
   b[i] = func(i);
#pragma omp parallel for private (i)
for (i=1; i<11; i++)
   a[i] = b[i-1] + b[i];
```

Correto: por causa da barreira implícita ao fim do primeiro for.
Desvantagem: threads criadas duas vezes.

Exemplo

- Tentativa 4:

```
#pragma omp parallel
{
#pragma omp for private(i)
for (i=1; i<11; i++)
   b[i] = func(i);
#pragma omp for private (i)
for (i=1; i<11; i++)
   a[i] = b[i-1] + b[i];
}
```

Melhor: por causa da barreira implícita ao fim do primeiro for e porque não é necessário criar as threads duas vezes.

Ordered

- Finalidade:
  - Um <bloco> no corpo de um laço for é executado na ordem sequencial das iterações

```
#pragma omp ordered
    <structured-block>
```

- Semântica:
  - A cada instante, apenas uma thread pode executar <bloco>
  - Threads executam o <bloco> na ordem das iterações do laço
  - Consequentemente, ORDERED serializa a execução do <bloco> pelas threads na ordem das iterações do laço
  - Serializa apenas o <bloco>, não o resto do laço

Exemplo

```
#pragma omp parallel for private(i,c), ordered
for (i=lb; i<ub; i++) {
      c[i] = work1(i);
#pragma omp ordered
      printf(“ c[%d] = %d\n”, i, c[i]);
      work2(c[i]);
}
```

- Apenas a impressão é feita na ordem de execução sequencial

Elementos de OpenMP

1. Diretivas
   - Sintaxe
   - Região Paralela
   - Cooperação de Trabalho
   - Combinação de Região Paralela e Cooperação de Trabalho
   - Sincronização (apenas primitiva para região crítica)
   - Ambiente
2. Funções
   - Ambiente
   - Sincronismo
   - Tempo
3. Variáveis de Ambiente
   - Ambiente

Condição de Corrida

- Duas pessoas retiram, simultaneamente, R$ 100 da mesma conta em dois terminais bancários (M1 e M2) conectadas ao mesmo computador central
- O saldo da conta era R$ 1000 antes dos saques
- Cada um obtém R$ 100 e o novo saldo é R$ 900!!!
- Eis como:
  - M1 pede o saldo e obtém resposta 1000
  - M1 subtrai, localmente, 100 do saldo que conhece
  - M2 pede o saldo e obtém resposta 1000
  - M1 comunica ao computador central o novo saldo de 900
  - M2 subtrai, localmente, 100 do saldo que conhece
  - M2 comunica ao computador central o novo saldo de 900

Condição de Corrida

- Condição de corrida ("race condition") é um problema de programação concorrente que ocorre quando o resultado de uma computação muda com a ordem (velocidade) de execução das tarefas componentes
  - Há ordens de execução que resultam em resultados corretos
- Aparentemente, o saldo errado ocorre porque uma mesma informação (saldo) está armazenada em múltiplos computadores simultaneamente
  - Saldo no computador central e nos dois terminais
- É possível evitar a replicação da armazenagem?
  - Por exemplo, apenas o computador central calcula o saldo

Condição de Corrida - Um Clássico

- Suponha que a variável m armazene o saldo (m=1000) na memória do computador central.
  - Suponha computador central executando threads T1 e T2, originadas respectivamente pelas máquinas M1 e M2. Cada thread possui seu conjunto de registradores (denotados RX)

- Eis ordem de execução que resulta em m=900:

```
T1: R1 <- m
T1: R1 <- R1 - 100
T2: R1 <- m
T2: R1 <- R1 - 100
T1: m <- R1
T2: m <- R1
```

- Há replicação de armazenagem mesmo em máquinas com um único núcleo (CPU).

Condição de Corrida

- Condição de corrida é um problema de programação concorrente, originalmente causado por replicação de armazenagem
  - quer em máquinas multi-processadas quer em máquinas mono-processadas
- A solução requer ações em dois níveis:
  - Na programação
  - No suporte de hardware
- A programação deve ser modificada para evitar que duas threads gerem conflitos de acesso à mesma posição de memória
- Deve haver suporte de hardware para implementar a solução requerida pela programação

Exclusão Mútua e Região Crítica

- Uma solução é impedir que tarefas cooperantes executem simultaneamente as operações que geram o problema
- Essa solução, na programação, é denominada exclusão mútua (“mutual exclusion”)
- No exemplo anterior, a exclusão mútua obriga que o trecho

  m = m - 100

  seja executado por uma thread de cada vez
- A região do programa que requer execução exclusiva é denominada região crítica (“critical region”)
- Região crítica é programada por duas operações primitivas:

  <entra na região crítica>

  m = m - 100

  <sai da região crítica>

Semântica da Região Crítica

- A operação <entra na região crítica>

  permite uma única thread na região crítica a cada instante;
  caso múltiplas threads tentem entrar simultaneamente, apenas uma entra enquanto threads que tentam entrar aguardam na entrada;

- A operação <sai da região crítica>

  remove a thread da região crítica, permitindo que outra thread adentre a região

- Threads competindo pela entrada na região crítica aguardam em uma das formas:
  1. Continuamente testando se podem entrar na região crítica (“busy waiting”);
  2. Colocadas em estado de espera pelo sistema operacional (“sleeping”);
     dependendo de como a região crítica é implementada

Elementos de OpenMP

1. Diretivas
   - Sintaxe
   - Região Paralela
   - Cooperação de Trabalho
   - Combinação de Região Paralela e Cooperação de Trabalho
   - Sincronização (apenas primitiva para região crítica)
   - Ambiente
2. Funções
   - Ambiente
   - Sincronismo
   - Tempo
3. Variáveis de Ambiente
   - Ambiente

Critical

- Restringe o acesso ao <bloco> a uma única thread a cada instante, implementando região crítica

```
#pragma omp critical (nome)
<structured-block>
```

- Semântica:
  - Threads aguardam no início da região até que nenhuma outra thread esteja executando qualquer região critica com o mesmo nome

- Detalhe:
  - nome é opcional

Critical

- Restrições:
  - <nome> é entidade global do programa
  - pode haver múltiplas diretivas CRITICAL com o mesmo nome
  - <nome> é opcional; todas as diretivas CRITICAL sem nome são mapeadas no mesmo nome (mesma região crítica)
  - são proibidos saltos de/para o <bloco>

Exemplo

- Fila de tarefas
  - Suponha duas filas de tarefas independentes, denominadas *x* e *y*
  - Suponha que dequeue(fila) retorna (e retira) a próxima tarefa da fila

```
#pragma omp parallel private(indx_next, indy_next)
do {
#pragma omp critical (x_axis)
   indx_next = dequeue(x);
    if (indx_next) work(indx_next);
#pragma omp critical (y_axis)
   indy_next = dequeue(y);
    if (indy_next) work(indy_next);
} while (indx_next | indy_next);
```

as duas regiões críticas são independentes

Implementando Primitvas de Exclusão Mútua

Locks

- Uma forma de implementar as operações de entrada e saída na região crítica é por uma trava (“lock”), também chamada de trava de exclusão mútua (“mutex lock”), dentre outros nomes

- Um “lock” é uma posição de memória que, quando inicializada, assume um de dois estados (“locked” ou “unlocked”) e possui duas operações:
  1. set_lock (lock), que implementa a entrada na região crítica;
  2. unset_lock (lock), que implementa a saída na região crítica

- A diretiva CRITICAL esconde o lock
  - lock é a construção primitiva

Semântica de Locks

- Semântica de set_lock (lock)
  - A thread que invoca esta função permanece bloqueada até que o lock esteja no estado “unlocked”
  - A execução da função torna o estado do lock em “locked”
  - É garantido que apenas uma thread (das que porventura estejam bloqueadas esperando) obtenha o estado “unlocked”
- Semântica de unset_lock (lock)
  - A execução da função torna o lock “unlocked”
- Como implementar “locks”?
  - Garantir que uma única thread adquira o “lock” é um problema de exclusão mútua; (retornamos ao problema anterior???)
  - A implementação requer suporte de hardware

Realização de Locks

- Represente os estados “unlocked” (0) e “locked” (1)
- A operação set_lock é implantado por uma instrução atômica (ou seja, indivisível) em hardware como, por exemplo, test-and-set
- test-and-set (t&s) executa atomicamente (i.e., sem interrupção):
  1. lê o conteúdo de uma posição de memória para um registrador
  2. se o conteúdo for igual a 0 escreve 1 na posição de memória, caso contrário retorna sem nenhuma ação.
- A operação unset_lock é implantada trivialmente
  - basta armazenar 0 na posição de memória

Implementação de Locks

- Código para set_lock ("busy waiting"):

```
try_get_lock:   t&s r0, lock          /* r0 <- (lock) e (lock) <- 1        */
                bnz r0, try_get_lock  /* tenta de novo se (lock) era 1 */
                                      /* bnz = branch if not zero          */
```

<região crítica>

- Consequentemente:
  - Como t&s é atômica, uma única thread adquire o lock
  - Se lock estava 1 ("locked"), a thread continua tentando adquirir o lock
  - Se lock estava 0 ("unlocked"), a thread adquire o lock, que se torna igual a 1 ("locked")

O que garante a semântica de set_lock

Implementação de Locks

Código assembly de região crítica guardada por lock com implementação ("busy waiting"):

```
/* set_lock  (2 instrs de máquina) */
try_get_lock:   t&s r0, lock        /* r0 <- (lock) e (lock) <- 1        */
                bnz r0, try_get_lock /* tenta de novo se (lock) era 1 */
                                     /* bnz = branch if not zero        */

/* região       inst1
   crítica */   inst2
                ...
                instn

                sto lock, 0         /* unset_lock  (1 instr de máquina) */
```

Outras Instruções de Sincronismo

- Test-and-set é um exemplo de instrução atômica de sincronismo
  - Há outras formas
- Intel e AMD oferecem compare-and-swap, instrução de máquina com três argumentos:
  - Endereço de memória (*m*)
  - Valor esperado armazenado no endereço de memória (*e*)
  - Valor a armazenar no endereço de memoria (*f*)
- Compare-and-swap realiza atomicamente (sem interrupção) as seguintes operações:
  - Se mem(*m*) == *e*, substitui *e* por *f* e retorna true;
  - Caso contrário, retorna false

Outras Instruções de Sincronismo

- Alpha, IBM Power, MIPS e ARM fornecem um par de instruções:
  - load-linked (LL)
  - store-conditional (SC)
- Load-linked lê o conteúdo de uma posição de memória para um registrador
- Store-conditional altera o conteúdo dessa posição para um novo valor se o conteúdo dessa posição foi inalterado desde o último Load-linked para essa posição
  - Retorna true ou false

Elementos de OpenMP

1. Diretivas
   - Sintaxe
   - Região Paralela
   - Cooperação de Trabalho
   - Combinação de Região Paralela e Cooperação de Trabalho
   - Sincronização
   - Ambiente
2. Funções Ambiente
   - Sincronismo (locks)
   - Tempo
   - Variáveis de Ambiente
3. Ambiente
   -

Locks Simples em OpenMP

- Locks permitem implementar exclusão mútua em trechos arbitrários de um programa
- A região crítica é definida dinamicamente, pois locks são implementados por invocações a funções
  - enquanto CRITICAL é definido estaticamente e obrigatoriamente começa e termina no mesmo procedimento
- omp_set_lock(svar)
  - A thread que invoca esta função permanece bloqueada até que svar (o lock) esteja no estado “unlocked”
  - A execução da função torna o estado do lock em “locked” e faz que o lock pertença à thread
  - É garantido que apenas uma thread obtenha o estado “unlocked”, ou seja, que apenas uma thread detenha o “lock”
- omp_unset_lock(svar)
  - Só pode ser invocada pela thread que detém o lock
  - A execução da função torna o lock “unlocked”

Locks Simples em OpenMP

- omp_test_lock(svar)
  - Similar a omp_set_lock(), exceto que threads que não obtém o lock não ficam bloqueadas; retornam com valor .FALSE.
  - A thread que obtém o lock retorna .TRUE.
- omp_init_lock(svar)
  - Cria o lock no estado “unlocked”
- omp_destroy_lock(svar)
  - Destrói o lock que está, obrigatoriamente, no estado “unlocked”
- Há muitos detalhes e outros tipos de locks (“nested locks”)
  - Vide o padrão

Elementos de OpenMP

1. Diretivas
   - Sintaxe
   - Região Paralela
   - Cooperação de Trabalho
   - Combinação de Região Paralela e Cooperação de Trabalho
   - Sincronização
   - Ambiente
2. Funções
   - Ambiente
   - Sincronismo
   - Tempo
3. Variáveis de Ambiente
   - Ambiente

Funções de Ambiente de Execução

- omp_get_num_threads ()
  - Quando invocada em região paralela, retorna o número de threads em execução
  - Quando invocada em região serial, retorna 1
- omp_get_thread_num ()
  - Quando invocada em região paralela, retorna o número desta thread
    - Inteiro entre 0 (master thread) e OMP_GET_NUM_THREADS()-1
  - Quando invocada em região serial, retorna 0 (master thread)
- omp_get_num_procs ()
  - Retorna o número de processadores disponíveis para o programa
- omp_in_parallel ()
  - Retorna FALSE (0) se fora de região paralela, TRUE (!=0) caso contrário

Procedimentos Thread Safe

Procedimentos Thread Safe

- Um procedimento é denominado “thread safe” quando instâncias desse procedimento podem ser executadas simultaneamente em um único espaço de endereçamento sem interferência mútua.

- Observe que variáveis locais ao procedimento são irrelevantes para “thread safe”
  - Pois ocupam posições em “stack” distintas;
  - Exceção: variáveis locais e estáticas

- Inibem “thread safe” as variáveis escritas pelo procedimento que sejam:
  - Globais ou
  - Estáticas

Procedimentos Thread Safe

- Observe que procedimentos “thread safe” carregam a parte mutável do estado da computação apenas por argumentos
- É particularmente difícil determinar se um procedimento é “thread safe” ou não quando o procedimento é a raiz de longa árvore de chamadas
- Para paralelizar múltiplas invocações de um procedimento, não basta que as computações das invocações do procedimento sejam independentes
  - É necessário que as codificações sejam independentes
- Thread safe é característica essencial de trechos paralelizáveis de programas no modelo fork-join

Sumário

1. Condição de corrida ocorre quando o resultado de uma computação muda com a ordem de execução das tarefas componentes (por competição entre operações de acesso à memória onde pelo menos uma das operações é de escrita)

2. Uma solução para eliminar a condição de corrida é exclusão mútua (impedir que tarefas cooperantes executem simultaneamente, ou seja, serializando o acesso à memória)

3. Exclusão mútua é implementada, na programação, por uma seção crítica, iniciada por operação para adquirir um lock e encerrada por operação que liberta o lock; o lock serializa o acesso à memória

4. A implementação do lock requer suporte de hardware por meio de instruções atômicas de sincronismo como test-and-set