Processamento Paralelo

AULA 8

Modelo de Memória OpenMP

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

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

Agenda

- Modelo de memoria OpenMP
- Comunicações não bloqueantes MPI
- Comunicações Coletivas MPI
- MPI2 – Comunicação Unilateral

Modelo de Memória OpenMP

Fontes:

Adve, Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, IEEE Computer, Dec. 1996

Padrão OpenMP

Modelo de Consistência de Memória

- Um modelo de consistência de memória é uma interface entre o programador e o sistema computacional
- O modelo de consistência de memória é um contrato entre a linguagem de programação e o hardware
- Fornece uma abstração de como a memória do sistema computacional se comporta e fornece limites para implementações/otimizações de hardware e compiladores
- Toda linguagem de programação contém um modelo de consistência de memória que, obrigatoriamente, é implementado pelos compiladores e pelo hardware.

Modelo de Consistência de Memória

- Em programação sequencial, usa-se a abstração que idas à memória ocorrem na ordem de execução do programa
  - Isto garante que a leitura de uma posição de memória retorna o último valor nela escrito, na ordem do programa (program order)
- Esta abstração continua válida em programas sequenciais (uma única thread), onde há um único fluxo de controle e um único fluxo de instruções
- "program order" é um modelo de consistência de memória

Modelo de Consistência de Memória

- Entretanto, operações de escrita e leitura na memória ***não necessariamente terminam*** na ordem do programa
- Há otimizações em software e em hardware que podem alterar a ordem de término das operações:
  - o compilador pode mover código de instruções independentes;
  - hardware pode executar instruções fora de ordem;
  - a arquitetura de memória pode demandar tempos distintos para a mesma operação em posições de memória distintas.
- Mas nenhuma otimização pode romper o modelo de consistência de memória (“program order” para programas com uma única thread)

Otimização: Compilador Muda Código

```
m = x;              (P1)
.......
m = m+1;            (P2)
.......
if (m < 10) {       (P3)
.......
}
```

sem referências a *m* nas entrelinhas.....

- O compilador é livre para
  - evitar a escrita de *m* em P1 (desde que a faça antes de P2)
  - adiantar o incremento de *m* em P2 (desde que a faça depois de P1)
  - adiar a escrita de *m* em P2 até imediatamente antes de P3
- Ou ainda, executar P2 imediatamente após P1
- Nenhuma dessas alterações rompe “program order”

Paralelismo em Memória Central

Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

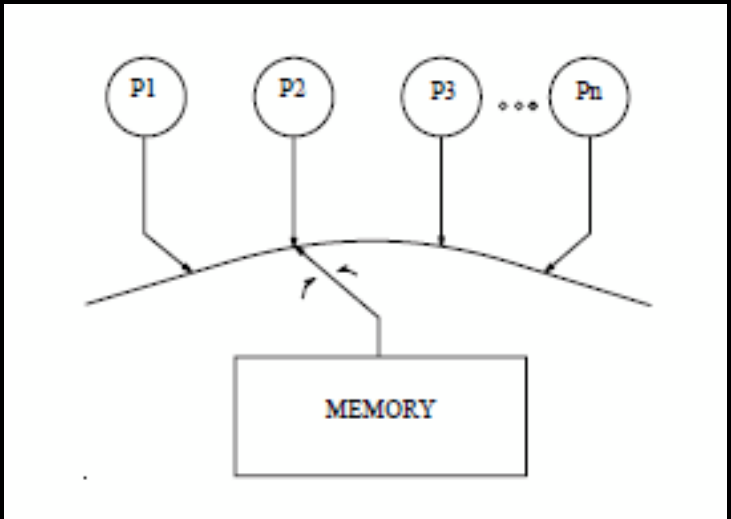
- O trecho acima (algoritmo de Dekker) garante que as duas threads não entram simultaneamente na seção crítica
  - Dependendo da ordem de execução dos comandos entre P1 e P2, é possível que nenhuma thread entre ou que apenas uma entre
- A demonstração usa as operações de memória das duas threads para garantir a correção do algoritmo
  - não é garantido por "program order" pois não é programa sequencial

Consistência Sequencial

- Sequential consistency é um modelo de consistência de memória para *programas paralelos em máquinas de memória central*

- Em uma máquina multiprocessada de memória central, consistência sequencial é definida por:
  1. As operações de acesso a memória do conjunto de processadores são executadas em alguma ordem sequencial
  2. As operações em cada processador ocorrem em "program order" dentro da sequência definida em (1)

Consistência Seqüencial



Initially Flag1 = Flag2 = 0

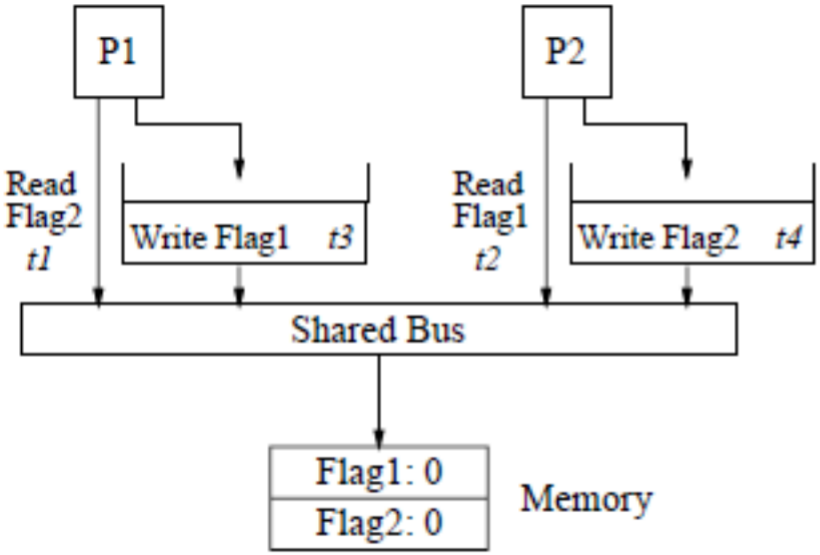
P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

- Consistência seqüencial garante a correção do algoritmo de Dekker
- Consistência seqüencial é como pensamos intuitivamente em um programa paralelo
- Mas consistência seqüencial inibe otimizações importantes em sw e em hw
  - por exemplo, compiladores não podem mover a escrita de FlagX para depois da leitura de FlagY, mesmo que independentes (no mesmo processo)

Otimização: Write Buffer

- Suponha que operações de leitura à memória terminem mais rapidamente que operações de escrita à memória
- Uma otimização simples seria montar um “buffer”, em hardware, para as escritas
- Escritas à memória retornariam assim que dispostas no buffer

Write Buffers



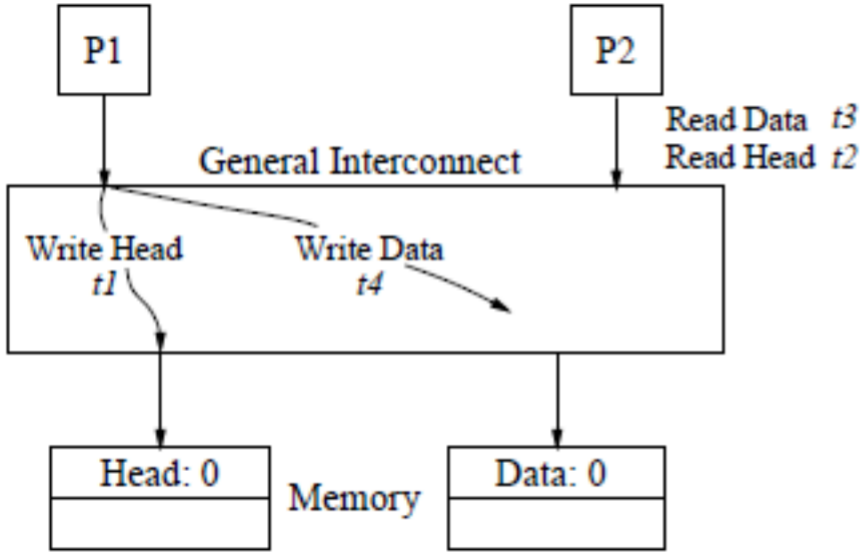
P1	P2
Flag1 = 1	Flag2 = 1
if (Flag2 == 0)	if (Flag1 == 0)
critical section	*critical section*

- "Write buffers" violam a consistência sequencial, pois P1 e P2 podem terminar a execução das leituras antes do término das escritas
  - seria uma otimização proibida por consistência sequencial

Otimização: Bancos de Memória

- Suponha que a memória central é dividida em “bancos”, com endereços consecutivos em bancos consecutivos
  - permite operações simultâneas à memória, desde que em bancos distintos
- Suponha que entre os processadores e a memória exista uma rede de interconexão
  - O tempo de resposta da rede varia com a distância entre o processador e o banco de memória acessado
- Suponha que “reads” e “writes” tenham velocidades distintas

Bancos de Memória



P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

- Bancos conectados por redes violam o princípio da consistência seqüencial, pois o valor de Data obtido por P2 pode não ser o escrito por P1
  - basta que a escrita de Data por P1 demore mais que a leitura de Data por P2

Otimização: Coerência de Cache

- Em uma máquina multiprocessada, cada processador possui pelo menos uma memória cache privada ao processador
  - Invisível aos outros processadores
- Suponha que todos os processadores tenham cópias em seu cache privado de uma mesma posição de memória, em determinado instante da execução
- Se um dos processadores altera essa posição de memória, é necessário algum procedimento para eliminar a inconsistência entre os valores na memória e nos caches
  - uma forma é invalidar os demais caches
  - outra forma é propagar o novo valor
- Operações caras

Solução: Outros Modelos de Memória

- O modelo de consistência sequencial impede otimizações importantes em hw e em sw
- Há propostas de diversos modelos que relaxam alguma condição imposta por consistência sequencial e permitem otimizações importantes
- Esses modelos são denominados modelos de consistência relaxada (relaxed consistency)
  - relaxada porque os valores em uma thread não são obrigatoriamente coerentes, o tempo todo, com os valores na memória
- OpenMP usa um desses modelos

Modelo de Memória OpenMP

- Modelo de memória de programas OpenMP:
  - central (shared memory)
  - consistência relaxada (relaxed consistency)
  - ordenação fraca (weak ordering)

- O modelo é definido por:
  - "program order" em cada thread
  - operações que sincronizam os valores da thread (no cache ou nos registradores do núcleo onde a thread está rodando) com o conteúdo da memória
  - ordenação entre as instâncias dessas operações

- A diretiva flush impõe sincronismo das caches usadas pelas threads com a memória

Mathematical Parenthesis:Weak Ordering

Informal Definition:

Weak ordering on a finite collection of objects:

Objects may be partitioned into classes such that all the objects in the same class are related and each class has exactly one *immediate predecessor* class and one *immediate successor* class with two exceptions:

- The *first* class has no predecessor; and
- The *last* class has no successor.

Mathematical Parenthesis:Weak Ordering

A binary relationship $\mathcal{R}$, expressed as $a \leq b$ (reads *a precedes or is equivalent to b*) between two objects is said to be a weak ordering if for any a and b:

- For any a and b, either $a \leq b$ or $b \leq a$
- If ($a \leq b$ and $b \leq a$) then $a \equiv b$
- $\mathcal{R}$ is transitive: if ($a \leq b$ and $b \leq c$) then $a \leq c$
- The equivalence relationship is transitive: if ($a \sim b$ and $b \sim c$) then $a \sim c$.
- We say that $a < b$ (reads *a precedes b*) if $a \leq b$ and $a \not\sim b$ and the relationship defined by $<$ describes a *strict weak order*.

Given n objects a_1, a_2, a_3, …, a_n which have a weak ordering, we may order them such that $a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_n$ however, there may be objects which are equivalent to each other at any point.

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

- Tais diretivas existem obrigatoriamente no escopo dinâmico de uma região paralela

- Construções:
  - barrier (já vimos)
  - ordered (já vimos)
  - critical (já vimos)
  - flush (agora)
  - ...

Diretiva de Sincronização *flush*

- Consistência de memória entre as threads

  #pragma omp flush (lista)

  - onde (lista) é a lista de variáveis “shared” a ser atualizada
  - (lista) é opcional; sua ausência atualiza todas as variáveis “shared” visíveis a essa thread

- Semântica:

  1. Se a thread atualizou o valor de uma variável da lista desde a última execução de *flush*, a execução de *flush* só termina após esse valor ser escrito na memória central
  2. Todos os valores locais da lista de variáveis são destruídos, obrigando o acesso à memória central no próximo uso da variável

Semântica de *flush*

- A diretiva *flush* recebe uma lista de variáveis “shared”
- *flush* é uma diretiva local a uma thread
- Compiladores não podem mover operações de memória a uma variável da lista além do *flush* dessa variável
- Se uma variável da lista for escrita pela thread que executa o *flush* antes do *flush*, o *flush* só termina quando terminar a escrita dessa variável na memória
- Cópias locais de todas as variáveis da lista são destruídas
  - A próxima leitura da variável obrigatoriamente irá à memória
  - Observe que são cópias locais (cache, registradores) de variáveis “shared” e não “private”.

Semântica de *flush*

- Há um *flush* (sem lista de variáveis) implícito:
  - Na entrada e na saída de PARALLEL, CRITICAL, ORDERED
  - Na saída de todas as operações de cooperação de trabalho
  - Durante uma BARRIER
  - Durante operações de LOCK
  - Em suma, nos pontos de sincronização das threads

- Cuidados:
  - *flush* atua apenas na thread que a invoca
  - A ordem de execução de *flush* em múltiplas threads pode alterar o resultado da computação
  - Nenhum sincronismo de execução entre as threads é garantido por *flush*

Conclusão

- Não há qualquer garantia que uma variável global alterada por uma thread seja atualizada em outra thread até que ambas as threads
  - passem por um ponto de sincronização do programa, ou
  - encontrem diretiva *flush*
- Muito cuidado ao substituir diretivas OpenMP por código que parece ser equivalente
  - Ter certeza que todas as threads usam os *flush* adequados, na ordem adequada, é não trivial

Exemplo Sequencial

```
int main() {
#define VSIZE 1024
  int i;
  int b[VSIZE], c[VSIZE];
  b[0]=0; c[0]=0;
  for (i=1; i<VSIZE; i++) {
       b[i]=i;
       c[i]=b[i-1];
  }
  printf("ultimo c= %d\n", c[VSIZE-1]);
  exit(0);
}
```

Exemplo Paralelo (Errado)

```
int i,  b[VSIZE], c[VSIZE];
#pragma omp parallel
{
#pragma omp serial
       {  b[0]=0; c[0]=0;  }
#pragma omp for
       for (i=1; i<VSIZE; i++)   {
            b[i]=i;
            c[i]=b[i-1];
       }
  }
  printf("ultimo c= %d\n", c[VSIZE-1]);
```

- Nada garante que b[i-1] será o valor atualizado

Exemplo Paralelo (Certo)

```
int i,  b[VSIZE], c[VSIZE];
#pragma omp parallel
  {
#pragma omp serial
      {b[0]=0; c[0]=0;}
#pragma omp for
      for (i=1; i<VSIZE; i++)  b[i]=i;
#pragma omp for
      for (i=1; i<VSIZE; i++)  c[i]=b[i-1];
  }
  printf("ultimo c= %d\n", c[VSIZE-1]);
```

- Sincronismo e flush ao final do primeiro “omp for” garante que b[i-1] será o valor atualizado

Exemplo 2: O que é Impresso?

```
 int x;
  x=2;
#pragma omp parallel num_threads(2) shared(x)
  {
     if (omp_get_thread_num() == 1)   x=5;
     printf("1: thread %d, x=%d\n", omp_get_thread_num(), x);
  }
  printf("2: thread %d, x=%d\n", omp_get_thread_num(),  x);
```

Exemplo 2 Imprime:

1: thread 0, x=2

1: thread 1, x=5

2: thread 0, x=5

Exemplo 3: O que é Impresso?

```
 int x;
  x=2;
#pragma omp parallel num_threads(2) shared(x)
  {
    if (omp_get_thread_num() == 1)   x=5;
#pragma omp barrier
    printf("1: thread %d, x=%d\n", omp_get_thread_num(), x);
  }
  printf("2: thread %d, x=%d\n", omp_get_thread_num(), x);
```

Exemplo 3 Imprime:

1: thread 0, x=5

1: thread 1, x=5

2: thread 0, x=5