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

Aula 22: Aspectos Básicos de OpenMP

Celso L. Mendes, Stephan Stephany

LAC / INPE

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

Paralelismo com Threads

- Programação com threads
  - Um “mal necessário”: forma mais efetiva de se utilizer todos os núcleos de um nó num sistema paralelo
  - Insuficiente por si só: limitada ao ambiente de um nó
- Principais problemas do uso de threads:
  - Acesso comum à memória, intrínseco no modelo de memória compartilhada, pode criar vários problemas, tanto de desempenho como de corretude da execução do programa!
  - Ordem de operações na memória por um thread pode não ser a esperada pelos outros threads (por exemplo, devido a algum reordenamento de instruções pelo compilador)

Paralelismo com Threads

- Exemplo:
  - Suponha dois threads, cada um com os seguintes códigos, onde *a* e *b* são variáveis comuns:

Thread X:
```
a = 1;
barreira();
b = 2;
a = 2;
```

Thread Y:
```
b = 1;
barreira();
while (a == 1)  ;
printf("b = %d\n", b);
```

  - Que valor de *b* será impresso pelo Thread-Y?
  - Este valor será impresso em *qualquer* execução dos threads?

Paralelismo com Threads

- Exemplo (cont.):
  - Códigos:

Thread X:
```
a = 1;
barreira();
b = 2;
a = 2;
```

Thread Y:
```
b = 1;
barreira();
while (a == 1)  ;
printf("b = %d\n", b);
```

  - Possíveis valores de saída:
    - 2 (esperado!)
    - 1 (código do Thread-X é reordenado pelo compilador!)
    - Nada! (valor de *a* nunca é mudado pelo Thread-X!)

Paralelismo com Threads

- Exemplo (cont.):
  - Códigos:

Thread X:
```
a = 1;
barreira();
b = 2;
a = 2;
```

Thread Y:
```
b = 1;
barreira();
while (a == 1)  ;
printf("b = %d\n", b);
```

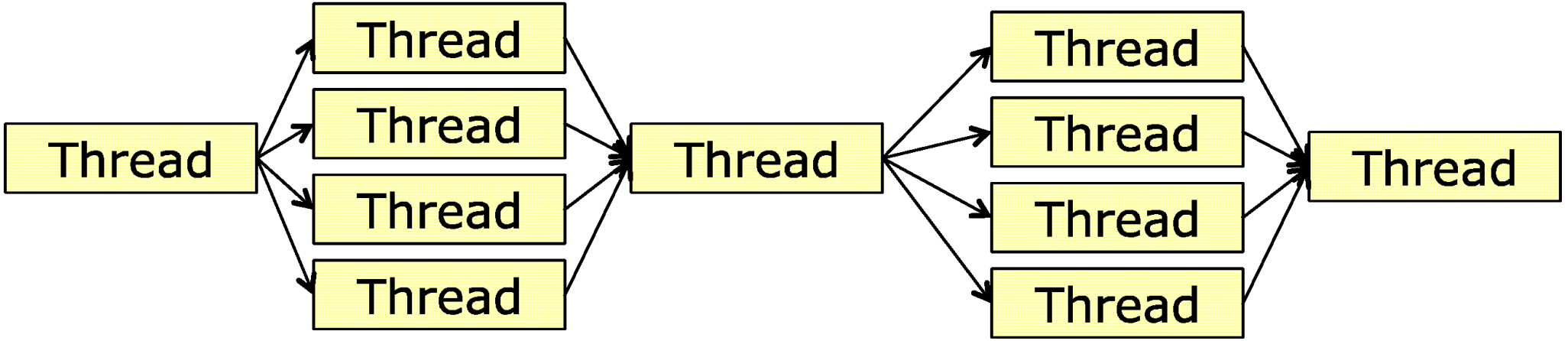
  - Problema de corretude – raiz: acessos às variáveis comuns
  - Pode ocorrer em qualquer modelo de programação com threads
  - Nas próximas aulas, veremos outros problemas de desempenho

Paralelismo com OpenMP

- Principais motivações:
  - Passar o gerenciamento de threads para o compilador
  - Simplificar a programação e migração de programas
- OpenMP:
  - Padrão definido pelo fórum OpenMP, década de 90
  - Baseado em dois componentes:
    1. Diretivas inseridas no código-fonte
    2. Biblioteca com funções de apoio
  - Disponível para programas em C e Fortran

Paralelismo com OpenMP

- Modelo de Paralelismo: *fork/join*
  - Threads são automaticamente criados/removidos pelo compilador, com base nas diretivas presentes
  - Trabalho de cada thread é designado pelo computador
  - Notar que, num *join*, os threads podem ser mantidos dormentes (ao invés de removidos), caso seja conveniente



Programação com OpenMP

- Principais mudanças exigidas no programa original
  - Inserção de header-file *omp.h*
  - Inserção de diretivas
    - C: #pragma omp construct [clause …]
    - Fortran: !OMP construct [clause …]
  - Inserção de chamadas a funções da biblioteca OpenMP:
    - C: n = *omp_get_...(…)* ou *omp_set_...(…)*
    - Fortran:
      - Subrotinas: *call omp_set_... (…)*
      - Funções: n = *omp_get_... (…)*

Programação com OpenMP

- Formas de expressar paralelismo com diretivas
  1. Regiões paralelas
     - Trechos genéricos de código, com entrada única, saída única
     - Cada thread executa uma instância de toda a região
     - Forma geral:

```
#pragma omp parallel
{
        trecho genérico de código   (executado por cada thread)
}
```

     - Um único thread executa antes da região, um único após a região
       - Normalmente o thread principal é chamado “master thread”
     - Início da região paralela: *fork* ; final da região paralela: *join*
     - Por default, há uma barreira entre os threads no fim da região

Programação com OpenMP

- Formas de expressar paralelismo com diretivas (cont.)
  2. Loops paralelos
     - Caso especial de região paralela
     - Cada thread executa um subconjunto das iterações do loop
     - Forma geral: *#pragma omp parallel for*
       for (i=0; i<n; i++) { … }
  3. Tarefas paralelas (a partir de OpenMP-3)
     - Comando após a diretiva dá origem dinamicamente a novos threads
     - Forma geral: *#pragma omp task*
       comando …

Exemplo: *Hello-World*

Programa Original:

```
#include <stdio.h>

int main(int argc, char *argv[])
{

        int id = 0;
        int np = 1;
        printf( "Hello world %d of %d\n", id, np);

return 0;
}
```

Hello-World em OpenMP

Programa Paralelizado com OpenMP:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
        omp_set_num_threads (4);
#pragma omp parallel
  {
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf( "Hello world %d of %d\n", id, np);
  }
return 0;
}
```

Região Paralela

Observações sobre *Hello-World*

- Escopo de variáveis
  - Variáveis externas à região paralela:
    - Compartilhadas por todos os threads
  - Variáveis internas à região paralela:
    - Exclusivas de cada thread, por default
  - Porém, variáveis externas podem ser tornadas privativas:

```
int id;
. . .
#pragma omp parallel private(id)
{
        id = omp_get_thread_num();
        printf("minha id = %d\n",id);
}
```

Exemplo com Loop em OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int global_var = 4 ;
void func ();
#define N 8
int main(argc,argv)
int argc; char *argv[];
{
    int i, count ;
    count = global_var ;
    /* Invoca a execucao dos varios threads */
#pragma omp parallel for num_threads(count)
    for (i=0; i<N; i++)  func(i) ;
    /* Thread principal imprime mensagem */
    printf("Este e' o thread principal\n");
}
void func (int arg) {
    int local_var;
    /* thread realiza trabalho individual */
    local_var = omp_get_thread_num();
    printf("--> Este e' o thread-omp %d de um total de %d, arg=%d\n",
        local_var, global_var, arg);
}
```

diretiva inserida no código

função da biblioteca OpenMP

Loop em OpenMP – Saída

Resultado da execução do exemplo:

```
--> Este e' o thread-omp 1 de um total de 4, arg=2
--> Este e' o thread-omp 1 de um total de 4, arg=3
--> Este e' o thread-omp 2 de um total de 4, arg=4
--> Este e' o thread-omp 2 de um total de 4, arg=5
--> Este e' o thread-omp 3 de um total de 4, arg=6
--> Este e' o thread-omp 3 de um total de 4, arg=7
--> Este e' o thread-omp 0 de um total de 4, arg=0
--> Este e' o thread-omp 0 de um total de 4, arg=1
Este e' o thread principal
```

Programas com OpenMP

- Principais características:
  - Compiladores podem ignorar as diretivas
  - Cada thread tem um identificador: 0, 1, 2,…
    - thread principal sempre existe, e tem identificador = 0
    - identificador pode ser recuperado com *omp_get_thread_num()*
    - total de threads pode ser recuperado com *omp_get_num_threads()*
  - É possível requerer que variáveis globais sejam privatizadas
    - implementado através de anotação extra na diretiva
  - Variáveis de índice dos loops são manipuladas adequadamente, de forma automática

Programas com OpenMP (cont.)

- Principais características (cont.):
  - Número de threads pode ser definido de várias formas:
    - na própria diretiva (como no exemplo dado)
    - chamada à função da biblioteca OpenMP:

      omp_set_num_threads(n)
    - com variável ambiental:

      export OMP_NUM_THREADS=n

      OBS: ver exemplos de uso na página de manual do Santos Dumont

Regiões *Master* em OpenMP

- Utilização:
  - Certos trechos da região paralela que precisem ser executados por um único thread (thread principal)
  - Exemplo: leitura/escrita de arquivos, saída de resultados, etc.
  - Forma geral:

```
#pragma omp parallel
{
. . .
#pragma omp master
   {
        int k = omp_get_num_threads();
        printf("numero de threads = %d\n",k);
   }
}
```

Execução de Loops em OpenMP

- Distribuição das iterações do loop pelos threads
  - Distribuição *static*: definida em tempo de compilação, várias formas disponíveis:
    - Distribuição em bloco: {i=0,1},{i=2,3},{i=4,5},{i=6,7}
    - Distribuição cíclica: {i=0,4},{i=1,5},{i=2,6},{i=3,7}
    - Outras formas de distribuição são possíveis
  - Distribuição *dynamic*: definida em tempo de execução
    - Útil quando a duração de cada iteração é imprevisível
  - Distribuição *guided*: caso especial de *dynamic*, reduz overhead
  - Forma geral: *#pragma omp parallel for schedule(<tipo>,chunk)*

OpenMP - Resumo

- Principais vantagens
  - Simplificação da programação em relação a Pthreads
  - Permite paralelização "incremental" de programas antigos
  - Padrão maduro – duas décadas de amadurecimento
  - Quase todos os compiladores atuais suportam o padrão
  - Fórum ainda ativo – novas versões em discussão
- Principais "perigos"
  - Erros causados por sincronização imprópria: detecção difícil!
  - Serialização inesperada pode matar desempenho paralelo
- Mais informações: www.openmp.org