Arquitetura de Computadores III

Arquitetura é a interface entre hardware / software

Conteúdo

- Introdução a programação multi-core com OpenMP
- Organização e hierarquia de memória
- Arquitetura de pipeline escalar de instruções
- Arquitetura de pipeline superescalar de instruções
- Suporte multithreading na arquitetura do processador
- Arquitetura de processadores multi-core
- Arquitetura de Redes-em-Chip (Networks-on-Chip - NoCs)
- Arquitetura de processadores many-core
- Arquitetura de máquinas paralelas

Arquitetura de Computadores III

Parte 1

Programação com OpenMP

O que é OpenMP

- Open specifications for Multi Processing via collaborative work from software/hardware industry, academia and government.

- Portável
  - versões para
    - C/C++
    - e Fortran
  - implementações para UNIX/Linux e Windows

Abordagem SPMD

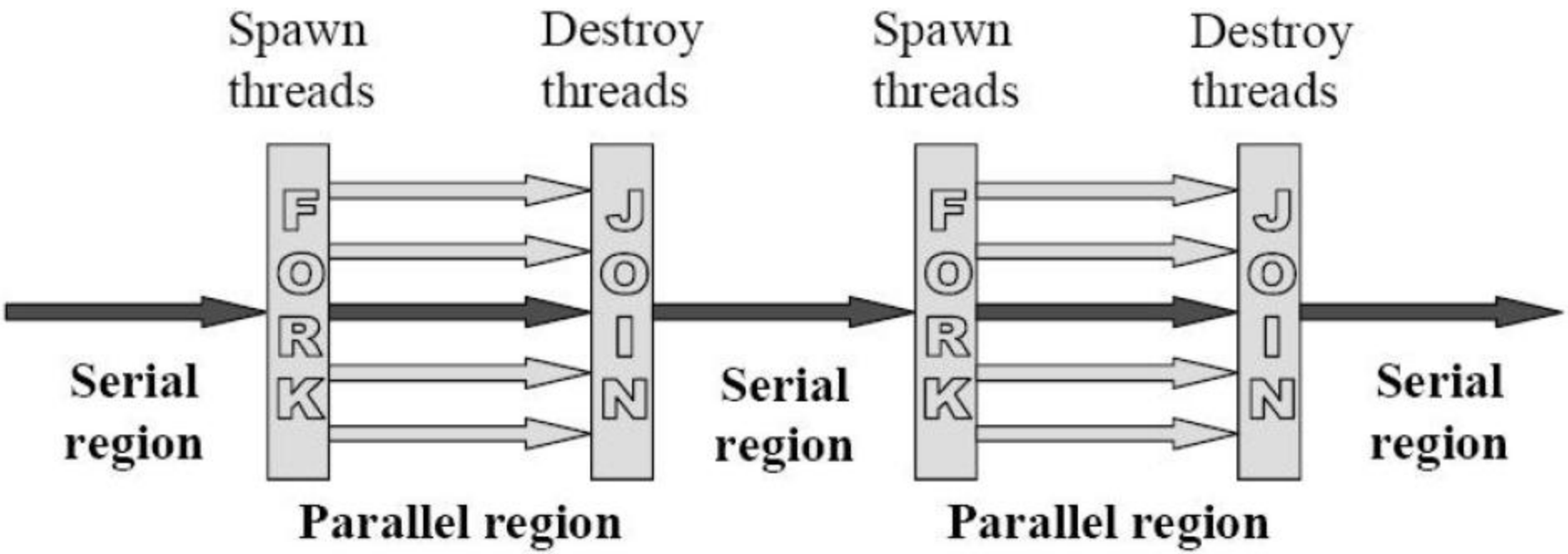
- Uso de anotações (diretivas) para uma linguagem usual.
  - **pragma openmp Whatever you want( )**
  - Uma precompilação permite tornar o código sequencial.
  - Funciona com C, C++, Fortran.

- Single Program Multiple Data
  - Um programa só é executado por todos os processadores.

- Norma de especificação alto-nível para máquina com memória compartilhada.
  - camada mais abstrata em cima de *threads.*

- Paralelismo de laços.
  - O usuário explicita o paralelismo dos laços.
  - Paralelismo tipo “Fork-Join”.

OpenMP – Modelo de Execução



Disponibilidade do OpenMP

- Consórcio de fabricantes de HW/SW:
  - (Compaq), HP, IBM, Intel, SGI, SUN.
  - PGI, KAI, PSR, Absoft
  - DOE, NAG, ASCI, . . .

- Compiladores *OpenMP*
  - PGI
  - Intel
  - gcc

Comunicação entre threads

- Via a memória compartilhada.
  - Possibilidade de definir as variáveis compartilhadas.
- Necessidade de sincronizar os acessos.
  - isso implica em sobrecusto;
  - OpenMP esconde isso ao programador;
  - necessidade de projetar o algoritmo (distribuição dos
  - dados, volume de acessos remotos. . .).

Idéia geral de código c/ OpenMP

```
#include <omp.h>
main ( ) {

int var1, var2, var3;
Código serial
    .
    .
Início da seção paralela, gera um time de threads e especifica o escopo das variáveis
#pragma omp parallel private (var1, var2) shared (var3)
  {
    Seção paralela executada por todas as threads
          .
          .
    Todas as threads se juntam a master thread e param de executar
   }
Volta a executar código serial
       .
       .
}
```

Hello World c/ OpenMP

```
#include <stdio.h>
#include <omp.h>
int main()
{
#pragma omp parallel
{
  printf("Ola Mundo!\n");
}
return 0;
}
```

```
int id;
omp_set_num_threads(2);
#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  printf("Ola Mundo! Aqui é a
    thread %d\n", id);
}
```

Qual a diferença entre os dois códigos?

```
icc -openmp hello.c -o hello
gcc –fopenmp -o hello hello.c
```

Cálculo de Pi com OpenMP

$\pi = 4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \ldots$

```
#include <stdio.h>
#define num_passos 20000000
int main()
{
double pi = 0;
int i;
for (i = 0; i < num_passos; i++)
{
  pi += 4.0 / (4.0 * i + 1.0);
  pi -= 4.0 / (4.0 * i + 3.0);
}
printf("O valor de pi é %f\n", pi);
}
```

```
#pragma omp parallel for
for (i = 0; i < num_passos; i++)
{
  pi += 4.0 / (4.0 * i + 1.0);
  pi -= 4.0 / (4.0 * i + 3.0);
}
```

- O índice i é shared ou private?
- Qual o problema nesta adaptação do trecho de código?

Cálculo de Pi com OpenMP

- O core 1 lê a variável pi da memória, que está configurada como 0.
- O core 2 lê a variável pi da memória, que está configurada como 0.

- O core 1 calcula pi com relação ao índice i=0, e a variável pi agora possui valor 2,667.
- O core 2 calcula pi com relação ao índice i=1, e a variável pi agora possui valor 0,229.

- O core 1 escreve pi na memória, com o valor 2,667.
- O core 2 escreve pi na memória, com o valor 0,229.

Cálculo de Pi com OpenMP

```
#pragma omp parallel for reduction (+:pi)
for (i = 0; i < num_passos; i++)
{
  pi += 4.0 / (4.0 * i + 1.0);
  pi -= 4.0 / (4.0 * i + 3.0);
}
```

- Utilização da diretiva reduction do OpenMP.
  - Cada thread possui seu próprio valor para a variável pi.
  - A operação “+” é aplicada ao final da execução de cada thread.

Multiplicação de Matrizes c/ OpenMP

```
#define SIZE 2
int i, j, k;
int A[2][2] = { {1, 2}, {3, 4} };
int B[2][2] = { {5, 6}, {7, 8} };
int C[2][2];

#pragma omp parallel for
for (i = 0; i < SIZE; i++)
{
  for (j = 0; j < SIZE; j++)
  {
    int soma = 0;
    for (k = 0; k < SIZE; k++)
    {
      soma += A[i][k] * B[k][j];
    }
    C[i][j] = soma;
  }
}
```

Os índices i, j e k são shared ou private?

Qual o problema no uso da diretiva OpenMP no código anterior (ao lado)?

Qual a consequência na alteração realizada abaixo?

```
#pragma omp parallel for private(i,j,k) shared(A,B,C)
for (i = 0; i < SIZE; i++)
{
  for (j = 0; j < SIZE; j++)
  {
    int soma = 0;
    for (k = 0; k < SIZE; k++)
    {
      soma += A[i][k] * B[k][j];
    }
    C[i][j] = soma;
  }
}
```

Por que a variável soma não é marcada como private?

Característica da diretiva private em OpenMP

```
int x = 3;
#pragma omp parallel private (x)
{
  printf("No começo da thread valor de
     x é %d\n", x);
  x = 5;
  printf("No fim da thread valor de x é
     %d\n", x);
}
printf ("O valor final de x é %d\n", x);
```

- Resultado:
  - No começo da thread valor de x é 0
  - No fim da thread valor de x é 5
  - O valor final de x é 3

- Se a diretiva for firstprivate
- Resultado:
  - No começo da thread valor de x é 3
  - No fim da thread valor de x é 5
  - O valor final de x é 3

- Se a diretiva for lastprivate
- Resultado:
  - No começo da thread valor de x é 0
  - No fim da thread valor de x é 5
  - O valor final de x é 5

Sections em OpenMP

- O número de threads que são executadas por região paralela de código é decidido dinamicamente.
- É possível configurar o número de threads:
  - Função: omp_set_num_threads
- Controle de distribuição de tarefas entre as threads:
  - Diretiva sections: distribui blocos de códigos para threads diferentes.

- Exemplo de uso:

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
  executaA();
#pragma omp section
  executaB();
#pragma omp section
  executaC();
}
```

6 threads ➔ 3 serão usadas

2 threads ➔ 1 executa duas seções diferentes.

Sections em OpenMP (aninhamento)

```
 int main()
{
 omp_set_num_threads(3);
 omp_set_nested(1);

#pragma omp parallel
   {
 #pragma omp sections
       {
       #pragma omp section
          executaA();

       #pragma omp section
          executaB();

       #pragma omp section
          executaC();

       } // end parallel and end sections
   }
}
```

```
void executaA() {
     omp_set_num_threads(5);
     #pragma omp parallel
        {
           int tid = omp_get_thread_num();
           #pragma omp for
           for(int i=0; i<5; i++) {
              printf(". %d %d %d\n", i, tid, gtid);
           }
        }
     }
```

Diretiva Barrier para sincronização em OpenMP

```
int th_id;
omp_set_num_threads(3);
#pragma omp parallel private(th_id)
{
  th_id = omp_get_thread_num();
  printf("Executando thread %d\n",
    th_id);
#pragma omp barrier
  printf("Continuando thread %d\n",
    th_id);
}
```

- Resultado
  - Executando thread 0
  - Executando thread 1
  - Executando thread 2
  - Continuando thread 0
  - Continuando thread 2
  - Continuando thread 1

Diretiva Master em OpenMP

```
int th_id;
omp_set_num_threads(3);
#pragma omp parallel private(th_id)
{
  th_id = omp_get_thread_num();
  printf("Iniciando thread %d\n", th_id);
#pragma omp master
  {
    printf("Apenas thread %d aqui\n",
      th_id);
  }
printf("Terminando thread %d\n", th_id);
}
```

- Resultado:
  - Iniciando thread numero 0
  - Apenas thread 0 aqui
  - Iniciando thread numero 2
  - Iniciando thread numero 1
  - Terminando thread numero 1
  - Terminando thread numero 2
  - Terminando thread numero 0

Diretiva Single em OpenMP

```
int th_id;
omp_set_num_threads(3);
#pragma omp parallel private(th_id)
{
th_id = omp_get_thread_num();
printf("Iniciando thread %d\n", th_id);
#pragma omp single
{
printf("Apenas thread %d aqui\n", th_id);
}
printf("Terminando thread %d\n", th_id);
}
```

- Resultado:
  - Iniciando thread 0
  - Apenas thread 0 aqui
  - Iniciando thread 1
  - Iniciando thread 2
  - Terminando thread 0
  - Terminando thread 2
  - Terminando thread 1

Diretiva Critical Section em OpenMP

```
#define MAX_SIZE 100
int A[MAX_SIZE];
int pointer = 0;
int consumir()
{
  int temp = -1;
  #pragma omp critical
  {
    if (pointer > 0)
    {
      --pointer;
      temp = A[pointer];
    }
  }
return temp;
}
```

```
void inserir(int valor)
{
  #pragma omp critical
  {
    if (pointer < MAX_SIZE)
    {
      A[pointer] = valor;
      pointer++;
    }
  }
}
```

Esta diretiva define que somente uma thread pode executar o código por vez.

Exemplo deste slide: Produtor / Consumidor