OpenMP

Slides baseados em tutorial de Tim Mattson da Intel

Modelo OpenMP

- **Como as threads interagem?**
- **OpenMP é um modelo baseado em multithreading**
- **Threads se comunicam pelo compartilhamento de variáveis**
- **Compartilhamento de variáveis pode levar à condição de corrida:**
  - Condição de corrida: quando o resultado do programa muda se as threads são escalonadas de modo diferente
- **Para controlar condições de corrida:**
  - Use sincronização para proteger conflitos de dados
- **Sincronização é cara então:**
  - Tente mudar o modo como os dados são acessados para minimizar a necessidade de sincronização

Sincronização

- Sincronização: trazer todas as threads para um ponto bem definido e conhecido da execução
- Duas formas mais comuns de sincronização:
  - Barreira: cada thread espera na barreira até todas as threads chegarem
  - Exclusão mútua: define um bloco de código que só pode ser executado por uma thread de cada vez

Sincronização:Barreira

- Barreira: Cada thread espera até que todas as threads cheguem

```
#pragma omp parallel
{
int id=omp_get_thread_num();
A[id] = big_calc1(id);
#pragma omp barrier
B[id] = big_calc2(id, A);
}
```

Sincronização: seção crítica

- Exclusão mútua: Somente uma thread pode entrar na seção crítica de cada vez

```
float  res;
#pragma omp parallel
{  float B;int i, id, nthrds;
   id = omp_get_thread_num();
   nthrds = omp_get_num_threads();
   for(i=id;i<niters;i+=nthrds){
        B = big_job(i);
   #pragma omp critical
   res += consume (B);
}
}
```

Sincronização: atomicidade

- Atomicidade provê exclusão mútua mas se aplica somente a uma localização da memória (atualização, leitura, escrita e captura)

```
#pragma omp parallel
{
   double tmp, B;
   B = DOIT();
   tmp = big_ugly(B);
   #pragma omp atomic
   X +=tmp;
}
```

Programa sequencial Pi

```
static long num_steps = 100000;
double step;
void main ()
{  int i; double x, pi, sum = 0.0;
   step = 1.0/(double) num_steps;
   for (i=0;i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Exercício 1

- Modifique o programa “pi” de modo que todas as threads criem uma variável escalar local que armazena a soma parcial e atualizem uma variável compartilhada somando o valor da variável local multiplicado pelo step
- Utilize o pragma critical

Exercício 1

```
#include <omp.h>
static long num_steps = 100000;   double step;
#define NUM_THREADS 2
int main ()
{double   pi;  step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel {
    int i, id,nthrds;double x, sum;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i=id, sum=0.0;i< num_steps; i=i+nthrds){
     x = (i+0.5)*step;
     sum += 4.0/(1.0+x*x);
    }
    #pragma omp critical
     pi += sum * step;
}
}
```

Exercício 2

- Modifique o programa “pi” de modo que todas as threads atualizem uma variável compartilhada somando o valor calculado em um step a esta variável compartilhada
- Utilize o pragma critical

Exercício 2

```
#include <omp.h>
static long num_steps = 100000;   double step;
#define NUM_THREADS 2
int main ()
{double   pi;  step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel {
    int i, id,nthrds;double x;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i=id, sum=0.0;i< num_steps; i=i+nthrds){
     x = (i+0.5)*step;
    #pragma omp critical
     pi += 4.0/(1.0+x*x);
    }
}
pi = pi * step;
}
```

Exercício 3

- Refaça o exercício 1 usando o pragma atomic

Exercício 3

- Refaça o exercício 1 usando o pragma atomic

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
int main ()
{double     pi;   step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel {
     int i, id,nthrds;double x, sum;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     for (i=id, sum=0.0;i< num_steps; i=i+nthrds){
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
     }
     #pragma omp atomic
     pi += sum * step;
}
}
```

SPMD vs. Divisão de trabalho

- Uma construção paralela cria um programa “Single Program Multiple Data” pois cada thread executa o mesmo código
- Como fazer com que threads executem partes diferentes do código ?
- Construção de laço

Divisão de trabalho por laço

- Uma construção de laço para divisão de trabalho divide as iterações do laço entre as threads

```
#pragma omp parallel
{
#pragma omp for
for (I=0;I<N;I++){
NEAT_STUFF(I);
}
}
```

- A variável I é privativa a cada thread por default. Pode também usar a cláusula private(I)

Divisão de trabalho por laço

- Código sequencial

```
for(i=0;i<N;i++){ a[i] = a[i] + b[i];}
```

Divisão de trabalho por laço

- Código sequencial

```
for(i=0;i<N;i++){ a[i] = a[i] + b[i];}
```

- Código com região paralela

```
#pragma omp parallel
{
int id, i, Nthrds, istart, iend;
id = omp_get_thread_num();
Nthrds = omp_get_num_threads();
istart = id * N / Nthrds;
iend = (id+1) * N / Nthrds;
if (id == Nthrds-1) iend = N;
for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

Divisão de trabalho por laço

- Código sequencial

```
for(i=0;i<N;i++){ a[i] = a[i] + b[i];}
```

- Código com região paralela

```
#pragma omp parallel
{
int id, i, Nthrds, istart, iend;
id = omp_get_thread_num();
Nthrds = omp_get_num_threads();
istart = id * N / Nthrds;
iend = (id+1) * N / Nthrds;
if (id == Nthrds-1) iend = N;
for(i=istart;i<iend;i++){ a[i] = a[i] + b[i];}
}
```

- Código com região paralela e com divisão de trabalho por laço

```
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

Cláusula schedule

- A cláusula `schedule` afeta como as iterações do laço são mapeadas nas threads
  - `schedule(static [,chunk])`
    - Reparte blocos de iterações de tamanho "chunk" para cada thread
  - `schedule(dynamic[,chunk])`
    - Cada thread apanha um bloco de "chunk" iterações de uma fila até que as iterações acabem
  - `schedule(guided[,chunk])`
    - As threads apanham dinamicamente blocos de iterações que começam grandes e vão diminuindo até o tamanho "chunk" à medida que a execução procede.
  - `schedule(runtime)`
    - O tipo de escalonamento e o tamanho do bloco são obtidos da variável de ambiente OMP_SCHEDULE ou da biblioteca de execução.
  - `schedule(auto)`
    - O escalonamento é feito pelo sistema de execução e não precisa ser nenhum dos acima

Combinação de região paralela e construção de laço

```
double  res[MAX];  int i;
#pragma omp parallel
{
#pragma omp for
for (i=0;i< MAX; i++) {
res[i] = huge();
}
}
```

```
double  res[MAX];  int i;
#pragma omp parallel for
for (i=0;i< MAX; i++) {
res[i] = huge();
}
```

Redução

```
double  ave=0.0, A[MAX];  int i;
for (i=0;i< MAX; i++) {
ave + = A[i];
}
ave = ave/MAX;
```

- Os valores estão sendo acumulados em uma variável que é uma situação muito comum de acontecer e se chama redução
- A maioria dos ambientes de programação paralela incluem operações de redução

Redução

- Claúsula de redução: `reduction (op : list)`
- Pode ser usada em uma região paralela ou de divisão de trabalho:
  - Uma cópia local das variáveis em `list` é feita e inicializada dependendo da operação `“op”` (0 para “+”).
- Atualizações são feitas nas cópias locais
- Cópias locais são reduzidas em um único valor utilizando a operação `“op”` e combinadas com os valores originais das variáveis globais em `list`
- As variáveis na lista `“list”` devem ser compartilhadas na região paralela

Operações de Redução e valores iniciais

Operador	Valor inicial
+	0
*	1
-	0
min	Maior número positivo
max	Número mais negativo

C/C++ only	
Operator	Valor inicial
&	~0
|	0
^	0
&&	1
||	0

Exercício 4

•Paralelize o programa sequencial PI com uma construção de laço modificando o mínimo possível o programa sequencial

```
static long num_steps = 100000;
double step;
void main ()
{      int i; double x, pi, sum = 0.0;
       step = 1.0/(double) num_steps;
       for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
       }
       pi = step * sum;
}
```

Exercício 4

```
#include <omp.h>
static long num_steps = 100000;
double step;
int main ()
{               int i; double x, pi, sum = 0.0;
                step = 1.0/(double) num_steps;
                #pragma omp parallel
                {
                        double x;
                        #pragma omp for
reduction(+:sum)
                        for (i=0;i< num_steps; i++){
                                x = (i+0.5)*step;
                                sum = sum +
4.0/(1.0+x*x);
                        }
                }
                pi = step * sum;
}
```