SSC-0742
PROGRAMAÇÃO CONCORRENTE

Aula 08 – Ferramentas de Apoio ao Desenvolvimento de Aplicações Concorrentes - OpenMP

Prof. Jó Ueyama

Créditos

Os slides integrantes deste material foram construídos a partir dos conteúdos relacionados às referências bibliográficas descritas neste documento

Visão Geral da Aula de Hoje

1. O que é OpenMP
2. Modelo de Programação
3. Modelo de Execução
4. diretivas de Compilação
7. Exercício e Leitura Recomendada

O QUE É OPENMP?

OpenMP

- **Significado**
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia
  - É um modelo de programação de memória compartilhada que nasceu da cooperação entre um grupo de grandes fabricantes de software e hardware
  - API para programação paralela de arquiteturas multiprocessor. Foi definida inicialmente para ser usada em programas C / C++ (1998) e Fortran (1997) tanto Linux quanto Windows
  - Não é uma implementação e sim uma **especificação**
- **Os principais compiladores suportam diretivas do OpenMP para a maioria das plataformas**

OpenMP

- **Objetivos**
  - Ser o padrão de programação para arquiteturas de memória compartilhada
  - Estabelecer um conjunto simples e limitado de diretivas de programação
  - Possibilitar a paralelização incremental de programas sequenciais
  - Permitir implementações eficientes em problemas de granularidade fina, média e grossa

OpenMP

- **Componentes**
  - Bibliotecas de Funções
  - Diretivas de Compilação
  - Variáveis de Ambiente

OpenMP

- **Memória Compartilhada X Passagem de Mensagens**
  - Programação em memória compartilhada é mais intuitivo para desenvolvedores que desejam paralelizar seus programas
    - Infelizmente como opera sobre a premissa de espaço de endereçamento compartilhado, é suscetível às limitações de escala inerente dos mecanismos que geram essa visão compartilhada da memória
  - Passagem de mensagem por sua vez assume que não há memória compartilhada e que todas as comunicações devem ser realizadas por meio do envio de mensagens explícitas
    - Não é intuitivo, pois exige que o programador gerencia o envio, recebimento e coordenação entre os processos paralelos

OpenMP

- **Alternativas ao OpenMP**
  - Unified Parallel C
  - Co-Array Fortran
  - Posix Threads
  - Gnu Portable Threads

OpenMP

- **Limitações do OpenMP**
  - Quanto mais variáveis compartilhadas entre as threads, mais difícil deve ser o trabalho para garantir que a visão das variáveis são atuais – coerência de cache
  - Em algum ponto, a sobrecarga associada com a garantia de coerência de cache fará com que a paralelização não seja escalável para mais processadores
  - Outras limitações são baseadas em hardware e está associada com a forma como muitos SMPs são fisicamente capazes de compartilhar o mesmo espaço de memória

OpenMP

- **Limitações do OpenMP**
  - A largura de banda da memória não é escalável a medida que mais processadores são introduzidos, ou seja a escalabilidade é limitada pela arquitetura de memória
  - A sincronização de um subconjunto de threads não é permitida

OpenMP

- O aspecto mais difícil de criar um programa de memória compartilhada é traduzir o que se deseja para uma versão multi-thread
- Outro ponto é fazer a versão muilti-thread do programa operar corretamente
  - **Sem problemas com variáveis compartilhadas**
  - **Situações de condições de corrida/disputa**
  - **Detecção e Depuração dessas condições**
  - **Questões relativas ao tempo de execução**

OpenMP

- Exemplo – **Hello World!**

```
#include <omp.h>
main () {
  int nthreads, tid; // Fork team-threads, cada uma cópia das variáveis.

#pragma omp parallel private(nthreads, tid)
{                                          // Obtem e escreve thread-id
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
                                           // apenas o master thread faz isto
  if (tid == 0){
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
  }
} /* Todos os theards juntam-se no master
}
```

OpenMP

```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

Code inserted by the OpenMP compiler

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (........, internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;

    // parallel segment

}
```

Corresponding Pthreads translation

MODELO DE PROGRAMAÇÃO

Modelos de Programação

- **Paralelismo Explícito**
  - Cabe ao programador anotar as tarefas para execução em paralelo e definir os pontos de sincronização. A anotação é feita por utilização de diretivas de compilação embarcadas no código do programa

Modelos de Programação

- **Multithread Implícito**
  - Neste caso um processo é visto em conjunto de threads que se comunicam por meio da utilização de variáveis compartilhadas
    - Criação
    - Início
    - Término
  - Feito de forma implícita pelo ambiente de execução sem que o programador tenha que se preocupar com tais detalhes

Modelos de Programação

- **Multithread Implícito**
  - Espaço de endereçamento global compartilhado entre as threads
  - Variáveis podem ser compartilhadas ou privadas para cada thread
  - Controle, manuseio e sincronização das variáveis envolvidas nas tarefas paralela é transparente ao programador

MODELO DE EXECUÇÃO

Modelos de Execução

- **Fork-Join**
  - Todos os programas iniciam sua execução com um processo mestre
  - O master thread executa sequencialmente até encontrar um construtor paralelo, momento em que cria um team de threads
  - O código delimitado pelo construtor paralelo é executado em paralelo pelo master thread e pelo team de threads
  - Quando completa a execução paralela, o team de threads sincroniza em uma barreira implícita com o master thread
  - O team de threads termina a sua execução e o master thread continua a execução sequencialmente até encontrar um novo construtor paralelo
  - A sincronização é utilizada para evitar a competição entre as threads

Modelos de Execução

- Estrutura Básica de um Programa OpenMP

```
#include <omp.h>  // incluir a biblioteca de funções OpenMP

...

main() {

  ...  // região sequencial executada apenas pelo master thread

  #pragma omp parallel  // construtor paralelo do OpenMP
  {    // master thread cria um team of threads

  ...  // região paralela executada por todos os threads

  }    // team of threads sincroniza com o master thread e termina

  ...  // região sequencial executada apenas pelo master thread

}
```

Modelos de Execução

- Definições da biblioteca em **omp.h**
- Implementação em **libgomp.so**
- Compilação de um programa OpemMP
  - É preciso utilizar a opção
    - **-fopenmp**

VARIÁVEIS

Variáveis

- Declarar acesso entre as threads
  - **Private**
    - Pertencem e são conhecidas apenas a cada segmento específico

```
int id, nthreads;
pragma omp parallel private(id)
{id = omp_get_thread_num();
```

  - **Shared**
    - Variáveis compartilhadas são conhecidas por todas as threads. Cuidados devem ser tomados quando utilizar essas variáveis, já que a manipulação incorreta poderá dificultar a detecção de erros como condições de disputa e deadlocks

```
int id, nthreads, A, B; A = getA();
B = getB();
#pragma omp parallel private(id,nthreads) shared(A,B)
{ id = omp_get_thread_num();
```

Variáveis

- Inicialização de Variáveis
  - **Firstprivate**
    - Permite inicializar uma variável privada na thread master antes de entrar em uma região paralela do código. Caso contrátio, todas as variáveis privadas serão consideradas não inicializadas no início da thread

```
int id, myPi; myPi = 3.1459
#pragma omp parallel private(id) firstprivate(myPi)
{ id = omp_get_thread_num();
```

OPERAÇÕES DE REDUÇÃO

Redução

- Permitem a uma variável utilizada privativamente para cada thread, ser agregada em um único valor

  reduce(*:i) # for N threads, get product of i1*i2*i3*...*iN

  reduce(+:i) # for N threads, get the sum of i1+i2+i3+...+iN

- **Operações de Redução – C/C++**

  Arithmetic: + - * / # add, subtract, multiply, divide

  Bitwise: & | ^ # and, or, xor

  Logical: && || # and, or

Redução

- O exemplo abaixo atribui um valor para a variável privada i. No final do bloco paralelo multiplica (*) i de todas as threads juntas e torna o produto final disponível para a thread master

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
  int i;
  #pragma omp parallel reduction(*:i)
  {
    i=omp_get_num_threads();
  }
  printf("ans=%d\n",c);
 return 0;
}
```

ROTINAS E VARIÁVEIS DE AMBIENTE

Rotinas

- **omp_get_num_threads**
- **omp_get_num_procs**
- **omp_set_num_threads**
- **omp_get_max_threads**
- **omp_in_parallel**

Variáveis de Ambiente

- **OMP_NUM_THREADS**
  - Informa a execução do número de threads para ser utilizada
- **OMP_SCHEDULE**
  - Esta variável de ambiente aplica-se ao PARALLEL DO e diretivas de trabalho compartilhado que tem o tipo de escalonamento RUNTIME
- **OMP_DYNAMIC**
  - Habilita/Desabilita ajustes dinâmicos do número de threads disponíveis para a execução de regiões paralelas
- **OMP_NESTED**
  - Habilita/Desabilita o paralelismo aninhado (nested)

DIRETIVAS DE COMPILAÇÃO

diretivas de Compilação

- Um programa simples em OpenMP

Programa sequencial

```
void main()
{
   double r[1000];

   for (int i=0; i<1000; i++) {
     large_computation(r[i]);
   }
}
```

Programa paralelo

```
void main()
{
   double r[1000];

#pragma omp parallel for
   for (int i=0; i<1000; i++) {
     large_computation(r[i]);
   }
}
```

diretivas de Compilação

- Ênfase na paralelização de ciclos
- As diretivas de compilação ou pragmas identificam os pontos e formas de paralelização

diretivas de Compilação

- #pragma omp directive-name [clause, …] newline
  - directive-name
    - Uma diretiva OpenMP válida.
    - Tem que aparecer depois do pragma e antes das cláusulas
  - [clause, …]
    - Opcional. Pode aparecer em qualquer ordem e se necessário repetida
  - newline
    - Obrigatório. Seguido do bloco estruturado que vai ser executado em paralelo
- Exemplo:
  - **#pragma omp parallel default(shared) private(beta,pi)**

Cláusulas

Cláusula	Descrição	Utilizado nas Diretivas
shared(lista)	Define as variáveis que serão compartilhadas entre os *threads* dentro da região paralela	parallel ,parallel for ,parallel sections
private(lista)	Indica as variáveis que cada *thread* terá acesso a uma cópia	parallel, for, sections, single, parallel, for, parallel sections
firstprivate(lista)	Como a "private", porem é inicializado com o valor da variável antes da região paralela	parallel, for, sections, single, parallel for, parallel sections
lastprivate(lista)	Semelhante a "private", mas na saída do bloco paralelo, a variável manterá o último valor atribuído	for, sections, parallel for, parallel sections
default(shared)	Define que todas as variáveis da região paralela serão compartilhadas, exceto as listadas com cláusulas como "private"	parallel ,parallel for ,parallel sections
default(none)	O programador deve especificar qual será o comportamento de cada variável	parallel ,parallel for ,parallel sections
reduction (operador:lista)	As varáveis são "private", e na saída do bloco é realizada a operação especificada, com as variáveis de cada *thread* (ver tabela 3)	parallel, for, sections, parallel for, parallel sections
copy(lista)	Define que copias de cada *threads* das variáveis definidas como "*thread*private" sejam iniciadas com o mesmo valor	parallel ,parallel for ,parallel sections
copyprivate(lista)	Transmite o valor de uma variável "private" de um *thread* para outros *threads* do grupo	single
nowait	Faz com que as barreiras implícitas existentes no final de certas diretivas sejam ignoradas	for, sections, single, parallel for, parallel sections
schedule (tipo,[tamanho])	Controla a forma como se distribui as iterações entre os *threads*	for, parallel for
if(expressão)	Indica uma condição na qual o bloco será executado de forma paralela ou seqüencial	parallel ,parallel for ,parallel sections
ordered	Utilizada nos construtores for, em que há a diretiva ordered	for, parallel for
num_*threads*(int)	Indica quantos *threads* irá executar na região paralela da diretiva em questão	parallel ,parallel for ,parallel sections

diretivas de Compilação

- Compilação Condicional (_OPENMP)

```
#ifdef _OPENMP
    bloco_código;
#endif
```

- Exemplo de código:
  - ***printf("%d processadores livres\n", omp_get_num_procs());***
- Possibilita instrumentalizar o código

diretivas de Compilação

Sintaxe parallel

- **#pragma omp parallel `clause' bloco_codigo;**
  - Indica que o bloco_codigo é para ser executado em paralelo
- Clause pode ser:
  - **if(exp)**
  - **private(list)**
  - **firstprivate(list)**
  - **num_threads(int_exp)**
  - **shared(list)**
  - **default(shared|none)**
  - **copyin(list)**
  - **reduction(operator: list)**

diretivas de Compilação

- Cláusulas private, shared, default e firstprivate permite ao programador controlar as variáveis na região paralela
- **private(list)**
  - Variávies da lista ficam privadas a cada thread do team de threads
  - Não são inicializadas
- **firstprivate(list)**
  - Permite que as variáveis privadas sejam inicializadas
- **shared(list)**
  - As variáveis da lista são compartilhadas por todas a threads
  - Por padrão as variáveis são "**shared**"

diretivas de Compilação

- **Quantidade de threads?**
  - **Número de threads determinado pelos fatores (ordem de precedência):**
    - **Cláusula num_threads**
    - **Função omp_set_num_threads()**
    - **Variável de ambiente OMP_NUM_THREADS**
    - **default – depende da implementação**
  - Os threas são numerados de 0 a N-1
  - Por padrão, um programa com várias regiões em paralelo vai utilizar o mesmo número de threads para cada região, ao menos que isso seja redefinido

diretivas de Compilação

- Outras cláusulas
  - **reduction**
    - Permite operar sobre as variáveis da lista
  - **copyin(list)**
    - Permite a atribuição do mesmo valor a variáveis THREADPRIVATE
  - **if(exp)**
    - Necessita avaliar como verdadeiro para que o team de threads seja criado, senão a execução será sequencial

Diretivas de Compilação

Diretivas	Descrição
parallel	Define a região paralela, portanto o código que será executado por múltiplos *threads*
For	Faz com que um laço "for", dentro da região paralela, seja dividido entre os *threads*
parallel for	Construção curta para regiões paralelas que terá apenas um laço que deve ser dividido entre os *threads*
single	Este construtor indica uma região que deve ser executada por penas um *thread*
selections	Indica uma região, dentro da região paralela, que possui blocos de execução independentes, que devem ser executados por um *thread* cada
selection	Indica o bloco que deve ser executado dentro de uma "selections"
parallel selection	Construção curta para regiões paralelas que terá blocos independentes que devem ser executados por apenas um *thread* cada
critical	Limita a execução de uma região, para apenas um *thread* por vez, no intuito de evitar condição de corrida
atomic	Mesmo principio que o "critical" porem o compilador tenta otimizar caso seja instruções simples, caso contrario ficará como o "critical"
barrier	Sincroniza todos os *threads* em um ponto do código, quando todos atingem este ponto eles voltam a executar ao mesmo tempo
flush[(lista)]	Utilizado para garantir que variáveis compartilhadas estejam com o valor atualizado na memória principal em determinada posição do programa
ordered	Permite que um laço seja executado na ordem que seria executado de forma seqüencial
master	Semelhante ao "single", porem é executado obrigatoriamente pelo *thread* mestre e não possui barreira implícita
*thread*private (lista)	Define quais variáveis serão privadas em todas as regiões paralelas

Construtores de Work-Sharing

Construtores de Work-Sharing

- Determinam regras de divisão de trabalho entre as threads (não cria os threads)
- Tipos
  - **for**
    - Divide as iterações de um ciclo pelo threads da team (paralelismo de dados)
  - **sections**
    - Responsável por dividir o trabalho em seções discretas, distintas e que são executadas pelas threads. Pode ser utilizado para paralelismo funcional
  - **single**
    - Responsável por serializar o código

Construtores de Work-Sharing

Sintaxe for

- **#pragma omp for `clause' {ciclo_for(); };**
- Clause pode ser:
  - **private(list)**
  - **firstprivate(list)**
  - **lastprivate(list)**
  - **reduction(operator: list)**
  - **ordered**
  - **schedule(type)**
  - **Nowait**
- **O compilador distribui as iterações pelas threads**

Construtores de Work-Sharing

- Exemplo – For

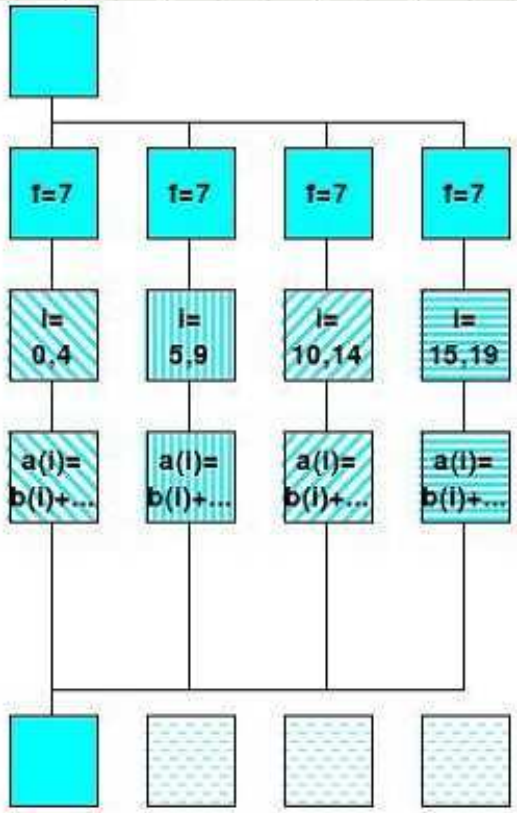
C / C++:

```
#pragma omp parallel private(f)
{
    f=7;

#pragma omp for
    for (i=0; i<20; i++)
        a[i] = b[i] + f * (i+1);



} /* omp end parallel */
```



Construtores de Work-Sharing

- **Claúsula Schedule**
  - Dividir as iterações do ciclo pela threads:
    - **Estático**
      - As iterações são agrupadas em conjuntos (chunks) e atribuídas às threads de forma estática
    - **Dinâmico**
      - As iterações são agrupadas em conjuntos e são dinamicamente distribuídas pelas threads. Quando uma termina recebe dinamincamente outro conjunto e prossegue o trabalho
    - **Guiado**
      - Indica o número mínimo de iterações a agrupar em uma tarefa
    - **Em tempo de execução**
      - Decisão realizada em tempo de execução a partir da variável **OMP_SCHEDULE**

Construtores de Work-Sharing

- Exemplo de uso da diretiva **for**

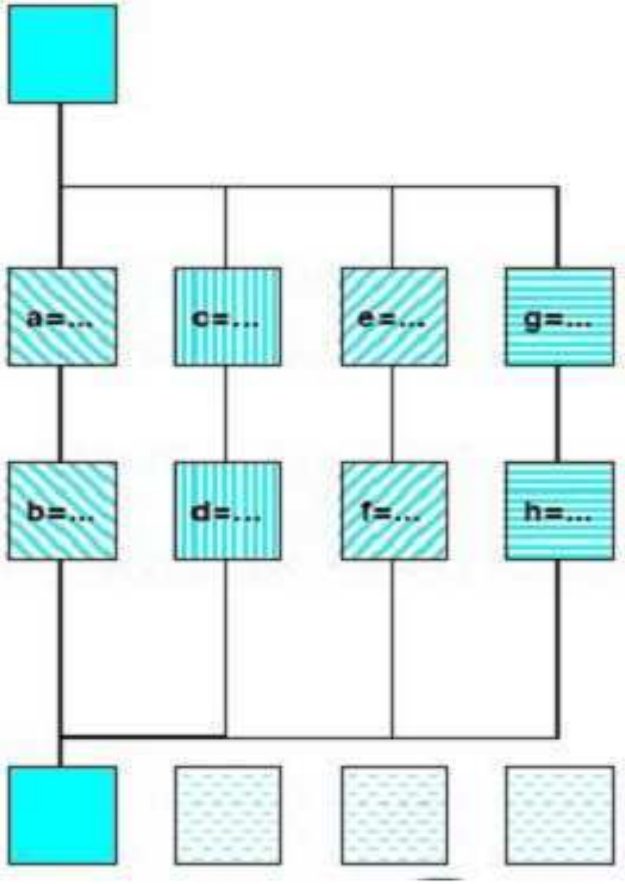
```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main() {
  int i, chunk;
  float a[N], b[N], c[N];
  /* Algumas inicializacoes */
  for (i=0; i < N; i++)
     a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i) {
#pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
     c[i] = a[i] + b[i];
}  // fim da secção paralela/
}
```

Construtores de Work-Sharing

- **Sections**

C / C++:

```
#pragma omp parallel
{
#pragma omp sections
  {{  a=...;
      b=...; }
#pragma omp section
    {  c=...;
       d=...; }
#pragma omp section
    {  e=...;
       f=...; }
#pragma omp section
    {  g=...;
       h=...; }
  } /*omp end sections*/
} /*omp end parallel*/
```



Construtores de Work-Sharing

Sintaxe sections

```
#pragma omp section `clause'
{       #pragma omp section newline
        codigo();
        #pragma omp section newline
        codigo();
};
```

- Clause pode ser:
  - **private(list)**
  - **firstprivate(list)**
  - **lastprivate(list)**
  - **reduction(operator: list)**
  - **Nowait**

Construtores de Work-Sharing

- Exemplo – **Sections**

```
#include <omp.h>
#define N 1000
main() {
  int i, chunk;
  float a[N], b[N], c[N];
  // Some initializations
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i) {
#pragma omp sections nowait {
#pragma omp section
  for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
#pragma omp section
  for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
} // fim de secções
}
```

CONSTRUTORES DE SINCRONIZAÇÃO

Construtores de Sincronização

- É necessário sincronizar ações em variáveis compartilhadas
- E também a ordenação correta de leituras e escritas
- Também é importante proteger a atualização de variáveis compartilhadas (não atômicas por padrão)

Construtores de Sincronização

- O OpenMP possui alguns construtores de sincronização:
  - **omp master**
    - Especifica uma região que será executada apenas pelo master
  - **omp critical**
    - Especifica uma região crítica de código que deve ser executada apenas por um thread de cada vez
  - **omp barrier**
    - Quando esta diretiva é alcançada por uma thread, esta espera até que as restantes cheguem ao mesmo ponto. Nenhuma thread pode prosseguir além da barreira.
    - Nenhuma ou todas as threads devem encontar a barreira. Caso contrátio → **Deadlock**

Construtores de Sincronização

- **Continuação…**
  - **omp atomic**
    - Especifica um endereço de memória para atualização atômica. Aplica-se apenas a uma única sentença
  - **omp flush**
    - Identifica um ponto de sincronização em que é necessário providenciar uma visão consistente da memória
  - **omp ordered**
    - As iterações deve ser executadas na mesma ordem, como se fossem executadas sequencialmente

Construtores de Sincronização

- **Seções Críticas**
  - Uma seção crítica é um bloco de código que somente pode ser executado uma thread por vez
  - Pode ser utilizado para proteger a atualização de variáveis compartilhadas
  - A diretiva CRITICAL permite que as seções críticas recebam nomes
  - Caso uma thread esteja numa seção crítica com um determinado nome, nenhuma outra thread pode estar na seção crítica com o mesmo nome (elas podem estara em seções criíticas com outros nomes)

Construtores de Sincronização

- **Sintaxe**

  #pragma omp critical [(name)]

  bloco

- Se o nome é omitido, um nome nulo é assumido (todas as seções críticas sem nome tem efetivamente o mesmo nome)

Construtores de Sincronização

- **Sintaxe**

  #pragma omp atomic

  statement

- Onde statement pode ter uma das seguintes formas:

  –X = binop = expr, x++, ++x, x– ou –x

  –Binop é : +, *, -, /, &, ^, <<, ou >>

Construtores de Sincronização

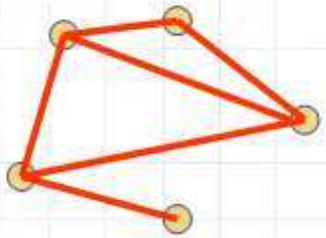
- **Continuação…**
- A avaliação de expr não é atômica
- Pode ser mais eficiente que utilizar diretivas **CRITICAL**, se
  - Diferentes elementos do arranjo de dados podem ser protegidos separadamente

Construtores de Sincronização

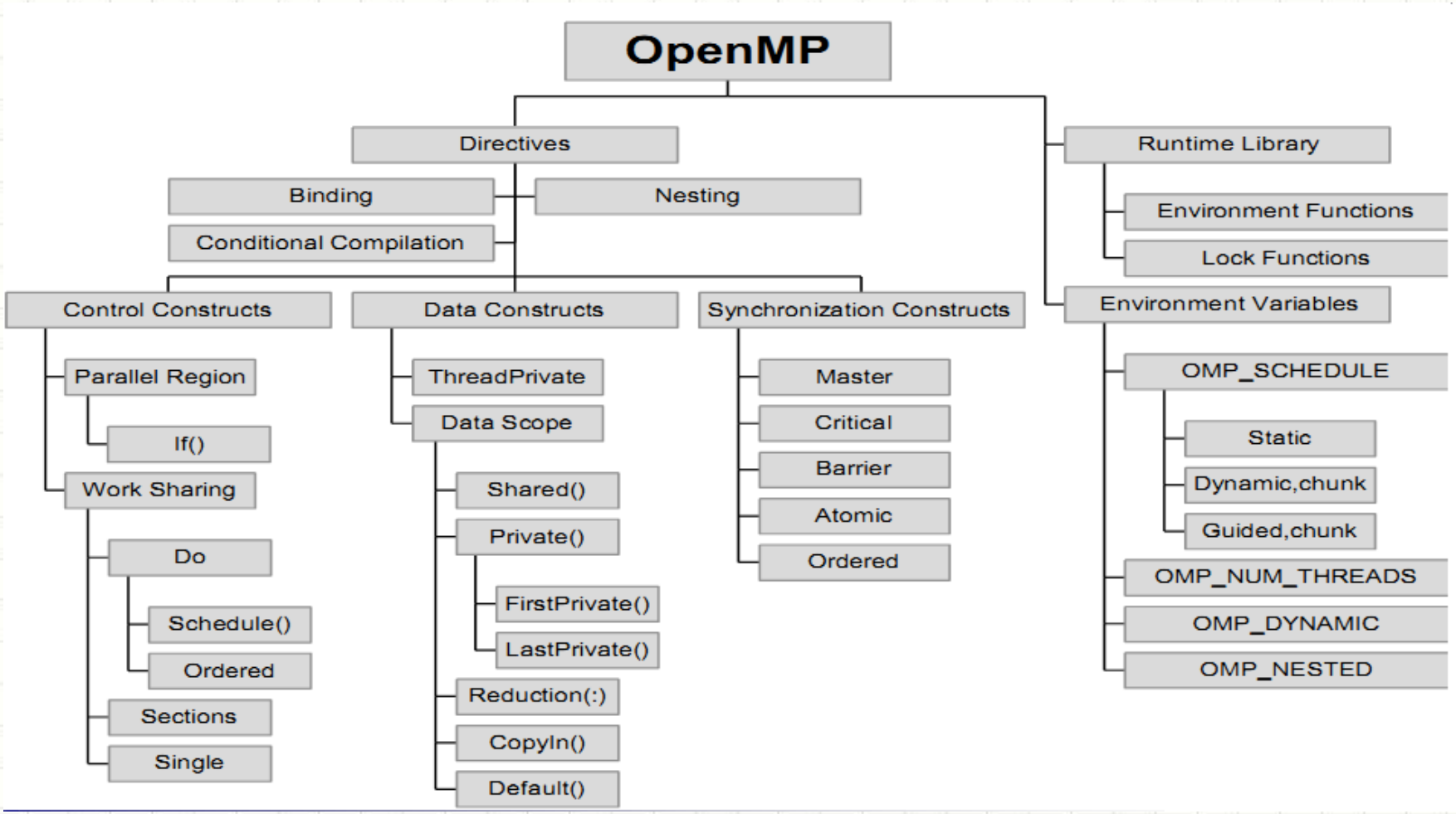
- Diretiva **ATOMIC**

Exemplo (computar o grau de cada vértice em um grafo):

```
#pragma omp parallel for
    for (j=0; j<nedges; j++){
#pragma omp atomic
      degree[edge[j].vertex1]++;
#pragma omp atomic
      degree[edge[j].vertex2]++;
    }
```



Visão Geral do OpenMP



Funções de Sincronização

Função	Descrição
void omp_set_num_*threads* (int num*threads*)	Insere o número padrão de *threads* a ser usado em regiões paralela, tem maior precedência que OMP_NUM_*THREADS* e menor que num_*threads*(int)
int omp_get_num_*threads*()	Retorna o número de *threads* ativo na região paralela onde a função foi chamada
int omp_get_max_*threads*()	Retorna o número máximo de *threads* que o programa pode utilizar, podendo ser usada na região paralela e na seqüencial
int omp_get_*thread*_num()	Retorna um número inteiro referente ao identificador do *thread* que está executando, o *thread* mestre é 0
int omp_get_num_procs()	Retorna o número de processadores disponiveis para execução do programa
int omp_in_parallel()	Retorna valor diferente de zero quando chamada em região paralela, e zero caso contrário
void omp_set_dynamic(int dynamic_*threads*)	Habilita (≠0) ou desabilita (0) o ajuste dinâmico do número de *threads* para as regiões paralelas
int omp_get_dynamic()	Retorna um valor diferente de zero se o ajuste dinâmico estiver habilitado e zero se estiver desabilitado
void omp_set_nested(int nested)	Habilita ou desabilita o paralelismo aninhado
int omp_get_nested()	Retorna um valor diferente de zero se o paralelismo aninhado estiver habilitado e zero caso contrário

Funções de Controle

Função	Descrição
void omp_init_lock (omp_lock_t *lock)	Inicia as variáveis de bloqueio, indicando para os *threads* as variáveis de bloqueio
void omp_init_nest_lock (omp_nest_lock_t *lock)	Realiza a mesma função que omp_init_lock, para variáveis aninhadas
void omp_destroy_lock (omp_lock_t *lock)	Finaliza qualquer associação de bloqueio a uma determinada variável
void omp_destroy_nest_lock (omp_nest_lock_t *lock)	Realiza a mesma função que omp_destroy_lock, para variáveis aninhadas
void omp_set_lock (omp_lock_t *lock)	Solicita o bloqueio de uma variável ou aguarda que uma variável bloqueada esteja disponível
void omp_set_nest_lock (omp_nest_lock_t *lock)	Realiza a mesma função que omp_set_lock, para variáveis aninhadas
void omp_unset_lock (omp_lock_t *lock)	Libera o bloqueio de uma determinada variável
void omp_unset_nest_lock (omp_nest_lock_t *lock)	Realiza a mesma função que omp_unset_lock, para variáveis aninhadas
void omp_test_lock (omp_lock_t *lock)	Testa e bloqueia uma variável, se ela não estiver bloqueada, mas não bloqueia a *thread* caso a variável esteja bloqueada
void omp_test_nest_lock (omp_nest_lock_t *lock)	Realiza a mesma função que omp_test_lock, para variáveis aninhadas

EXERCÍCIO E LEITURA RECOMENDADA

Exercício

- Acessar o Moodle

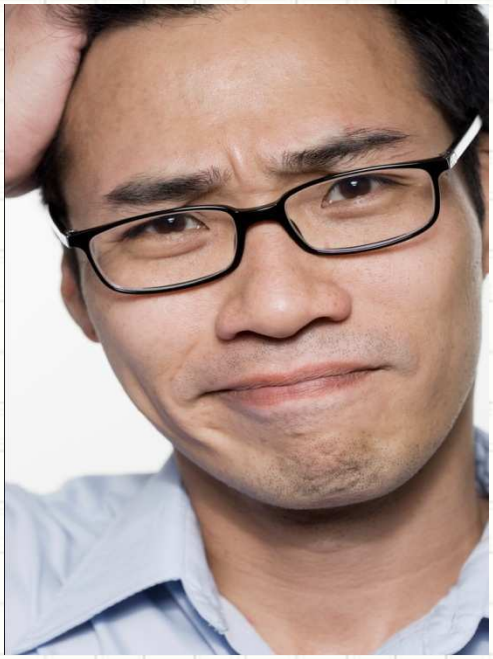
Bibliografia

- Introduction to Parallel Computing, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar - 2ª ed., Addison Wesley
- OpenMP
  - https://computing.llnl.gov/tutorials/openMP/
- Programação Paralela e Distribuída
  - http://www.dcc.fc.up.pt/~ricroc/aulas/1011/ppd/apontamentos/openmp.pdf
- Programação Paralela e Distribuída
  - http://www.dcc.fc.up.pt/~fds/aulas/PPD/0708/
- NetLab – Computação Paralela
  - http://netlab.ulusofona.pt/cp/

Bibliografia

- Programação Paralela
  - http://www.dcc.ufrj.br/~gabriel/progpar/OpenMP.pdf

Dúvidas



Próxima Aula...

- OpenMP Passo-a-Passo – Conceitos Básicos