

Bacharelado em Ciência da Computação

DCC-IM / UFRJ

Programação Paralela e Distribuída - OpenMP

Um curso prático

Gabriel P. Silva

Roteiro

- ◆ Introdução ao OpenMP
- ◆ Regiões Paralelas
- ◆ Diretivas de Compartilhamento de Trabalho
- ◆ Laços Paralelos
- ◆ Sincronização
- ◆ OpenMP 2.0

Introdução ao OpenMP

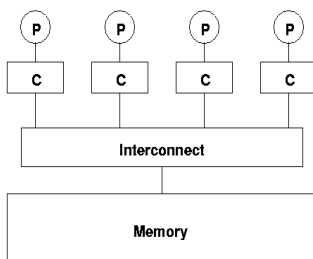
Breve História do OpenMP

- ◆ **Existe uma falta histórica de padronização nas diretivas para compartilhamento de memória. Cada fabricante fazia a sua própria.**
- ◆ **O fórum OpenMP foi iniciado pela Digital, IBM, Intel, KAI e SGI. Agora inclui todos os grandes fabricantes.**
- ◆ **O padrão OpenMP para Fortran foi liberado em Outubro de 1997. A versão 2.0 foi liberada em Novembro de 2000.**
- ◆ **O padrão OpenMP C/C++ foi liberado em Outubro de 1998. A versão 2.0 foi liberada em Março de 2002.**

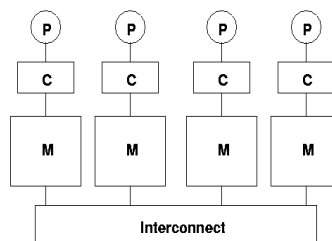
Sistemas de Memória Compartilhada

- ◆ O OpenMP foi projetado para a programação de computadores paralelos com memória compartilhada.
- ◆ A facilidade principal é a existência de um único espaço de endereçamento através de todo o sistema de memória.
 - Cada processador pode ler e escrever em todas as posições de memória.
 - Um espaço único de memória
- ◆ Dois tipos de arquitetura:
 - Memória Compartilhada Centralizada
 - Memória Compartilhada Distribuída

Sistemas de Memória Compartilhada



True shared memory



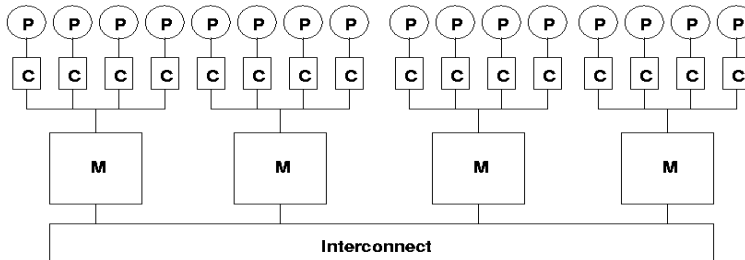
Distributed shared memory

Sun Enterprise/SunFire, Cray SV1, Compaq ES, multiprocessor PCs, nodes of IBM SP, NEC SX5

?

Sistemas de Memória Compartilhada

- ◆ A maioria dos sistemas de memória compartilhada distribuída são *clusters*:



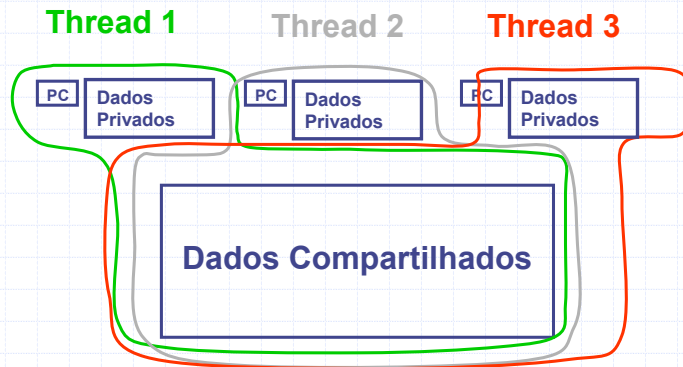
Clustered distributed shared memory

SGI Origin, HP Superdome, Compaq GS,
Earth Simulator, ASCI White

Threads

- ◆ Uma *thread* é um processo "peso leve".
- ◆ Cada *thread* pode ser seu próprio fluxo de controle em um programa.
- ◆ As *threads* podem compartilhar dados com outras *threads*, mas também têm dados privados.
- ◆ As *threads* se comunicam através de uma área de dados compartilhada.
- ◆ Uma equipe de *threads* é um conjunto de *threads* que cooperam em uma tarefa.
- ◆ A "*thread master*" é responsável pela coordenação da equipe de *threads*.

Threads



Diretivas e Sentinelas

- ◆ Uma diretiva é uma linha especial de código fonte com significado especial apenas para determinados compiladores.
- ◆ Uma diretiva se distingue pela existência de uma sentinela no começo da linha.
- ◆ As sentinelas do OpenMP são:
 - Fortran: !\$OMP (ou C\$OMP ou *\$OMP)
 - C/C++: #pragma omp

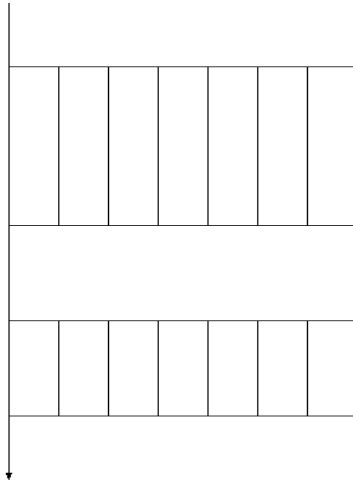
Região Paralela

- ◆ A região paralela é a estrutura básica de paralelismo no OpenMP.
- ◆ Uma região paralela define uma seção do programa.
- ◆ Os programas começam a execução com uma única *thread* (a "thread master").
- ◆ Quando a primeira região paralela é encontrada, a *thread master* cria uma equipe de *threads* (modelo *fork/join*).

Região Paralela

- ◆ Cada *thread* executa as sentenças que estão dentro da região paralela.
- ◆ No final da região paralela, a "*thread master*" espera pelo término das outras *threads* e continua então a execução de outras sentenças.

Região Paralela



```
PROGRAM FRED
.
!$OMP PARALLEL
.
.
.
.
.
.
.
!$OMP END PARALLEL
.
.
!$OMP PARALLEL
.
.
.
!$OMP END PARALLEL
.
.
```

Dados Privados e Compartilhados



- ◆ Dentro de uma região paralela, as variáveis podem ser privadas ou compartilhadas.
- ◆ Todas as *threads* vêm a mesma cópia das variáveis compartilhadas.
- ◆ Todas as *threads* podem ler ou escrever nas variáveis compartilhadas.
- ◆ Cada *thread* tem a sua própria cópia de variáveis privadas: essas são invisíveis para as outras *threads*.
- ◆ Uma variável privada pode ser lida ou escrita apenas pela sua própria *thread*.

Laços Paralelos

- ◆ Os laços são a principal fonte de paralelismo em muitas aplicações.
- ◆ Se as iterações de um laço são independentes (podem ser executadas em qualquer ordem), então podemos compartilhar as iterações entre *threads* diferentes.
- ◆ Por exemplo, se tivermos duas *threads* e o laço:

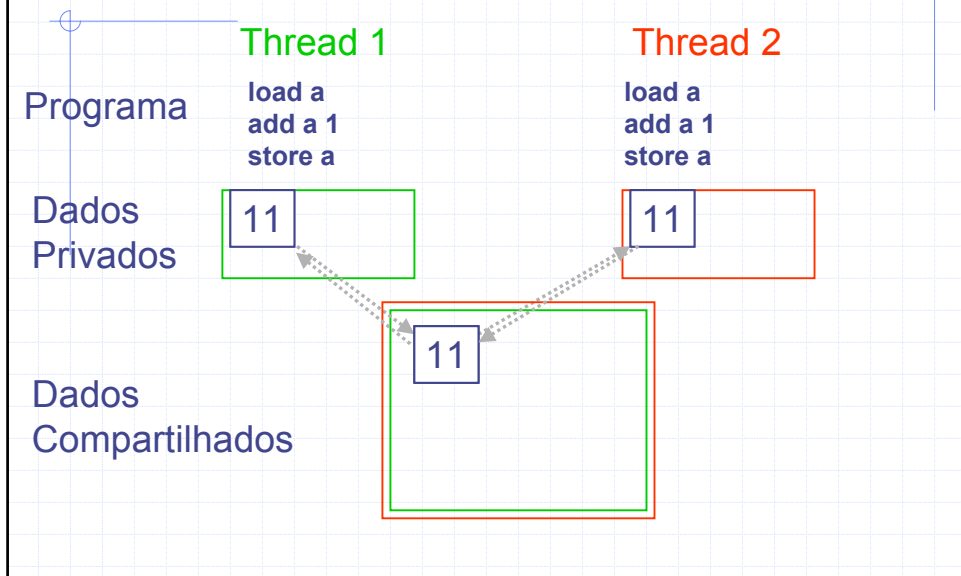
```
for (i = 0; i < 100; i++){  
    a[i] = a[i] + b[i];}
```

nós podemos fazer as iterações 0-49 em uma *thread* e as iterações 50-99 na outra.

Sincronização

- ◆ Há necessidade de assegurar que as ações nas variáveis compartilhadas ocorram na maneira correta: por ex.: a *thread* 1 deve escrever na variável **A** antes da *thread* 2 faça a sua leitura, ou a *thread* 1 deve ler a variável **A** antes que a *thread* 2 faça sua escrita.
- ◆ Note que as atualizações para variáveis compartilhadas (p.ex. $a = a + 1$) não são atômicas! Se duas *threads* tentarem fazer isto ao mesmo tempo, uma das atualizações pode ser perdida.

Exemplo de Sincronização



Reduções

- ◆ Uma redução produz um único valor a partir de operações associativas como soma, multiplicação, máximo, mínimo, e , ou. Por exemplo:

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

- ◆ Permitindo que apenas uma *thread* por vez atualize a variável **b** removeria todo o paralelismo.
- ◆ Ao invés disto, cada *thread* pode acumular sua própria cópia privada, então essas cópias são reduzidas para dar o resultado final.

Regiões Paralelas

Diretiva para Regiões Paralelas

- ◆ Um código dentro da região paralela é executado por todas as *threads*.

- ◆ Sintaxe:

C/C++:

```
#pragma omp parallel
{
  block
}
```

Diretiva para Regiões Paralelas

Exemplo:

```
call fred()
```

```
#pragma omp parallel
```

```
{
```

```
    call billy()
```

```
}
```

```
call daisy()
```

fred

billy

billy

billy

billy

daisy

Funções Úteis

- ◆ Frequentemente são utilizadas para encontrar o número de *threads* que estão sendo utilizadas.

C/C++:

```
#include <omp.h>
```

```
int omp_get_num_threads(void);
```

- ◆ **Nota importante:** retorna 1 se a chamada é fora de uma região paralela.

Funções Úteis

- ◆ Também são utilizadas para encontrar o número atual da *thread* em execução.

C/C++:

```
#include <omp.h>
int omp_get_thread_num(void);
```

- ◆ Retorna valores entre 0 e `omp_get_num_threads() - 1`

Exemplo

```
...
printf ("Alo paralelo das threads: \n");
#pragma omp parallel
{
    printf("%d \n", omp_get_thread_num());
}
printf("de volta ao mundo seqüencial \n");
...
```

Cláusulas

- ◆ Especificam informação adicional na diretiva de região paralela:

C/C++:

```
#pragma omp parallel [clausulas]
```

- ◆ Cláusulas são separadas por vírgula ou espaço no Fortran, e por espaço no C/C++.

Variáveis Privadas e Compartilhadas

- ◆ Dentro de uma região paralela as variáveis podem ser **compartilhadas** (todas as *threads* vêm a mesma cópia) ou **privada** (cada *thread* tem a sua própria cópia).
- ◆ Cláusulas SHARED, PRIVATE e DEFAULT

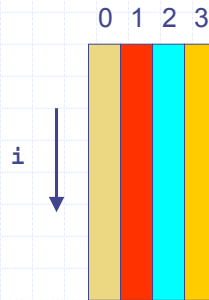
C/C++:

```
shared(list)  
private(list)  
default(shared|none)
```

Variáveis Privadas e Compartilhadas

Exemplo: cada *thread* inicia a sua própria coluna de uma matriz compartilhada:

```
#pragma omp parallel \  
default(none) private (i, myid) \  
shared(a,n)  
myid = omp_get_thread_num();  
  for(i = 0; i < n; i++){  
    a[i][myid] = 1.0;  
  }  
/* end parallel */
```



Variáveis Privadas e Compartilhadas

- ◆ Como decidir quais variáveis devem ser compartilhadas e quais privadas?
 - A maioria das variáveis são **compartilhadas**.
 - O índices dos laços são **privados**.
 - Variáveis **temporárias** dos laços são compartilhadas.
 - Variáveis **apenas** de leitura – compartilhadas
 - **Matrizes** principais – Compartilhadas
 - Escalares do tipo write-before-read – usualmente **privados**.
- ◆ Às vezes a decisão deve ser baseada em fatores de desempenho.

Valor inicial de variáveis privadas

- ◆ Variáveis privadas não tem valor inicial no início da região paralela.
- ◆ Para dar um valor inicial deve-se utilizar a cláusula **FIRSTPRIVATE**:

C/C++:

```
firstprivate(list)
```

Valor inicial de variáveis privadas

Exemplo:

```
    b = 23.0;
    .....
#pragma omp parallel firstprivate(b) private(i,myid)
{
    myid = omp_get_thread_num();
    for (i=0; i<n; i++){
        b += c[myid][i];
    }
    c[myid][n] = b;
}
```

Reduções

- ◆ Uma *redução* produz um único valor a partir de operações associativas como adição, multiplicação, máximo, mínimo, e, ou.
- ◆ É desejável que cada *thread* faça a redução em uma cópia privada e então reduzam todas elas para obter o resultado final.
- ◆ Uso da cláusula REDUCTION:

C/C++:

```
reduction(op:list)
```

Reduções

- ◆ Onde *op* pode ser:

Operação	Valor Inicial
▪ + --> soma	0
▪ - --> subtração	0
▪ * --> multiplicação	1
▪ & --> e	todos os bits em 1
▪ --> ou	0
▪ ^ --> equiv. lógica	0
▪ && --> not equiv. lógica	1
▪ --> máx	0

Reduções

Exemplo:

```
b = 0;
#pragma parallel private (i, myid) reduction (+:b)
myid = omp_get_thread_num();
for (i = 0; i < n; i++)
{
    b = b + c[i][myid];
}
/* omp end parallel */
/* b retorna a soma de todos os elementos da matriz
*/
```

Cláusula IF

- ◆ Podemos fazer a diretiva da região paralela ser condicional.
- ◆ Pode ser útil se não houver trabalho suficiente para tornar o paralelismo interessante.

C/C++:

```
if (scalar expression)
```

Cláusula IF

Exemplo:

```
#pragma omp parallel if (tasks > 1000)
{
  while(tasks > 0) donexttask();
}
```

Diretivas para
Compartilhamento de
Trabalho

Diretivas para Compartilhamento de Trabalho

- ◆ Diretivas que aparecem dentro de uma região paralela e indicam como o trabalho deve ser compartilhado entre as *threads*.
 - Laços do/for paralelos
 - Seções paralelas
 - Diretivas MASTER e SINGLE

Laços do/for paralelos

- ◆ Laços são a maior fonte de paralelismo na maioria dos códigos. Diretivas paralelas de laços são portanto muito importantes!
- ◆ Um laço do/for paralelo divide as iterações do laço entre as *threads*.
- ◆ Apresentaremos aqui apenas a forma básica.
- ◆ Sintaxe C/C++:

```
#pragma omp for [clausulas]
for loop
```

Laços do/for paralelos

- ◆ Sem cláusulas adicionais, a diretiva DO/FOR usualmente particionará as iterações o mais igualmente possível entre as *threads*.
- ◆ Contudo, isto é dependente de implementação e ainda há alguma ambigüidade:
Ex.: 7 iterações, 3 *threads*. Pode ser particionado como 3+3+1 ou 3+2+2

Laços do/for paralelos

- ◆ Como você pode dizer se um laço é paralelo ou não?
- ◆ Teste: se o laço dá o mesmo resultado se executado na ordem inversa então ele é quase certamente paralelo.
- ◆ Desvios para fora do laço não são permitidos.
- ◆ Exemplos:

1.

```
for (i=1; i < n; i++)  
{  
    a[i] = 2*a[i-1];  
}
```



Laços do/for paralelos

2.

```
ix = base;
for (i=0; i < n; i++) {
  a[ix] = a[ix]* b[i];
  ix = ix + stride;
}
```



3.

```
for (i=0; i<n; i++){
  b[i]= (a[i] - a[i-1]))*0.5;
}
```



Exemplo de Laços Paralelos

Exemplo:

```
#pragma omp parallel
#pragma omp for
  for (i=1; i <=n; i++){
    b[i] = (a[i]- a[i-1])*0.5;
  }
/* end parallel for */
```

A diretiva DO/FOR paralela

- ◆ Esta construção é tão comum que existe uma forma que combina a região paralela e a diretiva do/for:
- ◆ Sintaxe C/C++:

```
#pragma omp parallel for [clausulas]
    for loop
```

Exemplo (saxpy)

```
#pragma omp parallel for
for (i=0; i < n; i++)
{
    z[i] = a * x[i] + y;
}
```

- ◆ Este exemplo realiza a operação “multiply-add”, que é uma multiplicação de um vetor por um valor que em seguida é somado a outro.

Cláusulas

- ◆ A diretiva **DO/FOR** pode ter cláusulas **PRIVATE** e **FIRSTPRIVATE** as quais se referem ao escopo do laço.
- ◆ Note que a variável de índice do laço paralelo é **PRIVATE** por padrão (mas outros índices de laços não são).
- ◆ A diretiva **PARALLEL DO/FOR** pode usar todas as cláusulas disponíveis para a diretiva **PARALLEL**.

Seções Paralelas

- ◆ Permitem blocos separados de código serem executados em paralelo (ex. Diversas subrotinas independentes)
- ◆ Não é escalável: o código fonte deve determinar a quantidade de paralelismo disponível.
- ◆ Raramente utilizada, exceto com paralelismo aninhado (que não será abordado aqui).

Seções Paralelas

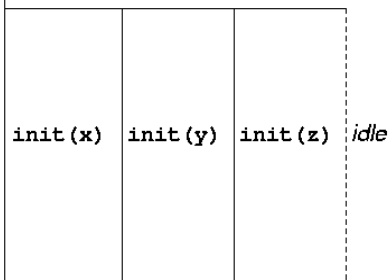
C/C++:

```
#pragma omp sections [clausulas]  
{  
  [#pragma omp section ]  
    structured-block  
  [#pragma omp section  
    structured-block  
    ...]  
}
```

Seções Paralelas

Exemplo:

```
#pragma omp parallel  
#pragma omp sections  
{  
  #pragma omp section  
    init(x);  
  #pragma omp section  
    init(y);  
  #pragma omp section  
    init(z);  
}
```



Seções Paralelas

- ◆ Diretivas **SECTIONS** podem ter as cláusulas **PRIVATE, FIRSTPRIVATE, LASTPRIVATE**.
- ◆ Cada seção deve conter um bloco estruturado: não pode haver desvio para dentro ou fora de uma seção.
- ◆ Forma abreviada C/C++:

```
#pragma omp parallel sections [clausulas]
{
...
}
```

Diretiva SINGLE

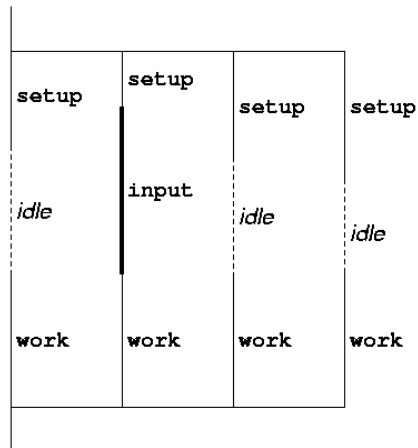
- ◆ Indica que um bloco de código deve ser executado apenas por uma *thread*.
- ◆ A primeira *thread* que alcançar a diretiva **SINGLE** irá executar o bloco.
- ◆ Outras *threads* devem esperar até que o bloco seja executado.
- ◆ Sintaxe C/C++:

```
#pragma omp single [clausulas]
    structured block
```

Diretiva SINGLE

Exemplo:

```
#pragma omp parallel
{
    setup(x);
    #pragma omp single
    {
        input(y);
    }
    work(x,y);
}
```



Diretiva SINGLE

- ◆ A diretiva **SINGLE** pode ter cláusulas **PRIVATE** e **FIRSTPRIVATE**.
- ◆ A diretiva deve conter um bloco estruturado: não pode haver desvio dentro ou para fora do dele.

Diretiva MASTER

- ◆ Indica que um bloco deve ser executado apenas pela *thread master (thread 0)*.
- ◆ Outras *threads* pulam o bloco e continuam a execução: é diferente da diretiva SINGLE neste aspecto.
- ◆ Na maior parte das vezes utilizada para E/S.
- ◆ Sintaxe C/C++:

```
#pragma omp master  
    structured block
```

Mais sobre laços paralelos do/for

Cláusula LASTPRIVATE

- ◆ Algumas vezes é necessário que se saiba o valor que uma variável privada terá na saída de um laço (normalmente é indefinido)

Sintaxe:

C/C++: lastprivate(list)

- ◆ Também se aplica à diretiva *sections* (a variável tem um valor atribuído a ela a última seção.)

Cláusula LASTPRIVATE

Exemplo:

```
#pragma omp parallel
#pragma for lastprivate (i)
  for (i=0,func(l,m,n)){
    d[i]=d[i]+e*f[i];
  }
  ix = i-1;
  ...
/* pragma end for */
```

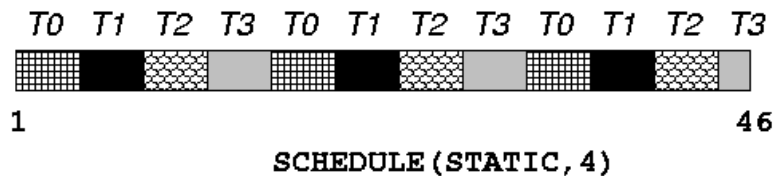
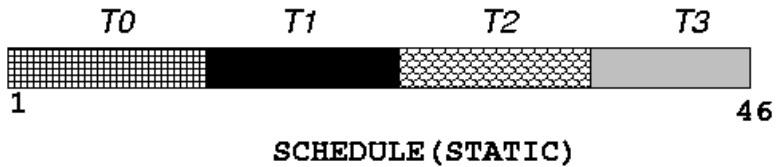
Cláusula SCHEDULE

- ◆ A cláusula **SCHEDULE** permite uma variedade de opções por especificar quais iterações dos laços são executadas por quais threads.
- ◆ Sintaxe:
C/C++: `schedule (kind[, chunksize])`
onde *kind* pode ser
STATIC, DYNAMIC, GUIDED ou **RUNTIME**
e *chunksize* é uma expressão inteira com valor positivo.
- ◆ Ex.: `#pragma for schedule(DYNAMIC,4)`

Escalonamento STATIC

- ◆ Sem a especificação de *chunksize*, o espaço de iteração é dividido em pedaços (aproximadamente) iguais e cada pedaço é atribuído a cada *thread* (escalonamento **block**).
- ◆ Se o valor de *chunksize* é especificado, o espaço de iteração é dividido em pedaços, cada um com *chunksize* iterações, e os pedaços são atribuídos ciclicamente a cada *thread* (escalonamento **block cyclic**)

Escalonamento STATIC



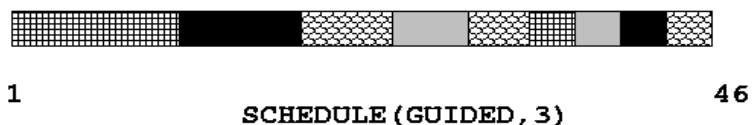
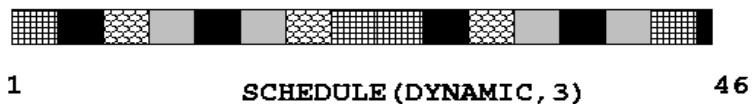
Escalonamento DYNAMIC

- ◆ O escalonamento DYNAMIC divide o espaço de iteração em pedaços de tamanho *chunksize*, e os atribui para as threads com uma política *first-come-first-served*.
- ◆ i.e. se uma *thread* terminou um pedaço, ela recebe o próximo pedaço na lista.
- ◆ Quando nenhum valor de *chunksize* é especificado, o valor padrão é 1.

Escalonamento GUIDED

- ◆ O escalonamento GUIDED é similar ao DYNAMIC, mas os pedaços iniciam grandes e se tornam pequenos exponencialmente.
- ◆ O tamanho do próximo pedaço é (a grosso modo) o número de iterações restantes dividido pelo número de *threads*.
- ◆ O valor *chunksize* especifica o tamanho mínimo dos pedaços.
- ◆ Quando nenhum valor de *chunksize* é especificado, o padrão é 1.

Escalonamentos DYNAMIC e GUIDED



Escalonamento RUNTIME

- ◆ O escalonamento RUNTIME delega a escolha do escalonamento para a execução, quando é determinado pelo valor da variável de ambiente OMP_SCHEDULE.

```
$ export OMP_SCHEDULE="guided,4"
```

- ◆ É ilegal especificar um valor de *chunksize* com o escalonamento RUNTIME.

Escolhendo um Escalonamento

Quando utilizar um escalonamento?

- ◆ **STATIC**: melhor para laços balanceados – menor sobrecarga.
- ◆ **STATIC,*n*** : melhor para laços com desbalanceamento suave.
- ◆ **DYNAMIC** : útil se as iterações tem grande variação de carga, mas acaba com a localidade espacial dos dados.
- ◆ **GUIDED**: freqüentemente menos cara que **DYNAMIC**, mas tenha cuidado com laços onde as primeiras iterações são as mais caras!
- ◆ Use **RUNTIME** para experimentação adequada.

Diretiva ORDERED

◆ Pode especificar o código dentro de um laço que deverá ser executado na ordem em que seria se executado seqüencialmente.

◆ **Sintaxe:**

```
C/C++: #pragma omp ordered  
        structured block
```

◆ Pode aparecer dentro de uma diretiva DO/FOR que tiver a cláusula ORDERED especificada.

Diretiva ORDERED

◆ **Exemplo:**

```
#pragma omp parallel for ordered  
    for (j =0; j < n; j++)  
        ...  
#pragma omp ordered  
    printf ("%d %d \n", j,count[j])  
    ...
```

Sincronização

O que é necessário?

- ◆ **É necessário sincronizar ações em variáveis compartilhadas.**
- ◆ **É necessário assegurar a ordenação correta de leituras e escritas.**
- ◆ **É necessário proteger a atualização de variáveis compartilhadas (não atômicas por padrão).**

Diretiva BARRIER

- ◆ Nenhuma *thread* pode prosseguir além de uma barreira até que todas as outras *threads* cheguem até ela.
- ◆ Note que há uma barreira implícita no final das diretivas DO/FOR, SECTIONS e SINGLE.
- ◆ Sintaxe:
C/C++:

```
#pragma omp barrier
```

- ◆ Ou nenhuma ou todas as *threads* devem encontrar a barreira: senão DEADLOCK!!

Diretiva BARRIER

Exemplo:

```
#pragma omp parallel private(i,myid,neighb)
{
    myid = omp_get_thread_num();
    neighb = myid - 1;
    if (myid == 0) neighb = omp_get_num_threads()-1;
    ...
    a[myid] = a[myid]*3.5;
    #pragma omp barrier
    b[myid] = a[neighb] + c
    ...
}
```

- ◆ Barreira requerida para forçar a sincronização em **a**

Cláusula NOWAIT

- ◆ A cláusula **NOWAIT** pode ser usada para suprimir as barreiras implícitas no final das diretivas **DO/FOR**, **SECTIONS** e **SINGLE**. (Barreiras são caras!)

- ◆ Sintaxe:

```
C/C++: #pragma omp for nowait  
      for loop
```

- ◆ Igualmente para **SECTIONS** e **SINGLE**.

Cláusula NOWAIT

- ◆ Exemplo: Dois laços sem dependências

```
#pragma omp parallel  
{  
  #pragma omp for nowait  
  for (j=0; j < n; j++){  
    a[j] = c * b[j];  
  }  
  #pragma omp for nowait  
  for (i=0; i < m; i++){  
    x[i] = sqrt(y[i]) * 2.0;  
  }  
}
```

Cláusula NOWAIT

- ◆ Use com **EXTREMO CUIDADO!**
- ◆ É muito fácil remover uma barreira que é necessária.
- ◆ Isto resulta no pior tipo de erro: comportamento não-determinístico da aplicação (às vezes o resultado é correto, às vezes não, o comportamento se altera no depurador, etc.).
- ◆ Pode ser um bom estilo de codificação colocar a cláusula NOWAIT em todos os lugares e fazer todas as barreiras explicitamente.

Cláusula NOWAIT

Exemplo:

```
#pragma omp for
  for (j =0; j < n; j++){
    a[j]= b[j] + c[j];
  }
#pragma omp for
  for (j =0; j < n; j++){
    d[j] = e[j] * f;
  }
#pragma omp for
  for (j =0; j < n; j++){
    z[j] = (a[j]+a(j+1)) * 0.5;
  }
```

Pode-se remover a primeira barreira, OU a segunda, mas não ambas, já que há uma dependência em a

Seções Críticas

- ◆ Uma seção crítica é um bloco de código que só pode ser executado por uma *thread* por vez.
- ◆ Pode ser utilizado para proteger a atualização de variáveis compartilhadas.
- ◆ A diretiva **CRITICAL** permite que as seções críticas recebam nomes.
- ◆ Se uma *thread* está em uma seção crítica com um dado nome, nenhuma outra *thread* pode estar em uma seção crítica com o mesmo nome (embora elas possam estar em seções críticas com outros nomes).

Diretiva CRITICAL

- ◆ Sintaxe:
C/C++: `#pragma omp critical [(name)]`
structured block
- ◆ Se o nome é omitido, um nome nulo é assumido (todas as seções críticas sem nome tem efetivamente o mesmo nome).

Diretiva CRITICAL

Exemplo: colocando e retirando de uma pilha

```
#pragma omp parallel shared(stack), private(inext,inew)
...
#pragma omp critical (stackprot)
{
    inext = getnext(stack);
}
work(inext,inew);
#pragma omp critical (stackprot)
    if (inew > 0) putnew(inew,stack);
}
...
```

Diretiva ATOMIC

- ◆ Usada para proteger uma atualização única para uma variável compartilhada.
- ◆ Aplica-se apenas a uma única sentença.
- ◆ Sintaxe:

C/C++: `#pragma omp atomic`
statement

Onde *statement* deve ter uma das seguintes formas:

$x \text{ binop} = \text{expr}$, $x++$, $++x$, $x--$, or $--x$

and *binop* is one of $+$, $*$, $-$, $/$, $\&$, \wedge , \ll , or \gg

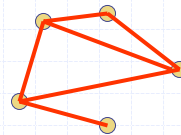
Diretiva ATOMIC

- ◆ Note que a avaliação de *expr* não é atômica.
- ◆ Pode ser mais eficiente que usar diretivas **CRITICAL**, por exemplo, se diferentes elementos do arranjo podem ser protegidos separadamente.

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]++;
  }
```



Rotinas Lock

- ◆ Ocasionalmente pode ser necessário mais flexibilidade que a fornecida pelas diretivas **CRITICAL** e **ATOMIC**.
- ◆ Um *lock* é uma variável especial que pode ser marcada por uma *thread*. Nenhuma outra *thread* pode marcar o *lock* até que a *thread* que o marcou o desmarque.
- ◆ Marcar um *lock* pode tanto pode ser bloqueante como não bloqueante.
- ◆ Um *lock* deve ter um valor inicial antes de ser usado e pode ser destruído quando não for mais necessário.
- ◆ Variáveis de *lock* não devem ser usadas para qualquer outro propósito.

Rotinas Lock – Sintaxe

C/C++:

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Existem também rotinas de *lock* aninháveis que permitem a uma mesma *thread* ativar um *lock* múltiplas vezes antes de liberá-lo o mesmo número de vezes.

Rotinas Lock – Exemplo

Exemplo:

```
    call omp_init_lock(iLOCK)
#pragma omp parallel shared(iLOCK)
...
do {
    do_something_else(); }
    while ( ~ omp_test_lock(iLOCK))

work();
omp_unset_lock(iLOCK);
...
```

Escolhendo a Sincronização

- ◆ Como uma regra simples, use a diretiva **ATOMIC** sempre que possível, já que permite o máximo de otimização.
- ◆ Se não for possível use a diretiva **CRITICAL**. Tenha cuidado de usar diferentes *nomes* sempre que possível.
- ◆ Como um último recurso você pode ter que usar as rotinas *lock*, mas isto deve ser uma ocorrência muito rara.

Funcionalidades Adicionais

OpenMP 2.0

Novidades no Fortran 2.0

- ◆ Suporte completo do Fortran 90/95:
 - Diretiva **WORKSHARE** para sintaxe de arranjos.
 - Diretiva **THREADPRIVATE/COPYIN** em variáveis (ex. para dados modulares).
 - Comentários "In-line" em diretivas.
- ◆ Reduções em arranjos.
- ◆ Cláusula **COPYPRIVATE** na diretiva **END SINGLE** (propaga o valor para todas as threads).
- ◆ Cláusula **NUM_THREADS** em regiões paralelas.
- ◆ Rotinas de temporização.
- ◆ ...Vários esclarecimentos (e.g. reprivatização de variáveis é permitida.)

Novidades no C/C++ 2.0

- ◆ Cláusula **COPYPRIVATE** na diretiva **END SINGLE** (propaga o valor para todas as threads).
- ◆ Cláusula **NUM_THREADS** em regiões paralelas.
- ◆ Rotinas de temporização.
- ◆ ...Várias correções e esclarecimentos.

Reduções em arranjos

- ◆ Arranjos podem ser usados como variáveis de redução (anteriormente só escalares e elementos de um arranjo).
- ◆ Exemplo:

```
#pragma omp parallel for private (i) reduction (+:b)
  for (j = 0; j < N; j++)
    for (i=0; i < M; i++)
      b(i) = b(i) + b(i,j);
```

Cláusula COPYPRIVATE

- ◆ Difunde o valor de uma variável privada para todas as threads no final de uma diretiva SINGLE.
- ◆ Talvez o uso mais importante seja a leitura de valores de variáveis privadas.
- ◆ Sintaxe:

C/C++:

```
#pragma omp single copyprivate(list)
```

Cláusula COPYPRIVATE

Exemplo:

```
#pragma omp parallel private (a,b)
{
    ...
    #pragma omp single copyprivate(a)
    {
        scanf ("Entre com o valor = %d", a);
    }

    b = a * a;

    ...
}
```

Cláusula NUMTHREADS

- ◆ A cláusula NUMTHREADS da versão 2.0 do OpenMP especifica o número de threads que vão executar a diretiva.

```
!$OMP PARALLEL DO NUM_THREADS(4)
    DO I = 1,4
!$OMP PARALLEL DO NUM_THREADS(TOTALTHREADS/4)
        DO J = 1,N
            A(I,J) = B(I,J)
        END DO
    END DO
```

Nota: O valor colocado na cláusula se sobrepõe ao valor da variável de ambiente OMP_NUM_THREADS (ou da chamada omp_set_num_threads())

Esclarecimentos

- ◆ Os padrões 2.0 tanto para Fortran como o C/C++ contém um grande número de correções e esclarecimentos.
- ◆ Se alguma coisa não está clara no padrão 1.0/1.1, vale a pena ler a seção correspondente na versão 2.0, mesmo que você não esteja utilizando um compilador compatível com a versão 2.0.

Usando o OpenMP

- ◆ **Comparações com Troca de Mensagem:**
 - Algoritmo de decomposição de domínio é o mesmo, mas a implementação é mais simples:
 - ◆ Nenhuma necessidade de troca de mensagens, células fantasma ou *buffers* de sombra.
 - ◆ Dados globais, variáveis de campo compartilhadas: leitura por qualquer *thread*, escrita pode ser compartilhada.
 - Paraleliza apenas partes do código que são significativas, não há necessidade de converter o código todo.
 - ◆ Pré-processamento, pós-processamento pode ser deixado à parte.

Usando o OpenMP

- ◆ **OMP_NUM_THREADS** – especifica o número de *threads* para serem usadas durante a execução de regiões paralelas.
- ◆ O valor padrão para esta variável é 1.
- ◆ **OMP_NUM_THREADS** *threads* serão usadas para executar o programa independente do número de processadores físicos disponíveis no sistema.
- ◆ Como resultado, você pode executar programas com mais *threads* do que o número de processadores físicos e eles vão executar corretamente. Contudo, o desempenho da execução dos programas nesta maneira poderá ser ineficiente.

Usando o OpenMP

- ◆ **OMP_SCHEDULE** – especifica o tipo de escalonamento para divisão das iterações do laço entre as *threads*, para uso pelas cláusulas "omp for" e "omp parallel for" com uso da cláusula de escalonamento "runtime".
- ◆ O valor padrão para esta variável é "static".
- ◆ Se o tamanho do *chunk* não for especificado, um valor de 1 é assumido, exceto no caso de escalonamento estático.
- ◆ Exemplos do uso da cláusula **OMP_SCHEDULE** são os seguintes:

```
$ setenv OMP_SCHEDULE "static, 5"  
$ setenv OMP_SCHEDULE "guided, 8"  
$ setenv OMP_SCHEDULE "dynamic"
```

Usando o OpenMP

- ◆ **MPSTKZ** – aumenta o tamanho das pilhas utilizadas pelas *threads* executando regiões paralelas. Para uso com programas que utilizam grandes quantidades de variáveis locais às *threads* nas rotinas chamadas nas regiões paralelas.
- ◆ O valor deve ser um inteiro <n> concatenado com M ou m para especificar o tamanho da pilha em n megabytes: por exemplo
\$ setenv MPSTKZ 8M

Usando o OpenMP

- ◆ **Compilando código OpenMP paralelizado para C/C++ usando o compilador Intel.**
- ◆ **C:**
icc -o myprog myprog.c -openmp -openmp_report2
- ◆ **C++:**
icc -o myprog myprog.C -openmp -openmp_report2
- ◆ **Habilitando -openmp_report2** oferece como saída diagnóstico de paralelização durante o tempo de compilação.

Limitações

- ◆ É fácil paralelizar código serial para o OpenMP.
- ◆ OpenMP pode executar em modo serial

Contudo:

- ◆ Para máquinas com memória compartilhada apenas.
- ◆ Limitada escalabilidade – depois de 8 processadores não há muito speed-up.
- ◆ Sobrecarga para paralelização de regiões paralelas e laços paralelos.

Limitações

- ◆ OpenMP atualmente não define ou especifica construções para controlar a correspondência entre threads e processadores.
- ◆ Processos podem migrar, causando overhead.
- ◆ Este comportamento é dependente de sistema.
- ◆ Soluções para este caso, específicas para cada sistemas, podem ser oferecidas em algumas implementações.

Referências OpenMP

- ◆ <http://www.openmp.org>
 - Official web site: language specifications, links to compilers and tools, mailing lists
- ◆ <http://www.compunity.org>
 - OpenMP community site: more links, events, resources
- ◆ <http://scv.bu.edu/SCV/Tutorials/OpenMP/>
- ◆ http://www.ccr.buffalo.edu/documents/CCR_openmp_pbs.PDF
- ◆ <http://www.epcc.ed.ac.uk/research/openmpbench/>
- ◆ Book: "Parallel Programming in OpenMP", Chandra et. al., Morgan Kaufmann, ISBN 1558606718.