Programa Campus Ambassador HPC

TUTORIAL OpenMP C/C++

Maceió, março de 2008

Programa Campus Ambassador HPC

TUTORIAL OpenMP C/C++

Maria Cecília Rodrigues Sena
maria.sena@sun.com

Joseaderson Augusto de Caldas Costa
joseanderson.decaldas@sun.com

orientação:

Carlos Alberto Thomaz
carlos.thomaz@sun.com

Eduardo Setton Sampaio da Silveira
eduardosetton@lccv.ufal.br

Antonio Augusto Russo
aarusso@usp.br

Heitor Soares Ramos Filho
heitor@*lccv.ufal.br*

Maceió, março de 2008

Edição 1, 03/2008

A documentação apresentada a seguir foi elaborada pelos embaixadores Joseanderson Caldas e Maria Cecília Sena, membros do Laboratório de Computação Científica e Visualização (LCCV-UFAL) e do Programa Campus Ambassador HPC, edição 2008, promovido pela Sun Microsystems do Brasil.

A elaboração do presente tutorial teve a orientação dos coordenadores Carlos Thomaz (Sun Microsystems), Eduardo Setton (LCCV-UFAL), Heitor Ramos (LCCV-UFAL) e Antonio Russo (USP).

ÍNDICE

PARTE I

PARTE II

INTRODUÇÃO

Ao longo dos anos, o desenvolvimento científico tem exigido das simulações numéricas resultados cada vez mais confiáveis e próximos da realidade. Dessa forma, para acompanhar esses avanços, a computação cientifica se depara com o desafio de superar as diversas barreiras que surgem em virtude da limitação física dos computadores, tais como a demanda por processamento numérico, armazenamento de dados, visualização, entre outros.

Nesse contexto, a computação de alto desempenho (*High Performance Computing* - HPC), termo amplamente utilizado na computação científica, tem se apresentado como uma importante frente de pesquisa nos últimos anos. O termo pode ser definido como qualquer conjunto de técnicas que visem otimizar ou até mesmo viabilizar o processamento de simulações numéricas. Podemos citar como uma dessas técnicas o uso de processamento paralelo em clusters e supercomputadores.

As técnicas de processamento paralelo mais difundidas atualmente são as técnicas de memória distribuída e as técnicas de memória compartilhada. Entende-se por técnicas de memória distribuída aquelas que se aplicam aos computadores que possuem arquitetura de memória distribuída, ou seja, máquinas com vários processadores que possuem seu próprio recurso de memória e são interconectados através de uma rede local. Atualmente, o padrão MPI (*Message Passing Interface*) tem sido o mais utilizado nesses casos e seu funcionamento se dá basicamente através da troca de dados entre os processadores. Por sua vez, as técnicas de memória compartilhada são utilizadas em ambientes que possuem vários núcleos de processamento compartilhando o mesmo recurso de memória. Nesses casos, a utilização do padrão OpenMP tem crescido bastante nos últimos anos, uma vez que as funcionalidades do mesmo facilitam o desenvolvimento de aplicações em memória compartilhada.

Visando estudantes e profissionais que atuam no campo da computação científica, mais especificamente, aqueles envolvidos com a área da computação de alto desempenho em ambientes de memória compartilhada, o presente tutorial descreve e ilustra as funcionalidades do padrão OpenMP. Portanto, a documentação a seguir tem por objetivo se tornar mais uma fonte de consulta para os desenvolvedores interessados nesse modelo de programação paralela.

Inicialmente, procurou-se expor alguns tópicos da Computação de Alto desempenho que justificam a elaboração desse material, abordando temas como arquiteturas de computadores, histórico do padrão OpenMP, novas

alternativas para computação de alto desempenho (tecnologia *multicore*), máquinas de memória compartilhada, métricas de desempenho, entre outros.

Na segunda parte, a documentação descreve a sintaxe das diretivas, funções e variáveis do padrão OpenMP. Com o intuito de esclarecer possíveis dúvidas, o tutorial também apresenta alguns exemplos que ilustram o uso desse padrão.

PARTE I

1 - ARQUITETURA DE COMPUTADORES

Devido à existência de uma grande variedade de arquiteturas de computadores, inúmeras taxonomias ao longo dos anos foram propostas, porém a classificação mais aceita atualmente nos ambiente de *hardware* é a conhecida taxonomia de Flynn (Flynn *apud* Dantas, 1972). Essa classificação leva em consideração o número de instruções executadas em paralelo e o conjunto de dados sob os quais as instruções são submetidas. Dessa forma, na Figura 1 são mostrados os tipos de computadores estabelecidos pela classificação de Flynn.

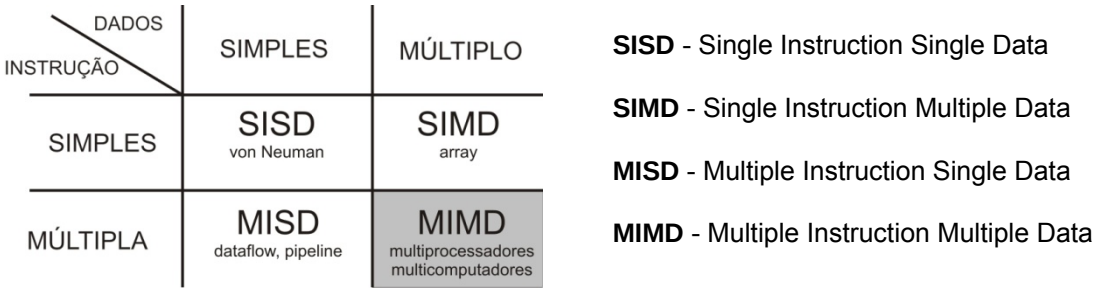

Figura 1 – Classificação das arquiteturas de computadores segundo Flynn, 1972

Os computadores denominados com arquitetura **MIMD** possuem múltiplos núcleos de processamento, cada qual podendo executar instruções independente dos demais. Essas máquinas também podem ser classificadas como multiprocessadores (memória compartilhada) ou multicomputadores (memória distribuída). O primeiro grupo pode também ser denominado de ambiente fortemente acoplado enquanto que o segundo fracamente acoplado (Figura 2). As máquinas denominadas multiprocessadas possuem, como principal característica, o compartilhamento da memória entre todos os processadores.

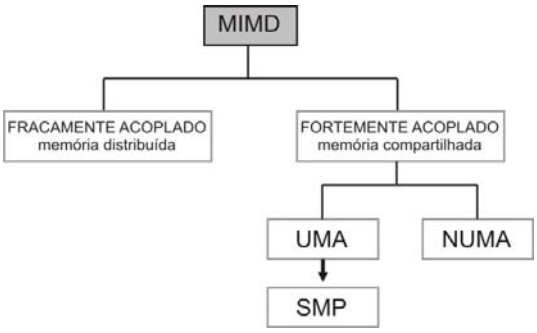

Figura 2 – Classificação das arquiteturas **MIMD**

Em arquiteturas de memória compartilhada, outra classificação que também merece destaque é a que define como os processadores acessam a

memória. Dessa forma, eles podem ser classificados como de arquitetura **UMA** (*Uniform Memory Access*) ou **NUMA** (*Non-Uniform Memory Access*).

UMA – Nas máquinas que possuem esse tipo de arquitetura, o tempo de acesso aos dados localizados em qualquer posição da memória é o mesmo para todos os processadores. A forma de interconexão mais comum neste tipo de máquina é o barramento e a memória geralmente é implementada com um único módulo

NUMA - Nos computadores que possuem arquitetura **NUMA**, o tempo de acesso à memória depende da posição em que a mesma se encontra em relação ao processador. Isso acontece devido ao fato de que, nesse tipo de arquitetura, existe um limite quanto ao número de processadores por barramento de memória. Dessa forma, um processador terá acesso mais rápido a uma região de memória localizada no seu barramento do que a uma região localizada em um barramento diferente.

2 – PROCESSAMENTO PARALELO

A Computação de Alto Desempenho consiste em uma disciplina da Computação Científica que tem como objetivo aumentar a velocidade de processamento ou até mesmo viabilizar as simulações numéricas de problemas físicos complexos e com elevado grau de discretização.

Uma das técnicas da Computação Científica de Alto Desempenho é o Processamento Paralelo, que consiste na divisão de uma determinada tarefa em tarefas menores e na execução de cada uma dessas tarefas em diferentes processadores. O Processamento Paralelo pode ser realizado em ambientes de memória distribuída e em ambientes de memória compartilhada.

Processamento Paralelo em ambientes de memória distribuída consiste em distribuir as tarefas para serem executadas em diferentes processadores que possuem seus próprios recursos de memória e são interconectados através de uma rede local. Nesse caso, o processamento paralelo é realizado por meio da troca de dados entre os processadores através da rede de interconexão.

Processamento Paralelo em ambientes de memória compartilhada consiste na divisão das tarefas entre vários processadores que compartilham o mesmo recurso de memória global. Nesse caso não há necessidade de troca de dados entre os processadores, uma vez que todos têm acesso ao mesmo endereço de memória. Por outro lado, surge a necessidade da sincronização do acesso aos dados na memória compartilhada.

Nas últimas décadas são crescentes as pesquisas no campo da Computação de Alto Desempenho. Diante do desenvolvimento tecnológico das arquiteturas paralelas e das limitações físicas que começam a surgir nos computadores pessoais, a Computação de Alto Desempenho, que inclui a computação paralela, se apresenta como forte alternativa para aumentar o desempenho dos computadores e viabilizar uma grande variedade de aplicações nos meios científicos e industriais.

2.1 – Processamento Paralelo em SMPs

SMP (*Symetric Multiprocessor System*) é uma máquina de arquitetura paralela **MIMD** com memória compartilhada, constituída por processadores que na maioria das vezes estão conectados à memória por meio de um barramento. Elas possuem arquitetura **UMA** (*Uniform Memory Access*), isto significa que todos os processadores têm acesso a todas as posições de memória e que o tempo de acesso é o mesmo para todas as posições.

Entretanto, vale ressaltar que nem toda memória é compartilhada numa SMP. A velocidade de processamento dos processadores tem crescido de forma bastante rápida, porém as instruções são realizadas sob dados cujo acesso na memória principal do computador não cresce na mesma proporção. Para que esse acesso aos dados aconteça de forma mais rápida, cada processador possui uma memória privada, menor e mais rápida, denominada memória *cache*. Os dados são então trazidos da memória principal para a memória *cache* para que o processador possa acessá-los. A memória *cache*, entretanto, é especifica de cada processador, somente sendo acessada por eles.

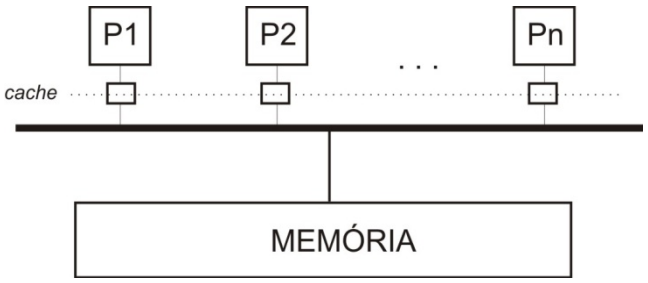

Figura 3 – Modelo de compartilhamento de memória de um SMP

O processamento paralelo em **SMPs** é feito através de programas com múltiplas *threads* (linhas de execução) ou múltiplos processos que executam simultaneamente, porém de forma independente, e cooperam para resolver um problema através do compartilhamento da memória. O desenvolvimento de programas paralelos com essa característica requer o uso de ferramentas que possibilitem a criação de *threads* e que implementem mecanismos de sincronização entre elas. As ferramentas de programação paralela mais comuns em **SMPs** atualmente são aquelas com *threading* explícito e as baseadas em diretivas de compilação.

Programação paralela com *threading* explícito - O programador cria explicitamente múltiplas *threads* dentro de um mesmo processo e divide, também explicitamente, o trabalho a ser realizado pelo programa entre essas *threads*. Assim como a criação, cabe ao programador gerenciar toda a sincronização das *threads*. A ferramenta de programação paralela com *threading* explícito mais utilizada atualmente é a POSIX *Threads*, que representa uma API (*Application Programming Interface*) do padrão POSIX.

Programação paralela baseada em diretivas – O programador utiliza diretivas de compilação inseridas no código seqüencial para informar ao compilador quais as regiões devem ser paralelizadas. Com base nestas diretivas o compilador gera um código paralelo. Dessa forma é necessário um compilador "especial", ou seja, capaz de entender as diretivas e gerar o código *multi-thread*. Pode-se citar o padrão OpenMP como uma interface que especifica um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela em arquiteturas com memória compartilhada.

3 - PADRÃO OpenMP

O padrão OpenMP é desenvolvido e mantido pelo grupo OpenMP *Architecture Review Board* (ARB), formado pelos maiores fabricantes de software e hardware do mundo, tais como SUN Microsystems, SGI, IBM, Intel, dentre outros, que, no final de 1997, reuniram esforços para criar um padrão de programação paralela para arquiteturas de memória compartilhada.

O *Open* significa que é padrão e está definido por uma especificação de domínio público e MP são as siglas de *Multi Processing*. O OpenMP consiste em uma API e um conjunto de diretivas que permite a criação de programas paralelos com compartilhamento de memória através da implementação automática e otimizada de um conjunto de *threads*. Suas funcionalidade podem atualmente ser utilizadas nas linguagens Fortran 77, Fortran 90, C e C++. A primeira versão desse padrão, o OpenMP 1.0, foi introduzida ao público no final de 1997. Em 2000, para Fortran, e em 2002, para C/C++, foi publicada a versão 2.0. Atualmente, o OpenMP encontra-se na versão 2.5.

O OpenMP não é uma linguagem de programação, ele representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação, nos programas seqüenciais, de diretivas que indicam como o trabalho será dividido entre os processadores ou *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código. Mesmo podendo ser usada em máquinas monoprocessadas, o OpenMP foi planejado para satisfazer a implementação em larga escala das arquiteturas **SMPs**. O sucesso do OpenMP é propagado pelas arquiteturas *multicore* e *multithread* e pode ser atribuído a fatores como a robusta estrutura de programação paralela suportada e a facilidade do seu uso.

Diretivas - Consiste em uma linha de código com significado "especial" para o compilador. Por exemplo, nas linguagens C e C++ as diretivas OpenMP são identificadas pelo `#pragma omp`, enquanto que na linguagem Fortran, as diretivas são identificadas pela sentinela `!$omp`.

3.1 – Modelo de programação do OpenMP

No OpenMP, a paralelização é explicitamente realizada com múltiplas *threads* dentro de um mesmo processo. A criação de *threads* é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas simultaneamente. Cada *thread* possui sua própria pilha de execução, porém compartilha o mesmo endereço de memória com as outras *threads* do mesmo processo (enquanto que cada processo possui seu próprio espaço de memória).

O modelo de programação inicia com uma única *thread* que executa sozinha as instruções até encontrar uma região paralela (identificada pela diretiva), ao encontrar essa região ela cria um grupo de *threads*, que juntas executam o código dentro dessa região. Quando as *threads* completam a execução do código na região paralela, elas sincronizam-se e somente a *thread* inicial segue na execução do código até que uma nova região paralela seja encontrada ou que o programador decida encerrar essa *thread*. Esse modelo de programação é conhecido na literatura como *fork-join* e está ilustrado na figura 4.

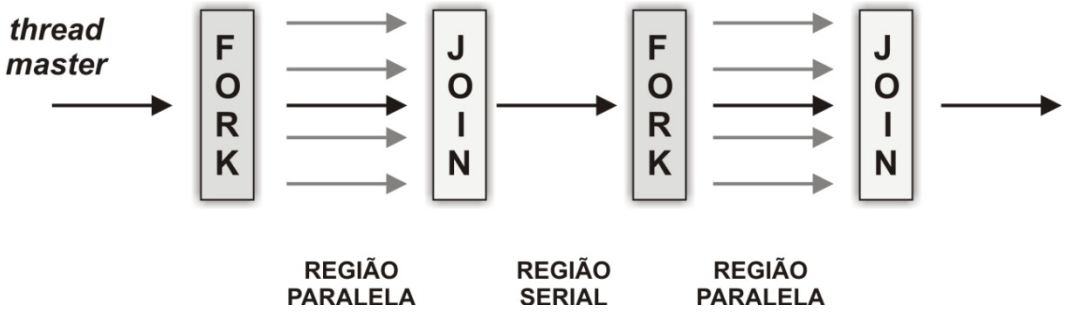

Figura 4 – Modelo de programação OpenMP

Como citado anteriormente, o OpenMP é constituído de variáveis de ambiente, diretivas de compilação e bibliotecas de serviço, que juntam integram todas as funcionalidades disponíveis do padrão OpenMP. Na figura 5 é ilustrada a sintaxe desses elementos para a linguagem C/C++.

□ VARIÁVEIS DE AMBIENTE

```
OMP_NOME
```

□ DIRETIVAS DE COMPILAÇÃO

```
#pragma omp diretiva [cláusula]
```

□ BIBLIOTECAS DE SERVIÇO

```
omp_serviço (...)
```

Figura 5 – Elementos da interface OpenMP

3.2 – Vantagens do uso do OpenMP

- Normalmente são feitas poucas alterações no código serial existente
- Possui uma robusta estrutura para suporte a programação paralela
- Fácil compreensão e uso das diretivas
- Suporte a paralelismo aninhado
- Possibilita o ajuste dinâmico do número de *threads*

4 – TECNOLOGIA *MULTICORE*

Uma das soluções apresentadas e implantadas nos últimos anos para aumentar o poder de processamento das máquinas era o aumento da velocidade de *clock*. Porém esse caminho possui limitações inerentes, principalmente relacionadas ao super consumo e a emissão de calor. Umas das alternativas para minimizar esses problemas são os processadores *Multicore*, eles são capazes de prover maior capacidade de processamento com um custo/benefício melhor do que os processadores Single-Core.

A tecnologia *Multicore* (múltiplos núcleos) consiste em colocar duas ou mais unidades de execução (*cores*) no interior de um único "pacote de processador" (um único chip). O sistema operacional trata esses núcleos como se cada um fosse um processador diferente, com seus próprios recursos de execução. Na maioria dos casos, cada unidade possui seu próprio *cache* e pode processar várias instruções simultaneamente. Adicionar novos núcleos de processamento a um processador possibilita que as instruções de aplicações sejam executadas em paralelo em vez de serialmente, como ocorre em um núcleo único.

É importante notar que, para uma total utilização do poder de processamento oferecido pela tecnologia *Multicore*, as aplicações devem ser escritas de modo a usar intensivamente o conceito de *threads*.

Uma arquitetura *Multicore* é geralmente um multiprocessador simétrico (SMP) implementado em um único circuito VLSI (*Very Large Scale Integration*). O objetivo é melhorar o paralelismo no nível de *threads*, ajudando especialmente as aplicações que não conseguem se beneficiar dos processadores superescalares atuais por não possuírem um bom paralelismo no nível de instruções.

5 – MÉTRICAS DE DESEMPENHO

Ao resolvermos um problema num sistema paralelo, o principal interesse é saber o ganho que teremos sobre a implementação serial deste problema. Uma das principais métricas que fornecem essa informação é o fator *speed-up* S(n), que representa o ganho de velocidade de processamento de uma aplicação quando executada com *n* processadores. Quanto maior o *speed-up*, mais rápido se encontra o código paralelo. A equação que define essa métrica é mostrada abaixo.

$$S_n = \frac{T_S}{T_n}$$

T_S – Tempo Serial
T_n – Tempo Paralelo

Porém, por maior que seja a disponibilidade de processadores a serem utilizados, o fator *speed-up* é limitado por um valor máximo, decorrente da parcela serial do código, isso é ilustrado pela lei de Amdahl.

5.1 – Lei de Amdahl

Como comentado acima, nem todos os trechos do código são paralelizáveis. Dessa forma pode-se sempre identificar, dentro de um código, uma região que sempre será executada em serial e uma outra que pode ser paralelizada (Figura 6). O aumento do número de processadores apenas influenciará no tempo necessário para executar a região passível de paralelização.

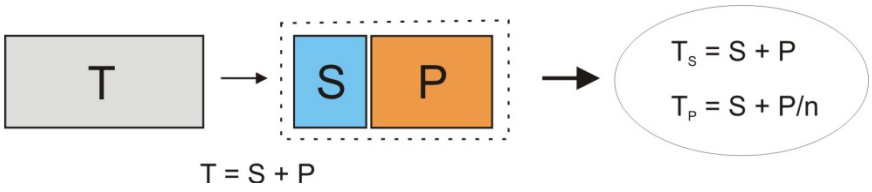

Figura 5 – Ilustração dos trechos serial e paralelo de um código

Observando a ilustração acima, em que n é o número de processadores, podemos combinar estas expressões com a definição do fator *speed-up*, resultando na expressão conhecida como Lei de Amdahl, que representa o *speed-up* teórico.

$$S_P = \frac{1}{S + (1-S)/n} \quad onde \lim_{n\to\infty} S_P = \frac{1}{S}$$

S – Fração serial do código
n – Número de processadores

A idéia da Lei de Amdahl é que existe sempre um limite ao qual a capacidade de ganho pela paralelização estará sujeita. Isso se deve a fatores como entrada e saída, dependência entre os dados e outros fatores intrínsecos à aplicação e à técnica de programação paralela utilizada.

Para ilustrar a lei de Amdahl, observemos o exemplo de um pintor que deve executar o trabalho de pintar um conjunto de estacas. As informações relacionadas ao pintor e ao trabalho são ilustradas abaixo (Figura 6).

Preparo da tinta = 30 seg
Pintura das estacas = 5 min = 300 seg
Tempo para a tinta secar = 30 seg

Figura 6 – Informações no pintor e do trabalho a ser executado

Se considerarmos que apenas um pintor seja contratado para executar o serviço, o mesmo levará um tempo de 360 segundos (30 + 300 + 30) para realizar a atividade. Entretanto, se dois pintores forem contratados, o tempo para pintar as estacas será reduzido a 150 segundos, porém o tempo para realizar as outras duas atividades permanecerá o mesmo, pois independe do número de pintores, de modo que todo o serviço levará um tempo total de 210 segundos. Essa redução no tempo representa um *speed-up* de 1,7. Seguindo com o mesmo raciocínio para um número maior de pintores, monta-se o quadro da figura 7.

Número de Pintores	Tempo	Speedup
1	360 = 30 + 300 + 30	1,0 x
2	210 = 30 + 150 + 30	1,7 x
10	90 = 30 + 30 + 30	4,0 x
100	63 = 30 + 3 + 30	5,7 x
infinito	60 = 30 + 0 + 30	6,0 x

Figura 7 –Valores de tempo total e speed-up

Observando o gráfico abaixo da figura 8, temos que a taxa de crescimento do *speed-up* decresce com o acréscimo de pintores a partir de um certo número. Esse fato ocorre devido à lei de Amdahl, que indica que somente a parcela do tempo referente à pintura das estacas pode ser reduzida. O fator *speed-up* ficará então limitado pela parcela das atividades que não podem ser paralelizadas, ou seja, as atividades de preparo da tinha e espera da tinta secar.

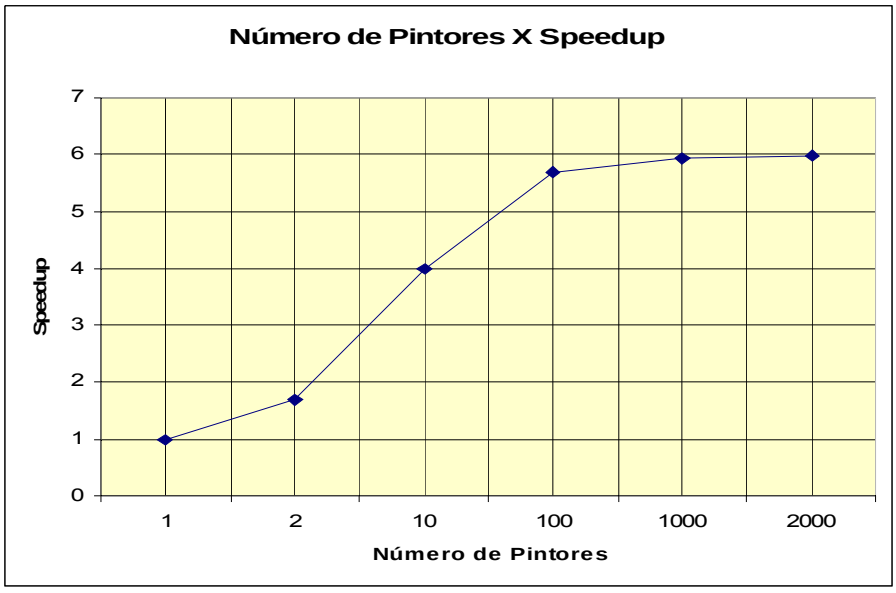

Figura 8 – Ilustração dos trechos serial e paralelo de um código

PARTE II

1- DIRETIVAS

As diretivas do OpenMP são baseadas na diretiva `#pragma` definida no padrão da linguagem C/C++. Os compiladores que suportam OpenMP em C/C++ possuem uma opção de linha de comando que ativa e permite a interpretação das diretivas do OpenMP.

1.1 – Formato das diretivas

O formato padrão de uma diretiva OpenMP é mostrado a seguir:

```
#pragma omp nome_da_diretiva  [cláusula,...] novalinha
```

1.2 - Condicional de compilação

Quando um código que contém diretivas do OpenMP é compilado por um compilador que não suporta o OpenMP (ou, no caso dos compiladores que suportam OpenMP, quando a opção de compilação que habilita o OpenMP não é utilizada) este simplesmente ignora as diretivas e compila o programa de forma seqüencial. Entretanto, esse código pode conter alguma instrução específica do OpenMP, como por exemplo, a chamada de funções que retornam informações do ambiente de execução relacionadas ao OpenMP e que estão definidas na `<omp.h>`. Nesse caso, o compilador não vai encontrar a definição dessas funções e o programa não será compilado.

Para tornar possível a compilação de um programa OpenMP tanto na versão paralela quanto na versão seqüencial, pode-se utilizar o condicional de compilação. Nesse caso, as funções do OpenMP ficam sob o controle de uma `#ifdef OPENMP` e só serão chamadas se essa macro estiver definida. O padrão requer que a macro `OPENMP` possua um valor do tipo yyyymm, onde yyyy é o ano e mm é o mês quando a versão específica do OpenMP foi lançada. Por exemplo, para o OpenMP 2.5, OPENMP é definido como 200505.

A seguir ilustra-se um exemplo da utilização do condicional de compilação.

```
#ifdef _OPENMP
#include <omp.h>
#else if
#define omp_get_thread_num() 0
#endif
      ...
int id = omp_get_thread_num();
```

1.3 - Construtor Paralelo

`#pragma omp parallel`

O construtor paralelo é a diretiva mais importante do OpenMP, uma vez que é o responsável pela indicação da região do código que será executada em paralelo. Se esse construtor não for especificado o programa será executado de forma seqüencial.

SINTAXE:

```
#pragma omp parallel [cláusula,...] novalinha
   Instrução
```

As cláusulas que podem ser utilizadas nesse construtor são as descritas abaixo e serão definidas posteriormente.

```
if (espressão lógica)
private (lista de variáveis)
shared (lista de variáveis)
firstprivate (lista de variáveis)
default (shared | none)
copyin (lista de variáveis)
reduction (operador: lista de variáveis)
num_threads (variável inteira)
```

Quando a *thread* inicial encontra um construtor paralelo ela cria um grupo de *threads* que irão executar o código e torna-se a *thread* mestre desse grupo. Porém, esse construtor não divide o trabalho entre as *threads*, apenas cria a região paralela.

Dentro de uma região paralela, quando as *threads* encontram outro construtor paralelo, cada uma delas cria um novo grupo de *threads* e torna-se a *thread* mestre desse novo grupo. Essas regiões são denominadas regiões paralelas aninhadas e por padrão são executadas de forma seqüencial, ou seja, o novo grupo criado contém apenas uma *thread*, que e a própria *thread* mestre do grupo.

Uma região paralela é chamada de inativa quando é executada por apenas uma *thread* e é chamada de ativa quando é executada por várias *threads*. Pode-se ativar uma região paralela aninhada através da função `omp_set_nested` (será definida posteriormente).

No final de toda região paralela existe uma barreira implícita que faz com que as *threads* esperem até que todas as *threads* cheguem naquele ponto. A partir daí, apenas a *thread* inicial continua a execução do código. Entretanto o openMP especifica uma cláusula que pode ser usada para que o programador decida sobre a existência dessa barreira.

Exemplo

A seguir é ilustrado um trecho de código que realiza a soma de dois vetores. Um construtor paralelo foi colocado no início do bloco de instruções

para indicar ao compilador a existência de uma região paralela. Note que essa diretiva não é suficiente para que o bloco seja executado de forma paralela.

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

A inserção do construtor paralelo indica que será criado um grupo de *threads* e que todas as *threads* desse grupo irão executar as mesmas instruções, ou seja, não há nenhuma divisão de trabalho.

1.4 - Construtores de compartilhamento de trabalho

Esses construtores representam a segunda diretiva mais importante do OpenMP, já que são responsáveis pela distribuição de trabalho entre as *threads*. Os construtores de compartilhamento de trabalho indicam a maneira como o trabalho será dividido entre as *threads*, porém não criam novas *threads*.

Para que a diretiva funcione de forma correta, ela deve estar dentro de uma região paralela, caso contrário, a diretiva é ignorada.

Existe uma barreira implícita no final do construtor, ou seja, todas as *threads* esperam até que a última *thread* finalize sua execução.

Em C/C++ existem três tipos de Construtores de Compartilhamento de Trabalho:

- Diretiva **`for`**
- Diretiva **`sections`**
- Diretiva **`single`**

Alguns cuidados devem ser tomados quando se utiliza esses construtores:

- Cada construtor deve ser encontrado por todas as *threads* ou por nenhuma delas.
- A seqüência de construtores e barreiras deve ser a mesma para todas as *threads* do grupo.

`#pragma omp for`

Esse construtor faz com que as iterações da estrutura de repetição situada logo abaixo da diretiva sejam executadas em paralelo. As iterações são distribuídas entre as *threads* do grupo criado na região paralela onde o laço se encontra.

SINTAXE:

```
#pragma omp for [cláusula,...]
     for-loop
```

As cláusulas que podem ser utilizadas no construtor `for` são as seguintes:

```
private (lista de variáveis)
firstprivate (lista de variáveis)
lastprivate (lista de variáveis)
reduction (operador: lista de variáveis)
schedule (tipo, [,tamanho do chunk])
nowait
ordered
```

Em C/C++, o construtor `for` só pode ser usado em estruturas de repetição no qual o número de iterações é previamente conhecido e não sofre alteração durante a execução. Ou seja, não se pode utilizar esse construtor em estruturas do tipo `while` ou `do-while`.

Vale ressaltar que o construtor `for` implementa **SIMD** (*Single Instruction Multiple Data*), ou seja, cada *thread* executa as mesmas instruções sob um conjunto diferente de dados.

Exemplo

O exemplo a seguir mostra o mesmo código do exemplo anterior, porém com um construtor `for` inserido antes do laço:

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

           printf("Thread %d executa a iteração %d do
                        loop\n",omp_get_thread_num(),i);
        }
  }
```

Dessa vez, como um construtor de compartilhamento de trabalho foi inserido no código, as iterações do laço serão divididas entre as *threads*, de forma que cada uma irá calcular alguns termos do vetor **c**. O padrão do OpenMP é dividir as iterações do laço igualmente e de forma ordenada entre as *threads*. Por exemplo, se existirem 12 iterações e 3 *threads*, uma *thread* fica com as 4 primeiras iterações, a outra com as 4 iterações seguintes e última *thread* com as últimas 4 iterações. Caso a divisão do número de iterações pelo número de *threads* não seja exata, o resto da divisão é distribuído igualmente

entre algumas *threads*. Por exemplo, se existirem 19 iterações e 4 *threads*, três *threads* ficarão com 5 iterações e uma *thread* ficará com as últimas 4 iterações. Porém, pode-se alterar a forma como as iterações são distribuídas entre as *threads* por meio da cláusula `schedule` (será definida posteriormente).

Além disso, cada *thread* irá imprimir seu identificador (um valor inteiro retornado pela função `omp_get_thread_num()`) e a iteração que a mesma está executando. Como o programa será executado de forma paralela, não se pode esperar que as iterações sejam impressas na ordem seqüencial. Dessa forma, ao executar o código acima, pode-se esperar como possível resultado a saída abaixo.

```
Thread 0 executa a iteração 0
Thread 0 executa a iteração 1
Thread 0 executa a iteração 2
Thread 3 executa a iteração 7
Thread 3 executa a iteração 8
Thread 1 executa a iteração 5
Thread 1 executa a iteração 6
Thread 2 executa a iteração 3
Thread 2 executa a iteração 4
```

`#pragma omp sections`

O construtor `sections` é utilizado para dividir tarefas entre as *threads* em blocos de código que não possuem iterações. Dessa forma, cada *thread* irá executar um bloco de código diferente.

SINTAXE:

```
#pragma omp sections[cláusula,...] novalinha
{
  #pragma omp section novalinha
  instrução
  #pragma omp section novalinha
  instrução
}
```

Além do construtor `sections`, que indica que cada thread irá executar um bloco de instruções diferentes, é necessário incluir no código a diretiva `#pragma omp section`, que indica qual a instrução que cada thread irá executar.

As cláusulas que podem ser utilizadas no construtor `sections` são as seguintes:

```
private (lista de variáveis)
firstprivate (lista de variáveis)
lastprivate (lista de variáveis)
reduction (operador: lista de variáveis)
nowait
```

Quando houver mais blocos de código do que *threads*, algumas *threads* irão executar mais de um bloco. Por outro lado, se houver mais *threads* do que tarefas a serem executadas, apenas algumas *threads* irão trabalhar e as outras ficarão ociosas. Se houver apenas uma *thread* na região paralela, ela vai executar todas as tarefas de forma seqüencial.

Não se pode garantir a ordem de execução dos blocos. Mesmo quando houver apenas uma *thread* (região paralela inativa), as funções podem ser executadas fora de ordem.

Enquanto o construtor `for` implementa **SIMD**, o construtor `sections` implementa **MIMD** (*Multiple Instruction Multiple Data*), ou seja, cada *thread* executa uma instrução diferente sob um conjunto diferente de dados.

A figura 9 ilustra o funcionamento do construtor `sections`.

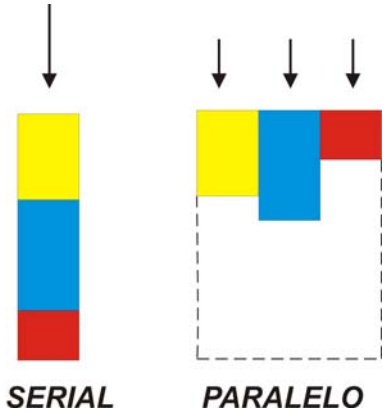

Figura 9 – Funcionamento de uma região que possui o construtor `sections`

Exemplo

No exemplo a seguir, o construtor `sections` foi utilizado para que duas operações diferentes sobre os vetores a e b fossem executadas ao mesmo tempo.

```
#pragma omp parallel
{
    #pragma omp sections
    {
       #pragma omp section
          soma_vetores (a,b,c,n);

       #pragma omp section
          subtrai_vetores (a,b,d,n);
        }
}
     . . .
void soma_vetores(double *a, double *b, double *c, int n)
{
     for (int i = 0; i < n; i++)
       c[i] = a[i]+b[i];
}
```

```
void subt_vetores(double *a, double *b, double *d, int n)
{
     for (int i = 0; i < n; i++)
       d[i] = a[i]-b[i];
}
```

De acordo com a definição do construtor `sections`, o código acima será executado por duas *threads*, uma delas irá executar a função `soma_vetores` e a outra irá executar a função `subtrai_vetores`. Vale ressaltar que as duas funções serão executadas simultaneamente, exceto se existir apenas uma *thread*.

`#pragma omp single`

Esse construtor indica que o trecho de código abaixo dessa diretiva deve ser executado apenas por uma *thread*, não necessariamente a *thread* mestre, mas a primeira *thread* que atingir esse ponto de execução.

As outras *threads* esperam numa barreira implícita, no final do construtor `single`, até que a *thread* que encontrou o construtor termine a execução.

SINTAXE:

```
#pragma omp single [cláusula,...]
    instrução ...
```

As cláusulas que podem ser utilizadas no construtor `single` são as seguintes:

```
private (lista de variáveis)
firstprivate (lista de variáveis)
copyprivate (lista de variáveis)
nowait
```

Exemplo

Para exemplificar a utilização do construtor `single`, será utilizado o mesmo código do exemplo do construtor `for`, com apenas algumas modificações.

```
#pragma omp parallel
{
     #pragma omp single
          printf("Inicio da região paralela …
                  número de threads = %d\n",
                   omp_get_num_threads());

     #pragma omp for
        for (i = 0; i < n; i++)
```

```
            {
              c[i] = a[i]+b[i];
              printf("Thread %d executa a iteracao %d do
                        loop\n",omp_get_thread_num(),i);
            }
      }
```

No código acima, o construtor `single` foi utilizado para que apenas uma *thread* imprimisse na tela algumas informações da execução do código.

1.5 - Construtores Combinados

Quando a região paralela possui apenas um construtor de compartilhamento de trabalho, podem-se combinar os construtores paralelo e de compartilhamento de trabalho em uma apenas uma diretiva, conforme é mostrado abaixo:

SINTAXE:

```
#pragma omp parallel for   [cláusula,...]
      for-loop

#pragma omp parallel sections   [cláusula,...]
      novalinha
{
      #pragma omp section novalinha
            instrução
      #pragma omp section novalinha
          instrução
}
```

Os construtores combinados permitem todas as cláusulas do construtor paralelo e dos construtores de compartilhamento de trabalho (`for` e `section`) com os mesmos significados e restrições, exceto a cláusula `nowait`.

Podem existir algumas vantagens de desempenho com essa combinação: quando o construtor combinado é utilizado, o compilador sabe o que esperar e pode gerar um código mais eficiente. Por exemplo, ele não irá inserir mais do que uma barreira no final da região paralela.

1.6 - Diretivas de sincronização

No modelo de memória do OpenMP, uma variável pode ser do tipo `private` (privada) ou do tipo `shared` (compartilhada). Uma vez que as variáveis compartilhadas são visíveis por todas as *threads* que executam o código, o acesso as mesmas deve acontecer de maneira organizada e sincronizada a fim de evitar as chamadas "condições de corrida" (*race conditions*).

Condições de corrida são situações que acontecem quando duas ou mais *threads* tentam atualizar, ao mesmo tempo, uma mesma variável ou quando uma *thread* atualiza uma variável e outra *thread* acessa o valor dessa variável ao mesmo tempo. Quando uma condição de corrida acontece, o

resultado vai depender da ordem de execução das *threads* e não há nenhuma garantia que a variável seja atualiza com o valor correto.

Dessa forma, as diretivas de sincronização garantem que o acesso ou atualização de uma determinada variável compartilhada aconteça no momento certo.

`critical`

O construtor `critical` restringe a execução de uma determinada tarefa a apenas uma *thread* por vez. É utilizado para evitar condições de corrida, ou seja, que uma mesma variável compartilhada seja atualizada por mais de uma *thread* ao mesmo tempo.

SINTAXE:

```
#pragma omp critical [(nome)] novalinha
    Instrução
```

Quando uma *thread* encontra uma sessão crítica, ela espera no início da mesma até que nenhuma *thread* esteja executando as instruções da região critica de mesmo nome. Quando nenhuma *thread* estiver executando essa sessão crítica, a *thread* que estava esperando entra na região e executa as instruções, conforme é ilustrado na figura 10.

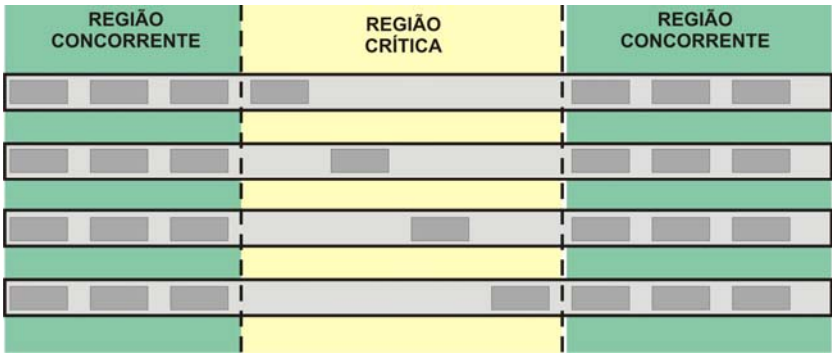

Figura 10 – Funcionamento de uma região crítica

Exemplo

Para exemplificar a utilização da diretiva `critical`, será utilizado um código que realiza o produto escalar de dois vetores.

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

      #pragma omp critical
```

```
            dot += aux_dot;

        }
```

Nesse exemplo, cada *thread* calcula uma parcela do produto escalar (**aux_dot**). Em seguida a variável **dot** é atualizada com os valores calculados por cada uma das *threads*. O construtor `critical` é utilizado para impedir que mais de uma *thread* tente atualizar a variável **dot** ao mesmo tempo.

`atomic`

As atualizações para as variáveis compartilhadas não são atômicas. Isto significa que se duas *threads* tentarem fazer isto ao mesmo tempo, uma das atualizações pode ser perdida.

O construtor `atomic` permite que certa região da memória seja atualizada atomicamente, impedindo que várias *threads* acessem essa região ao mesmo tempo. Ele também é um construtor que habilita múltiplas *threads* a atualizarem dados compartilhados sem interferência. Essa diretiva é usada para proteger uma atualização única para uma variável compartilhada, sendo utilizada para uma única instrução.

É importante destacar que essa restrição é aplicada a todas as *threads* que executam o programa, não somente as *threads* pertencentes a um mesmo bloco (time).

SINTAXE:

```
#pragma omp atomic
      Instrução
```

O construtor `atomic` aplica-se apenas à instrução localizada logo abaixo da diretiva. O formato dessa instrução é ilustrado abaixo. A utilização do construtor `atomic` é muito parecida com a utilização do construtor `critical`, porém permite uma melhor otimização. É importante destacar que em operações de incremento, por exemplo, o construtor `atomic` é mais eficiente quando comparado ao construtor `critical`.

x binop (+,,-,',&,^, |,<<,>>) = expr*
x++
++*x*
x--
--*x*

Exemplo

```
int ic, i;
ic = 0;
#pragma omp parallel shared(n, ic) private(i)
for(i=0;i<5;i++)
```

```
{
   #pragma omp atomic
      ic++;
}
printf("counter = %d\n", ic);
```

barrier

A diretiva `barrier` é utilizada para sincronizar todas as *threads* em um determinado ponto do código. Quando uma *thread* encontra uma barreira, ela espera até que todas as *threads* cheguem naquele ponto e, a partir daí, elas continuam a execução do código ao mesmo tempo, conforme é ilustrado na figura 11.

SINTAXE:

```
#pragma omp barrier
     novalinha
```

Em alguns pontos do código não é necessário à colocação de barreiras, uma vez que já existem barreiras implícitas definidas, como, por exemplo, no final dos construtores `parallel`, `for`, `sections` e `single`.

Em C/C++ existem algumas restrições quanto à posição da barreira no código. Ela deve ser colocada em um lugar no qual se a mesma for deletada ou ignorada, o programa permanece com a sintaxe correta.

Uma das principais utilizações da barreira é evitar condições de corrida, pois ela pode impedir que uma *thread* acesse uma variável antes que ela seja atualizada.

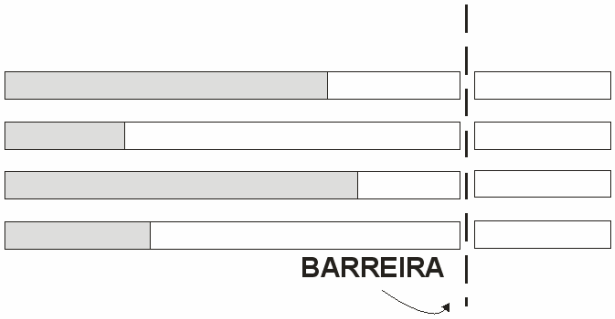

Figura 11 – Esquema ilustrativo de uma barreira

Exemplo

O código a seguir ilustra a utilização da diretiva `barrier`:

```
#pragma omp parallel private(id)
{
id = omp_get_thread_num();

if( id < omp_get_num_threads()/2) system("sleep 3");
     print_time(id,"antes");
```

```
    #pragma omp barrier

    print_time(id,"depois");
    }
```

No código acima, utilizou-se uma barreira para que as *threads* cujo **id** é maior que `omp_get_num_threads()/2` esperem as *threads* que terão que executar a função `sleep`.

`flush`

A diretiva `flush` é utilizada para garantir que todas as *threads* tenham acesso ao valor correto de uma variável compartilhada que foi recentemente atualizada.

Quando uma variável compartilhada é atualizada por uma *thread*, o novo valor dessa variável fica inicialmente armazenado somente na memória *cache* do processador, pois o acesso a essa memória é bem mais rápido que o acesso à memória principal. Entretanto, a memória *cache* de um processador não é visível pelos outros processadores. Dessa forma, quando as *threads* de outros processadores precisarem acessar o valor atualizado dessa variável, o mesmo pode ainda não estar disponível na memória compartilhada e as *threads* terão acesso a um valor desatualizado.

O padrão OpenMP especifica que todas as modificações em variáveis compartilhadas sejam escritas na memória principal em pontos de sincronização específicos, tornando-se, dessa forma, acessíveis a todas as *threads*. Entretanto, nas regiões localizadas entre os pontos de sincronização, não há nenhuma garantia de que as variáveis sejam atualizadas instantaneamente na memória principal. Para garantir essa atualização nos trechos entre os pontos de sincronização, utiliza-se a diretiva `flush`.

SINTAXE:
```
#pragma omp flush [(lista de variáveis)] novalinha
```

A diretiva `flush` atualiza na memória principal os valores das variáveis presentes na lista de variáveis. Se não for fornecida nenhuma lista, a diretiva atualiza todas as variáveis compartilhadas.

Existem flush's implícitos em alguns locais específicos das diretivas do OpenMP, tais como:

- Em todas as barreiras explicitas e implícitas.
- Na entrada e na saída de uma região critica.
  Na entrada e na saída de uma diretiva `ordered`.
- Na entrada e na saída de rotinas `lock` (abordadas posteriormente).
- Na saída de um construtor de divisão de trabalho.

Exemplo

No código abaixo, cada *thread* atualiza uma única posição dos vetores **v** e **w**. Em seguida, a diretiva `flush` é utilizada para atualizar os valores dos vetores **v** e **w** na memória principal. Dessa forma, assegura-se que todas as *threads* tenham acesso aos valores corretos de todas as posições dos vetores, uma vez que os mesmos serão utilizados nas instruções seguintes.

```
#pragma omp parallel private(id) shared(n,p,v,w,k)
  {
     id = omp_get_thread_num();

     v[id] = id*(n+1)
     w[id] = id*(p+1)

     #pragma omp flush(v);

     #pragma omp for
        for (i = 0; i < omp_get_num_threads(); i++)
        {
             k[i]= v[i]+w[i];
        }
  }
```

`ordered`

O construtor `ordered` permite que um laço seja executado na ordem seqüencial, como ilustra a figura 12.

SINTAXE:

```
#pragma omp ordered  novalinha
        Instrução
```

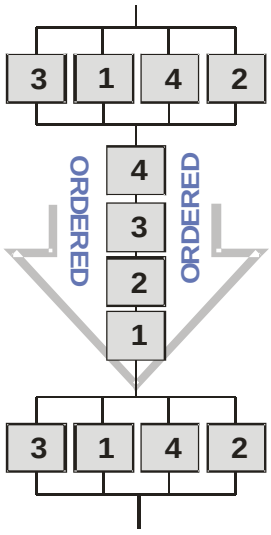

Figura 12 – Funcionamento de uma região que possui o construtor `ordered`

Exemplo

A seguir é mostrado o mesmo exemplo utilizado para ilustrar a utilização do construtor `for`, porém com uma simples modificação. Antes da função para impressão (`printf`) foi inserido um construtor `ordered` para que as iterações sejam impressas na ordem seqüencial, conforme se observa abaixo:

```
#pragma omp parallel
{
   #pragma omp for ordered
     for (i = 0; i < n; i++)
     {
         c[i] = a[i]+b[i];

         #pragma omp ordered
         printf("Thread %d executa a iteração %d do
                   loop\n",omp_get_thread_num(),i);
  }
```

```
Thread 0 executa a iteração 0
Thread 0 executa a iteração 1
Thread 0 executa a iteração 2
Thread 2 executa a iteração 3
Thread 2 executa a iteração 4
Thread 1 executa a iteração 5
Thread 1 executa a iteração 6
Thread 3 executa a iteração 7
Thread 3 executa a iteração 8
```

`master`

O construtor `master` define um bloco de código que será executado apenas pela *thread* mestre. É válido ressaltar que o construtor `master` não tem barreira implícita nem na entrada nem na saída.

SINTAXE:

```
#pragma omp master  novalinha
     Instrução
```

Exemplo

O exemplo a seguir é o mesmo que foi utilizado para exemplificar o uso do construtor `single`, entretanto, no lugar do mesmo, utilizou-se o construtor `master`.

```
#pragma omp parallel
{
   #pragma omp master
        n = 10;
   #pragma omp barrier

   #pragma omp for
      for (i = 0; i < n; i++){
          c[i] = a[i]+b[i];
          printf("Thread %d executa a iteracao %d do
                    loop\n",omp_get_thread_num(),i);
      }
  }
```

No código acima, o construtor mestre foi utilizado para que apenas a *thread* mestre atualize a variável **n**. Como essa variável é compartilhada, todas as *threads* terão acesso ao valor atualizado que será utilizado logo em seguida.

`threadprivate`

A diretiva `threadprivate` especifica quais variáveis serão privadas em todo o escopo do código, ou seja, em todas as regiões paralelas.

SINTAXE:

```
#pragma omp threadprivate (lista de variáveis) novalinha
```

Cada cópia de uma variável do tipo `threadprivate` é inicializada uma vez, da maneira especificada pelo programa, porém em um ponto indefinido antes da primeira referenciar à variável.

Nos trechos seqüenciais e nos construtores `master`, apenas as cópias da *thread* mestre das variáveis `threadprivate` são acessíveis.

Uma variável definida como `threadprivate` não pode estar presente em nenhuma lista de variáveis de cláusulas, exceto nas listas das cláusulas `copyin`, `copyprivate`, `schedule`, `num_threads`, ou `if` (essas cláusulas serão definidas posteriormente).

Pode-se garantir que os valores das cópias das variáveis `threadprivate` das outras *threads* (exceto a *thread* mestre) irão persistir entre duas regiões paralelas consecutivas apenas se todas as condições a seguir forem satisfeitas:

- Não houver nenhuma região paralela aninhada dentro de outra região paralela.
- O número de *threads* usado na execução de ambas as regiões paralelas for o mesmo.
- O ajuste dinâmico do número de *threads* estiver desabilitado na entrada da primeira região paralela e permanece desabilitado até a entrada da segunda região paralela.

Exemplo

O exemplo a seguir ilustra como usar a diretiva `threadprivate` para que cada *thread* tenha um contador separado.

```
int counter;

#pragma omp threadprivate(counter)
int increment_counter()
{
   counter++;
   return(counter);
}
```

2 - CLÁUSULAS

As cláusulas definem o comportamento dos construtores aos quais estão associadas e das variáveis envolvidas na execução da região paralela.

Além de outras atribuições, as cláusulas definem quais variáveis são compartilhadas entre as *threads* e quais são privadas. Por padrão, as variáveis são consideradas compartilhadas, exceto as que estão especificadas em uma diretiva `threadprivate` e as que são declaradas dentro da região paralela.

A maioria das cláusulas é aplicada às variáveis presentes na lista de variáveis que acompanha a mesma. Uma variável pode estar presente em apenas uma cláusula por diretiva, exceto as cláusulas `firstprivate` e `lastprivate` que podem conter a mesma variável nas suas listas.

Nem todas as cláusulas se aplicam a todas as diretivas. A seguir serão definidas todas as cláusulas do OpenMP, juntamente com uma listagem das diretivas às quais cada cláusula pode estar associada.

`shared`

A cláusula `shared` indica quais variáveis terão endereço de memória acessível por todas as *threads* dentro da região paralela associada à cláusula. Ou seja, as variáveis `shared` podem ser acessadas e modificadas por todas as *threads* do grupo dentro da região paralela.

SINTAXE:

```
shared (lista de variáveis)
```

As diretivas que podem utilizar essa cláusula são as seguintes:

```
#pragma omp parallel
#pragma omp parallel for
#pragma omp parallel sections
```

`private`

A cláusula `private` indica que as variáveis presentes na sua lista são do tipo privada, ou seja, cada *thread* possui acesso exclusivo a uma cópia dessas variáveis.

SINTAXE:

```
private (lista de variáveis)
```

Quando uma *thread* faz uma modificação na sua cópia da variável privada, ela não influencia no valor da mesma variável privada das outras *threads*.

Algumas observações podem ser listadas com relação a variáveis privadas:

- Quando uma variável é criada dentro de uma região paralela, a mesma é, por definição, privada.
- Por definição, a variável que registra as iterações de um laço é privada.
- Os valores das variáveis privadas no início e no final da região paralela são indefinidos.

As diretivas que podem utilizar essa cláusula são as seguintes:

```
#pragma omp parallel
#pragma omp for
#pragma omp sections
#pragma omp single
#pragma omp parallel for
#pragma omp parallel sections
```

Exemplo

O exemplo a seguir ilustra a utilização das cláusulas `shared` e `private`.

```
#pragma omp parallel shared (a,b,c,n) private(id,i)
  {
      #pragma omp for
       for (i = 0; i < n; i++)
       {
           id = omp_get_thread_num();
           c[i] = a[i]+b[i];

          printf("Thread %d executa a iteração %d
                do loop\n",id,i);
       }
  }
```

`firstprivate`

Da mesma forma que a cláusula `private`, a cláusula `firstprivate` indica quais variáveis serão privadas.

As variáveis definidas como `private` têm seus valores indefinidos na entrada do construtor ao qual está associada. Já as variáveis presentes na lista da cláusula `firstprivate` entram na região paralela com os valores que possuíam antes de encontrar o construtor ao qual à cláusula está associada.

SINTAXE:

```
firstprivate (lista de variáveis)
```

As diretivas que podem utilizar essa cláusula são as seguintes:

```
#pragma omp parallel
#pragma omp for
#pragma omp sections
#pragma omp single
#pragma omp parallel for
#pragma omp parallel sections
```

Exemplo

O exemplo a seguir ilustra a utilização da cláusula `firstprivate`.

```
for (i = 0; i < 3; i++)
   c[i] = -i;

value = 3;

#pragma omp parallel private(id,i) shared(c,n)
                                   firstprivate(value)
{
     id = omp_get_thread_num();

    value +=n*id;

    for (i = value; i < value+n; i++)
           c[i] = i;
  }
```

No exemplo acima, a cláusula `firstprivate` é utilizada para que cada cópia da variável **value** entre na região paralela com o valor que foi atualizado para a mesma fora da região paralela. Dentro da região, entretanto, cada *thread* atualiza sua cópia da variável **value** com um valor específico.

`lastprivate`

A cláusula `lastprivate` também indica quais variáveis serão consideradas como privadas dentro da região definida pelo construtor. Além disso, essa cláusula permite que a variável saia da região definida pelo construtor com o último valor que foi atualizado para a mesma dentro da região.

SINTAXE:

```
lastprivate (lista de variáveis)
```

As diretivas nas quais essa cláusula pode ser utilizada são as seguintes:

```
#pragma omp for
#pragma omp sections
#pragma omp parallel for
#pragma omp parallel sections
```

Quando essa cláusula é utilizada em um construtor `#pragma omp for`, a variável é atualizada com o valor atribuído à variável na última iteração do laço da ordem seqüencial. Quando a cláusula é utilizada em um construtor `#pragma omp sections`, a variável é atualiza com o valor que foi atribuído à mesma na última seção do construtor.

Exemplo

No trecho de código abaixo é ilustrada a utilização da cláusula `lastprivate`.

```
#pragma omp parallel for shared(a,b,c,n) lastprivate (i)
{
      for (i = 0; i < n; i++)
              c[i] = a[i]+b[i];
}

printf("Depois da região paralela ... i = %d\n",i);
```

O valor da variável **i** fora da região paralela (valor que será impresso) será o valor utilizado na última iteração do laço, ou seja, **n-1**.

`default`

A cláusula `default` define um padrão de compartilhamento de dados para as variáveis.

SINTAXE:

```
default (none |shared)
```

O `default(shared)` indica que todas as variáveis da região paralela definida pelo construtor serão compartilhadas entre as *threads*. Para definir as exceções devem-se utilizar as cláusulas `private`, `firstprivate` e `lastprivate`.

Se `default(none)` for utilizado, o programador deve especificar qual será o comportamento de cada uma das variáveis no construtor.

Caso a cláusula default não esteja presente no construtor, o comportamento padrão é o mesmo que o definido por `default(shared)`.

Uma restrição da cláusula `default` é que ela só pode estar presente no construtor paralelo.

`reduction`

A cláusula `reduction` especifica que uma ou mais variáveis que são privadas de cada *thread* serão submetidas a uma operação de redução no final da região definida pelo construtor ao qual à cláusula está associada.

Nessa cláusula são especificados um operador e uma lista de variáveis. Uma cópia privada de cada variável da lista é criada para cada uma das *threads*, da mesma forma como aconteceria se a cláusula `private` tivesse sido utilizada. Cada cópia é então inicializada com o valor de inicialização definido pelo operador (ver quadro abaixo).

No final da região definida pelo construtor ao qual a cláusula está associada, a lista de variáveis original é atualizada com os valores finais das cópias privadas, usando o operador especificado.

SINTAXE:

```
reduction (operador: lista de variáveis)
```

A tabela a seguir mostra os operadores válidos e seus respectivos valores de inicialização:

Operador	Valor inicial
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Os valores das variáveis originais da lista tornam-se indefinido quando a primeira *thread* encontra o construtor que especifica a cláusula e permanece dessa forma até que a operação de redução seja finalizada. Normalmente, a operação de redução é finalizada no final do construtor, entretanto, se houver uma cláusula `nowait` no construtor, os valores originais permanecerão indefinidos até que uma barreira seja encontrada.

Algumas restrições da cláusula `reduction` devem ser observadas:

- O tipo das variáveis presentes da lista de variáveis deve ser compatível com o operador de redução.
- Vetores, ponteiros e referências não podem aparecer na cláusula `reduction`.
- Uma variável especificada no `reduction` não deve ser do tipo `const`.
- Variáveis que são privadas dentro de uma região paralela ou que aparecem na cláusula `reduction` de uma diretiva paralela não pode estar especificada em uma cláusula `reduction` de uma diretiva de compartilhamento de trabalho inserida no construtor paralelo.

As diretivas nas quais a cláusula `reduction` pode estar presente são as seguintes**:**

```
#pragma omp parallel
#pragma omp for
#pragma omp sections
#pragma omp parallel for
#pragma omp parallel sections
```

Exemplo

O trecho de código mostrado abaixo calcula o produto escalar de dois vetores. Esse mesmo problema já foi solucionado utilizando-se a diretiva `critical`, entretanto pode-se obter um código mais robusto com a utilização da cláusula `reduction`.

```
#pragma omp parallel
{
     #pragma omp for shared(a,b,c,n) private(i)
                                          /reduction(+:dot)
     for (i = 0; i < n; i++)
     {
             dot += a[i]*b[i];
     }
}
```

No código acima, cada *thread* calcula sua parcela do produto escalar e armazena na sua cópia da variável **dot**. No final da região paralela, a cópia original da variável **dot** é atualizada com a soma dos valores calculados por todas as *threads*.

`copyin`

A cláusula `copyin` permite que as cópias de cada *thread* de uma variável definida como `threadprivate` sejam iniciadas com o mesmo valor. O valor da variável da *thread* mestre é copiado, no início da região paralela, para as variáveis correspondentes das outras *threads*.

SINTAXE:

```
copyin (lista de variáveis)
```

Essa cláusula pode ser utilizada apenas no construtor paralelo e nos construtores combinados entre construtor paralelo e de divisão de trabalho.

Exemplo

No exemplo abaixo, as variáveis **tol** e **size** são inicializadas pela *thread* mestre e a cláusula `copyin` é utilizada para copiar esses valores nas cópias das variáveis das outras *threads*.

```
#pragma omp threadprivate (tol,size)

tol = t;
size = n;

#pragma omp parallel copyin(tol,size)
{
     work = (int*)malloc(size*sizeof(int));

     for (i = 0; i < size; i++)
           work[i] = tol;
}
```

`copyprivate`

A cláusula `copyprivate` é utilizada para transmitir o valor de uma variável privada de uma *thread* para as outras *threads* do grupo.

SINTAXE:

```
copyprivate (lista de variáveis)
```

Essa cláusula é comumente utilizada quando uma *thread* lê ou inicializa uma variável privada cujo valor, em seguida, será utilizado pelas outras *threads*.

Uma restrição da `copyprivate` é que ela só pode ser aplicada no construtor `single`.

Exemplo

O exemplo abaixo mostra como a cláusula `copyprivate` pode ser utilizada.

```
#pragma omp threadprivate (a,b)
#pragma omp parallel
{
```

```
            #pragma omp single copyprivate(a,b)
                    scanf("%d   %d", &a, &b);
    }
```

Um construtor `single` foi inserido no código para que apenas uma *thread* capture os valores das variáveis **a** e **b**. Utilizou-se, então, a cláusula `copyprivate` para que todas as cópias das variáveis **a** e **b** sejam atualizadas com os valores lidos pela função `scanf`.

`nowait`

Como já foi visto anteriormente, existem barreias implícitas no final da maioria dos construtores do OpenMP. Porém, algumas vezes essas barreiras são desnecessárias e acabam comprometendo o desempenho da aplicação. Nesses casos, pode-se utilizar a cláusula `nowait`, uma vez que ela faz com que essas barreiras sejam ignoradas pelas *threads*. Se existir uma cláusula `nowait` no construtor, quando as *threads* chegam ao final da região definida pelo mesmo, elas continuam a execução do código sem esperar pelas outras.

SINTAXE:

```
                                nowait
```

As diretivas nas quais a cláusula `nowait` pode estar presente são as seguintes**:**

```
#pragma omp for
#pragma omp sections
#pragma omp single
#pragma omp parallel for
#pragma omp parallel sections
```

Exemplo

No trecho de código abaixo, a cláusula `nowait` foi utilizada para suprimir a barreira implícita no final do primeiro construtor `for`, uma vez que a mesma é desnecessária já que não há dependência de dados entre os dois construtores.

```
#pragma omp parallel
  {
       #pragma omp for nowait
         for (i = 0; i < n; i++)
                c[i] = a[i]+b[i];

       #pragma omp for nowait
         for (i = 0; i < n; i++)
              d[i] = a[i]*b[i];
  }
```

schedule

A cláusula `schedule` é utilizada apenas no construtor `for` e controla a forma como as iterações são distribuídas entre as *threads*. A utilização adequada dessa cláusula pode ter grande impacto no desempenho da aplicação.

SINTAXE:

```
schedule(tipo,[chunk_size])
```

Existem quatro tipos diferentes de `schedule` definidos no padrão OpenMP. Os mesmos são descritos a seguir:

static

Nesse caso, as iterações são divididas em pedaços de tamanho definido pelo parâmetro *`chunk_size`*. A distribuição dos blocos de iteração entre as *threads* é feita de forma estática e cíclica, respeitando a ordem de numeração das mesmas. Se a divisão do número de iterações pelo *`chunk_size`* não for exata, o último bloco terá um número menor de iterações. Quando o *`chunk_size`* não é especificado, cada *thread* fica com um bloco de tamanho aproximadamente igual ao número de iterações dividido pelo número de *threads*. Quando a cláusula `schedule` não é utilizada no construtor for, o padrão do OpenMP é o mesmo definido pelo `static`.

dynamic

Nesse caso, as iterações são distribuídas entre as *threads* à medida que as mesmas solicitam mais iterações. Cada *thread* executa um bloco de iterações (do tamanho definido pelo parâmetro *`chunk_size`*), em seguida solicita outro bloco até que já não existam mais blocos de iterações. Quando o *`chunk_size`* não é especificado, o padrão é 1 (um).

guided

Da mesma forma que o `dynamic`, as iterações são distribuídas entre as *threads* à medida que as mesmas solicitam iterações. Cada *thread* executa um bloco de iterações (do tamanho definido pelo parâmetro `chunk_size`), em seguida solicita outro bloco até que já não existam mais blocos de iterações.

Para um tamanho de bloco igual a 1, o tamanho de cada bloco é proporcional ao número de iterações não executadas, dividido pelo número de *threads*, decaindo até 1.

Para um tamanho de bloco "k" (k>1), o tamanho de cada bloco é determinado da mesma forma, com a restrição de que o bloco não pode conter um número de iterações menor que k , com exceção do último bloco a ser executado, que pode ter número de iterações menor que k.

Quando o *`chunk_size`* não é especificado, o padrão é 1.

`runtime`

Quando essa opção de schedule é utilizada, a decisão sobre o tipo de schedule a ser utilizado é feita na hora da execução do código. O tipo de schedule e o tamanho do bloco (opcional) são definidos através da variável de ambiente `OMP_SCHEDULE` (será definida posteriormente).

Exemplo

Para exemplificar a utilização da cláusula `schedule`, utilizou-se o exemplo da soma de dois vetores já abordado anteriormente. Dessa vez, o tipo `dynamic` da cláusula `schedule` foi utilizado para dividir as iterações entre as *threads*.

```
#pragma omp parallel
  {
      n = 10;

     #pragma omp for schedule(dynamic,2)
        for (i = 0; i < n; i++)
        {
           c[i] = a[i]+b[i];

           printf("Thread %d executa a iteração %d do
                        loop\n",omp_get_thread_num(),i);
        }
  }
```

De acordo com a descrição do tipo `dynamic`, cada *thread* irá executar um bloco de duas iterações e em seguida irá solicitar mais um bloco de iterações, até que todas as iterações já tenham sido executadas. É importante ressaltar que a distribuição das iterações não segue a ordem de numeração das *threads*. Se considerarmos que três *threads* estarão executando o código acima, a impressão dos resultados poderá ser do tipo:

```
Thread 0 executa a iteração 0 do loop
Thread 0 executa a iteração 1 do loop
Thread 1 executa a iteração 4 do loop
Thread 1 executa a iteração 5 do loop
Thread 2 executa a iteração 2 do loop
Thread 2 executa a iteração 3 do loop
Thread 2 executa a iteração 8 do loop
Thread 2 executa a iteração 9 do loop
Thread 1 executa a iteração 6 do loop
Thread 1 executa a iteração 7 do loop
```

`if`

A cláusula `if` indica uma condição na qual o construtor paralelo será executado.

SINTAXE:

```
if (espressão lógica)
```

Se a expressão lógica for verdadeira, ou seja, se tiver positivo, então o construtor será executado. Se a expressão for falsa, o bloco de código será executado por apenas uma *thread* (região paralela inativa).

Exemplo

O exemplo abaixo ilustra a utilização da cláusula `if`.

```
#pragma omp parallel if (n > 100*var)
{
       #pragma omp for
         for (i = 0; i < n; i++)
                c[i] = a[i]+b[i];
}
```

No exemplo acima, a região paralela só será ativa, ou seja, só será executada por mais de uma *thread*, se a variável **n** satisfazer a condição a condição indicada na cláusula `if`. Caso contrário, o bloco de código dentro da região será executado de forma seqüencial.

A única diretiva que pode utilizar a cláusula `if` é o construtor paralelo.

`ordered`

Se dentro de um construtor `for` existir um construtor `ordered`, então a cláusula `ordered` deve ser utilizada nesse construtor `for`.

SINTAXE:

```
ordered
```

Exemplo

No exemplo a seguir, a cláusula `ordered` foi utilizada no construtor paralelo, pois existe uma diretiva `ordered` dentro da região paralela.

```
#pragma omp parallel for ordered private(id)shared(n,a,b)
for (i=0;i<n;i++){
  id = omp_get_thread_num();
   a[i] += i;
```

```
        printf("Thread %d atualiza a[%d] =
                                        %d\n",id,i,a[i]);
        #pragma omp ordered
        printf("Thread %d imprime valor de  a[%d] =
                                         %d\n",id,i,a[i]);}
```

```
Thread 0 atualiza a[0];
Thread 0 imprime o valor de a[0] = 0;
Thread 2 atualiza a[4];
Thread 1 atualiza a[2];
Thread 1 atualiza a[3];
Thread 0 atualiza a[1];
Thread 0 imprime o valor de a[1] = 1;
Thread 1 imprime o valor de a[2] = 3;
Thread 1 imprime o valor de a[3] = 6;
Thread 2 atualiza a[5];
Thread 2 imprime o valor de a[4] = 10;
Thread 2 imprime o valor de a[5] = 15;
```

num_threads

A cláusula num_threads é utilizada para especificar o número de *threads* que irá executar as instruções da região paralela definida pelo construtor ao qual a cláusula está associada. Pode ser utilizada apenas no construtor paralelo.

SINTAXE:

```
num_threads (valor inteiro)
```

3 – FUNÇÕES DE INTERFACE

O padrão OpenMP disponibiliza três grupos de funções que estão declaradas no arquivo de cabeçalho <omp.h>. Um deles está relacionado ao controle do ambiente de execução, outro relacionado às funções que são responsáveis pela sincronização do acesso aos dados e o terceiro relacionado à medição de tempo das aplicações.

3.1 - Funções de ambiente de execução

omp_set_num_threads()

A função omp_set_num_threads() insere um número padrão de *threads* a serem usadas nas regiões paralelas em que não está definida a cláusula num_threads.

SINTAXE:

```
void omp_set_num_threads(int numthreads)
```

Esta função tem precedência sobre a variável de ambiente `OMP_NUM_THREADS` e somente terá efeito se o ajuste dinâmico do número de *threads* estiver habilitado. Se o valor retornado da chamada da função for diferente de zero, o comportamento da função será indefinido.

O valor padrão do número de *threads* pode ser facilmente anulado pela simples especificação da cláusula `num_threads` na diretiva paralela.

omp_get_num_threads()

Essa função retorna o número de *threads* ativas na região paralela onde a função foi chamada. Como citado, se o número de *threads* não for definido pelo usuário, então ele será definido pela implementação. O retorno dessa função será 1 se ela for chamada de um alguma parte do código que seja serial.

SINTAXE:
```
int omp_get_num_threads(void)
```

omp_get_max_threads()

O número de *threads* retornado por essa função é igual ao número de processadores/*cores* disponíveis, a não ser que o número de *threads* tenha sido redefinido pela variável de ambiente `NUM_THREADS` ou pela chamada da função `omp_set_num_threads()`.

Por exemplo, quando as funções `omp_get_num_threads()` e `omp_get_max_threads()` são chamadas em um trecho do código seqüencial, a primeira irá retornar o valor 1 e a segunda irá retornar o número de cores disponíveis (ou o número redefinido pela variável de ambiente ou pela chamada da função).

SINTAXE:
```
int omp_get_max_threads (void)
```

omp_get_thread_num()

Essa função retorna um número inteiro que representa o identificador da *thread* dentro do grupo de *threads* que está executando a região paralela. Esse número estará compreendido entre zero e `omp_get_num_threads() -1`, sendo que a *thread master* sempre terá identificador zero.

SINTAXE:
```
int omp_get_thread_num (void)
```

omp_get_num_procs()

A função `omp_get_num_procs()` retorna o número de processos disponíveis para o programa no momento da chamada da função.

SINTAXE:
```
int omp_get_num_procs (void)
```

omp_in_parallel()

A função `omp_in_parallel()` retorna um valor diferente de zero se for chamada dentro de uma região definida como paralela. Caso contrário seu retorno será zero.

SINTAXE:
```
int omp_in_parallel (void)
```

omp_set_dynamic()

A função `omp_set_dynamic()` habilita ou desabilita o ajuste dinâmico do número de *threads* disponíveis para a execução da região paralela.

SINTAXE:
```
void omp_set_dynamic (int dynamic_threads);
```

Se o valor do parâmetro `dynamic_threads` for diferente de zero, o número de *threads* que será usado será ajustado automaticamente pelo ambiente, visando à otimização dos recursos disponibilizados pelo sistema. Como conseqüência, a aplicação utilizará o número de *threads* disponíveis, que será um número entre 1 e o número informado pela função `omp_get_num_threads()`. Quando o valor do parâmetro `dynamic_threads` for zero o ajuste dinâmico será desabilitado.

A função `omp_set_dynamic()`tem precedência sobre a variável de ambiente `OMP_DYNAMIC` e só terá o efeito esperado se for chamada em um trecho de código seqüencial ou dentro de uma região paralela inativa, ou seja, quando o retorno da chamada da função `omp_in_parallel()` for zero. O valor padrão para o ajuste dinâmico é definido pela implementação, dessa forma, se alguma aplicação depender de um número fixo de *threads* para a execução correta do código, o ajuste dinâmico do número de *threads* deve ser desabilitado de forma explícita.

omp_get_dynamic()

A função `omp_get_dynamic()` retorna um valor diferente de zero se o ajuste dinâmico estiver habilitado e retornará zero se estiver desabilitado. Se o valor do parâmetro `dynamic_threads` for diferente de zero, o número de *threads* que será usado será ajustado automaticamente pelo ambiente, visando à otimização dos recursos disponíveis.

SINTAXE:
```
int omp_get_dynamic (void);
```

omp_set_nested()

A função `omp_set_nested()` habilita ou desabilita o paralelismo aninhado.

SINTAXE:
```
void omp_set_nested (int nested);
```

O valor padrão do parâmetro *nested* é zero, que corresponde ao paralelismo aninhado estar desabilitado. Nesse caso, a região paralela aninhada será executada serialmente pela *thread* corrente. No caso em que o parâmetro *nested* é diferente de zero, o paralelismo aninhado será habilitado e será possível a criação de um grupo com mais de uma *thread*. Assim como a função descrita anteriormente, ela somente terá efeito se o retorno da função `omp_in_parallel` for zero. A chamada da função `omp_set_nested()` tem precedência sobre a variável de ambiente `OMP_NESTED`.

omp_get_nested()

A função `omp_get_nested()` retorna um valor diferente de zero se o paralelismo aninhado estiver habilitado e zero em caso contrário.

SINTAXE:
```
void omp_get_nested (int nested);
```

3.2 - Funções de Bloqueio

As bibliotecas do OpenMP incluem um conjunto de rotinas de bloqueio que podem ser utilizadas para promover a sincronização entre as *threads*. Essas rotinas operam sobre as variáveis de travamento do OpenMP, que são denominadas de *"lock"* e podem ser do tipo `omp_lock_t` ou `omp_nest_lock_t`.

Uma *"lock"* pode estar em um dos seguintes estados: não-inicializada, desbloqueada ou bloqueada. Se uma *"lock"* está desbloqueada, então uma *thread* deve bloquear essa *"lock"*. A *thread* que bloqueia uma *"lock"* é dita como proprietária da mesma e deve desbloqueá-la. Uma *thread* não pode bloquear ou desbloquear uma *"lock"* que pertence a outra *thread*.

Existem dois tipos de *"lock"*: *"locks"* simples e *"locks"* aninhadas. Uma *"lock"* aninhada deve ser bloqueada várias vezes antes de ser desbloqueada.

As rotinas de bloqueio do OpenMP acessam as *"locks"* de tal forma que elas sempre lêem e atualizam o valor mais atualizado da *"lock"*. Dessa forma, não é necessário incluir uma diretiva `flush` para assegurar que o valor da *"lock"* está consistente entre as *threads*.

`omp_init_lock()` e `omp_init_nest_lock()`

Essas rotinas são utilizadas para inicializar uma *"lock"*. Além disso, o contador *nesting* de uma *"lock"* aninhadas é inicializado com o valor zero.

SINTAXE:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

`omp_destroy_lock()` e `omp_destroy_nest_lock()`

Essas rotinas alteram o estado da *"lock"* para o estado não-inicializado.

SINTAXE:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nested_lock(omp_nested_lock_t *lock);
```

`omp_set_lock()` e `omp_set_nest_lock()`

Essas rotinas bloqueiam a *thread* até que a *"lock"* especificada esteja disponível e, então, bloqueia essa *"lock"*.

Uma *"lock"* simples está disponível se a mesma estiver desbloqueada. Uma *"lock"* aninhada está disponível se a mesma estiver desbloqueada ou a *thread* já possuir essa *"lock"*.

SINTAXE:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nested_lock(omp_nested_lock_t *lock);
```

`omp_unset_lock()` e `omp_unset_nest_lock()`

Essas rotinas desbloqueiam uma *"lock"* simples e decrementa o contador *nesting* de uma *"lock"* aninhada. Quando o contador *nesting* é igual a zero, então a *"lock"* aninhada é desbloqueada. Se uma *"lock"* é desbloqueada e houver uma ou mais *threads* esperando por essa *"lock"*, então uma dessas *thread* é escolhida para ser a proprietária da *"lock"*.

SINTAXE:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nested_lock(omp_nested_lock_t *lock);
```

`omp_test_lock()` e `omp_test_nest_lock()`

Essas rotinas bloqueiam uma *"lock"* da mesma maneira que as rotinas `omp_set_lock()` e `omp_set_nest_lock()`, exceto pelo fato de que elas não bloqueiam a *thread* que está executando a rotina.

Para uma *"lock"* simples, a rotina `omp_test_lock()` retorna verdadeiro se a *"lock"* foi bloqueada com sucesso; caso contrário ela retorna

falso. Para uma *"lock"* aninhada, a rotina `omp_test_nest_lock()` retorna o novo contador *nesting* se a *"lock"* foi bloqueada com sucesso; caso contrario, ela retorna zero.

SINTAXE:
```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nested_lock(omp_nested_lock_t *lock);
```

3.3 – Funções de tempo

omp_get_wtime()

A função `omp_get_wtime()` retorna um valor em precisão dupla igual ao tempo decorrido em segundos desde "algum tempo no passado". Esse "tempo no passado é arbitrário, mas é garantido que o mesmo não muda durante a execução do programa.

SINTAXE:
```
double omp_get_wtime(void);
```

No fragmento de código abaixo está ilustrado como a função `omp_get_wtime()` pode ser facilmente utilizada.

```
#include <omp.h>
double start;
double end;
start = omp_get_wtime(void);
... work to be timed ...
end = omp_get_wtime(void);
printf("Work took %f sec. time \n", end-start);
```

omp_get_wtick()

A função `omp_get_wtick()` retorna o valor do número em segundos entre sucessivos clocks, ou seja, o tempo decorrido em chamadas sucessivas

A seguir é ilustrada a sintaxe dessa função.

SINTAXE:
```
double omp_get_wtick(void);
```

4 – VARIÁVEIS DE AMBIENTE

O OpenMP possui algumas variáveis de ambiente cuja função é controlar as variáveis de controle interno que influenciam na execução de aplicações paralelas. Se as variáveis de ambiente forem modificadas ao longo

da execução do programa, essas mudanças são ignoradas pelo OpenMP. Por outro lado, as variáveis de controle interno podem ter seus valores modificados durante a execução do programa por meio da utilização de uma cláusula ou da chamada de uma função do OpenMP.

As variáveis de ambiente são: `OMP_SCHEDULE`, `OMP_NUM_THREADS`, `OMP_DYNAMIC` e `OMP_NESTED`.

`OMP_SCHEDULE`

Essa variável especifica qual será o tipo de `schedule` e o tamanho do bloco que será utilizado nos construtores `for` e `parallel for` que possuírem a cláusula `schedule runtime`. Para as diretivas `for` e `parallel for` que possuam outro tipo `schedule`, o `OMP`_SCHEDULE será simplesmente ignorado.

O valor atribuído a essa variável de ambiente deve ser da forma:

```
tipo [chunk_size]
```

Onde,

- `tipo: static, dynamic ou guided`
- *`chunk_size`*: valor opcional e positivo. Caso o mesmo não seja atribuído, o valor assumido será 1, exceto no caso do `static`, em que o *`chunk_size`* é definido como sendo o número de iterações do laço dividido pelo número de *threads*.

SINTAXE:

CSH:

```
setenv OMP_SCHEDULE "static"
setenv OMP_SCHEDULE "dynamic,6"
```

BASH

```
export OMP_SCHEDULE = "static"
export OMP_SCHEDULE = "dynamic,6"
```

`OMP_NUM_THREADS`

A variável `OMP_NUM_THREADS` define o número de *threads* que será utilizado durante a execução do código. Entretanto, esse valor pode ser alterado explicitamente através da chamada da função `omp_set_num_threads()` ou pela cláusula `num_threads` na diretiva `parallel`.

O valor dessa variável de ambiente deve sempre um inteiro positivo. Se nenhum valor for especificado para a variável `OMP_NUM_THREADS`, se o valor especificado não for um inteiro positivo ou se o valor for maior que o número de *threads* do que o sistema pode suportar, o número de *threads* será então definido pela implementação.

SINTAXE:

CSH:
```
setenv OMP_NUM_THREADS 16
```
BASH:
```
export OMP_NUM_THREADS = 16
```

OMP_DYNAMIC

A variável `OMP_DYNAMIC` habilita ou desabilita o ajuste dinâmico do número de *threads* disponíveis para execução da região paralela. Entretanto, o esse ajuste pode ser habilitado ou desabilitado de forma explicita pela chamada da função `omp_set_dynamic()` durante a execução do programa.

O valor dessa variável de ambiente deve ser `TRUE` ou `FALSE`. Se for `TRUE`, o número de *threads* que será usado para executar as regiões paralelas será ajustado pela implementação do OpenMP para melhor otimizar a utilização dos recursos do sistema. Quando o valor da variável for `FALSE`, o ajuste dinâmico é desabilitado.

SINTAXE:

CSH:
```
setenv OMP_DYNAMIC TRUE
```
BASH:
```
export OMP_DYNAMIC = TRUE
```

OMP_NESTED

A variável de ambiente `OMP_NESTED` habilita ou desabilita o paralelismo aninhado, a menos que o mesmo seja habilitado ou desabilitado pela chamada da função `omp_set_nested()`. Se a variável de ambiente for definida como `TRUE`, o paralelismo aninhado será habilitado. Se for definida como `FALSE`, será desabilitado. Vale ressaltar que o valor padrão dessa variável é `FALSE`.

SINTAXE:

CSH:
```
setenv OMP_NESTED TRUE
```
BASH
```
export OMP_NESTED = TRUE
```

REFERÊNCIAS BIBLIOGRÁFICAS

DANTAS, M. *Computação Distribuída de Alto Desempenho – Redes, Clusters e Grids Computacionais***.** Rio de Janeiro: Axcel Books, 2005.

CHAPMAM, B. , JOST G. , VAN DER PAS, R. *Using OpenMP.* Massachusetts: Massachusetts Institute of Technology, 2008.

TORELLI, J. C., BRUNO, O. M., *Programação paralela em SMP's com OpenMP e Posix Threads: um estudo comparativo* in IV Congresso Brasileiro de Computação – CBComp, 2004.

CHANDRA, R., DAGUM, L., MENON, R. *Parallel Programming in OpenMP*. San Francisco: Morgan Kaufmann Publishers, 2001.

CARDOSO, B., ROSA, S. R. A. dos S., FERNANDES, T.M., *Multicore***.**

Tutorial do Lawrence Livermore National Laboratory: https://computing.llnl.gov

Website do OpenMP: www.openmp.org

Website do Top500: www.top.org