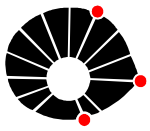


Universidade Estadual de Campinas

Centro Nacional de Processamento de Alto Desempenho
São Paulo

Apostila de Treinamento:

Introdução ao OpenMP

Revisão: 2010

Índice

1. Conceitos

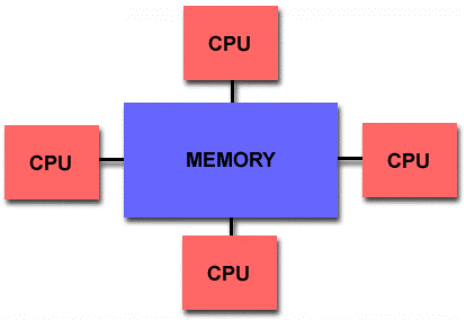
O uso de paralelismo em computadores, e entre computadores, é agora a regra e não mais a exceção. Internamente em um processador, diversas unidades funcionais executam tarefas em paralelo que estão escondidas no conjunto de instruções da arquitetura do processador, transparente para o usuário.

Em algumas aplicações específicas ou em determinados programas, os compiladores podem automaticamente detectar e explorar o paralelismo entre vários processadores de uma maneira eficiente (Ex.: “loops”). No entanto, na maioria dos casos, os programadores necessitam instruir aos compiladores, quando e onde utilizar ações paralelas. Não é uma tarefa trivial, pois inclui técnicas de replicação de dados, movimentação de dados, balanceamento entre a carga de execução e comunicação.

Para simplificar a tarefa dos programadores, alguns modelos de sistemas de programação paralela foram desenvolvidos, cada um com o seu próprio modelo de programação. Os mais importantes são:

“Shared Memory”;
“Distributed Memory”/”Message Passing”;
“Data-parallel programming”;
“Threads”/”MultiThreads”.

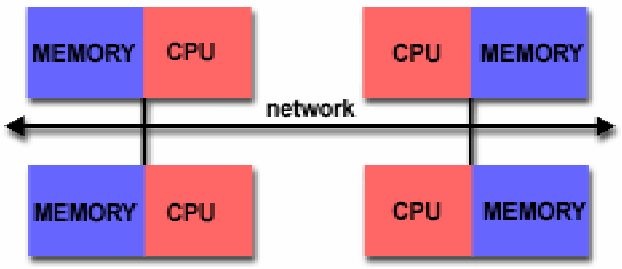
1 – “Shared Memory”



Ambiente com vários processadores que compartilham o espaço de endereçamento de uma única memória. Os processadores podem operar independentemente, mas compartilham os recursos dá mesma memória; mudanças num endereço da memória por um processador, é visível por todos os outros processadores, tornando a programação mais simples.

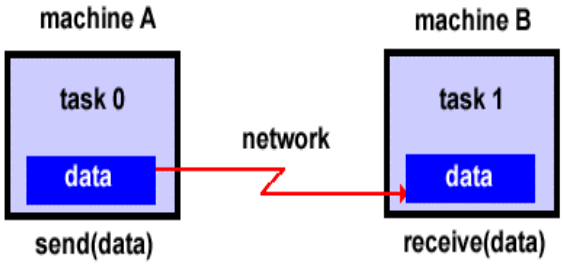
Os processos compartilham um espaço de endereçamento comum, no qual o acesso é feito no modo assíncrono; não há necessidade de especificar explicitamente a comunicação entre os processos. A implementação desse modelo pode ser feita pelos compiladores nativos do ambiente.

2 – “Distributed Memory”



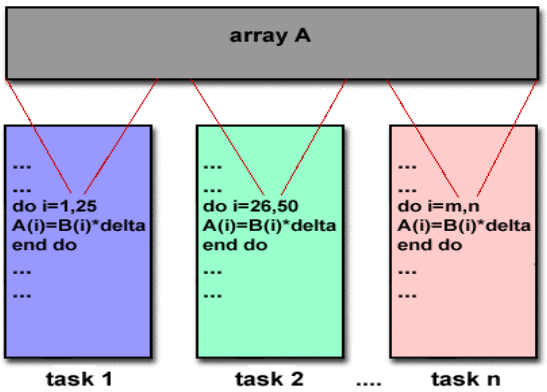
Ambiente com vários processadores, cada um com sua própria memória e interconectados por uma rede de comunicação. Programação complexa. O programador é responsável pela comunicação entre os processadores.

As tarefas compartilham dados através de comunicação de envio e recebimento de mensagens (“message-passing”); múltiplas tarefas iniciadas e distribuídas pelos processadores do ambiente, utilizando o seu próprio endereçamento de memória;



O programador é responsável por determinar todo o paralelismo e a comunicação entre as tarefas; Implementações: PVM, Linda, MPI.

3 – “Data Parallel Programming”



Modelo de programação que utiliza memória compartilhada, permitindo a todas as tarefas acesso a estrutura de dados na memória global; O trabalho paralelo é efetuado em um conjunto de dados, e os dados devem estar organizados na forma de conjuntos (“loops”), aonde cada tarefa irá trabalhar com partições diferentes dessa estrutura de dados e efetuar a mesma operação em sua partição da estrutura de dados; Implementações: Fortran90/95, HPF - High Performance Fortran, MPI e OpenMP.

4 – “Threads”/”Multithreads”

A estrutura responsável pela manutenção de todas as informações necessárias à execução de um programa, como conteúdo de registradores e espaço de memória, chama-se processo. O conceito de processo pode ser definido como sendo o ambiente onde se executa um programa. O processo pode ser dividido em três elementos básicos que juntos mantêm todas as informações necessárias à execução do programa:

O **Contexto de Hardware** constitui-se, basicamente, no conteúdo de registradores: “program counter” (PC), “stack pointer” (SP) e “bits” de estado. Durante a execução de um processo, o contexto de hardware estará armazenado nos registradores do processador. No momento em que o processo perde a utilização da CPU, o sistema salva suas informações.

O **Contexto de Software** especifica as características do processo que vão influir na execução de um programa, como, por exemplo, o número máximo de arquivos abertos simultaneamente ou o tamanho do buffer para operações de E/S. Essas características são determinadas no momento da criação do processo, mas algumas podem ser alteradas durante sua existência.

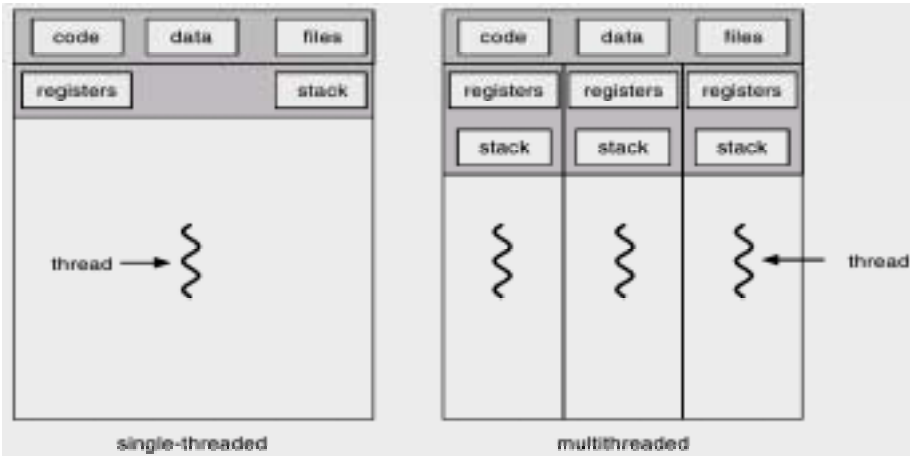
O **Espaço de Endereçamento** é a área de memória do processo aonde o programa será executado, além do espaço para os dados utilizados por ele. Cada processo possui seu próprio espaço de endereçamento, que deve ser protegido do acesso dos demais processos.

Um programa é uma seqüência de instruções, composto por desvios, repetições (iterações) e chamadas de procedimentos e/ou funções. Em um ambiente de programação “monothread”, um processo suporta apenas um programa no seu espaço de endereçamento e apenas uma instrução do programa é executada por vez. Caso seja necessário criar aplicações concorrentes e paralelas, são implementados múltiplos processos independentes e/ou subprocessos. A utilização desses subprocessos independentes permite dividir uma aplicação em partes que podem trabalhar de forma concorrente.

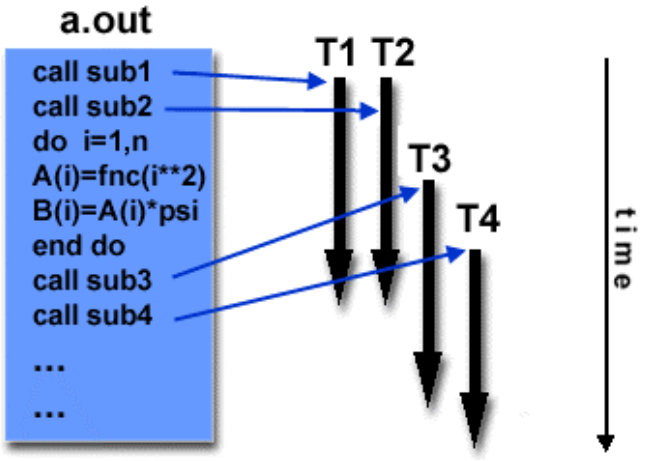
O uso de subprocessos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado, o sistema deve alocar recursos (contexto de hardware, contexto de software e espaço de endereçamento) para cada processo, além de consumir tempo de CPU. No caso do término do processo, o sistema dispensa tempo para liberar os recursos previamente solicitados.

Como cada processo possui seu próprio espaço de endereçamento, a comunicação entre os subprocessos torna-se difícil e lenta, pois utiliza mecanismos tradicionais como “pipes”, sinais, semáforos, memória compartilhada ou troca de mensagem. Além disto, o compartilhamento de recursos comuns aos subprocessos concorrentes, como memória e arquivos abertos, não é simples.

Na tentativa de diminuir o tempo gasto na criação e eliminação de subprocessos, bem como economizar recursos do sistema como um todo, foi introduzido o conceito de “thread”. No ambiente “multithread”, cada processo pode responder a várias solicitações concorrentemente ou mesmo simultaneamente, se houver mais de um processador. Em um ambiente “multithread”, não existe a idéia de um programa, mas de “threads” (linhas). O processo, neste ambiente, tem pelo menos um “thread” de execução, podendo compartilhar o seu espaço de endereçamento com inúmeros “threads”, que podem ser executados de forma concorrente e/ou simultânea, no caso de múltiplos processadores. Cada “thread” possui seu próprio conjunto de registradores (contexto de hardware), porém, compartilha o mesmo espaço de endereçamento, temporizadores e arquivos, de forma natural e eficiente com as demais “threads” do processo.



A grande diferença entre subprocessos e “threads” é em relação ao espaço de endereçamento. Enquanto subprocessos possuem, cada um, espaços independentes e protegidos, “threads” compartilham o mesmo espaço de endereçamento do processo, sem nenhuma proteção, permitindo que um “thread” possa alterar dados de outro “thread”. Apesar dessa possibilidade, “threads” são desenvolvidas para trabalhar de forma cooperativa, efetuando uma tarefa em conjunto.



2. Introdução ao OpenMP

1 – O que é OpenMP?

OpenMP é uma interface de programação (API), portável, baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores. É composto por três componentes básicos:

- Diretivas de Compilação;
- Biblioteca de Execução;
- Variáveis de Ambiente.

OpenMP está disponível para uso com os compiladores C/C++ e Fortran, podendo ser executado em ambientes Unix e Windows (Sistemas Multithreads). Definido e mantido por um grupo composto na maioria por empresas de hardware e software, denominado como OpenMP ARB (Architecture Review Board). Hardware: Compaq (Digital), Hewlett-Packard Company, Intel Corporation, International Business Machines (IBM) , Kuck & Associates, Inc. (KAI) , Silicon Graphics, Inc. , Sun Microsystems, Inc., U.S. Department of Energy ASCI program . Software: Absoft Corporation , Edinburgh Portable Compilers , GENIAS Software GmBH , Myrias Computer Technologies, Inc. , The Portland Group, Inc. (PGI) , ADINA R&D, Inc. , ANSYS, Inc. , Dash Associates , Fluent, Inc. , ILOG CPLEX Division , Livermore Software Technology Corporation (LSTC) , MECALOG SARL , Oxford Molecular Group PLC , The Numerical Algorithms Group Ltd. (NAG).

2 – Histórico

No início da década de 90, fabricantes de máquinas de memória compartilhada e similar, desenvolveram extensões do compilador Fortran com um conjunto especial de instruções denominadas de diretivas de execução que permitiam:

- Usuário, programador Fortran (programação serial), adicionar instruções para identificar “loops” que poderiam ser paralelizados;
- O compilador passa a ser responsável pela paralelização automática desses “loop” por entre os processadores do ambiente SMP.

O primeiro padrão, ANSI X3H5, para testes foi lançado em 1994, mas nunca foi adotado devido ao grande uso e interesse, na época, por máquinas de memória distribuída (clusters).

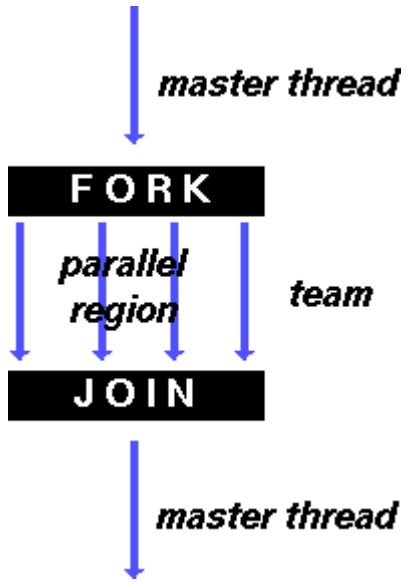
Um novo padrão foi desenvolvido em 1997 a partir do padrão anterior, quando as arquiteturas de memória compartilhada se tornaram mais comum.

Em 28 de Outubro de 1997 foi divulgado e disponibilizado o OpenMP Fortran API e no final de 1998 foi disponibilizado o OpenMP C/C++ API.

Informações do OpenMP em http://www.openmp.org/

3. Modelo de Programação Paralela

- **Paralelismo baseado em "Threads":** Processo de memória compartilhada que consiste de múltiplos "threads";
- **Paralelismo explícito:** O OpenMP é um recurso de programação paralela que permite ao programador total controle sobre a paralelização do código;
- **Modelo "Fork" – "Join" :**



1. Todos os programas OpenMP começam como um processo simples: **"Thread" Mestre**. O "thread" mestre executa sequencialmente até a primeira definição de uma região paralela ser encontrada.
2. **"FORK"** (bifurcar): O thread mestre cria um conjunto (**"team")** de threads paralelos.
3. Os comandos de um programa que estão inseridos na região paralela serão executados em paralelo pelos diversos "threads" criados.
4. **"JOIN"** (unir): Quando o conjunto de "threads" completar a execução dos comandos na região paralela, os "threads" são sincronizados e finalizados, permanecendo apenas o "thread" mestre.

- **Baseado em diretivas de compilação:** Todo paralelismo do OpenMP é especificado através de diretivas de compilação.

- **Suporta paralelismo recursivo:** Permite que em uma região paralela existam outras regiões que podem ser executadas em paralelo, e assim sucessivamente. Depende da implementação OpenMP.

- **"Threads" dinâmico**: É possível durante a execução do programa alterar o número de "threads" que serão criados. Depende da implementação OpenMP

Exemplo de estrutura OpenMP

Fortran

```
PROGRAM HELLO
      INTEGER VAR1, VAR2, VAR3
***       Código serial
          .
          .
          .
***       Início da seção paralela. “Fork” um grupo de “threads”.

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)

 ***      Seção paralela executada por todas as “threads”
          .
          .
          .
 ***      Todas as “threads” efetuam um “join” a thread mestre e finalizam

!$OMP END PARALLEL

***       Código serial
          .
          .
          .
      END
```

C / C++

```
#include <omp.h>
main ()  {
int var1, var2, var3;
***      Código serial
     .
     .
     .
***       Início da seção paralela. “Fork” um grupo de “threads”.

#pragma omp parallel private(var1, var2) shared(var3)
  {
***      Seção paralela executada por todas as “threads”
          .
          .
          .
 ***      Todas as “threads” efetuam um “join” a thread mestre e finalizam
  }
***       Código serial
     .
     .
     .
}
```

4-Diretivas OpenMP

4.1-Formato FORTRAN

Identificador (“sentinel”)	Diretiva	[atributos,…]
Todas as diretivas OpenMP do Fortran devem começar com um identificador (“sentinel”). Os identificadores aceitos dependem do código fonte Fortran: **!$OMP** **C$OMP** ***$OMP**	Diretiva OpenMP válida	Campo opcional. Os atributos podem aparecer em qualquer ordem.

- Exemplo:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
```

- Formato fixo (Fortran77):

  !$OMP C$OMP *$OMP são aceitos e devem começar na coluna 1;

- Formato livre (Fortran90,95):

  !$OMP é o único identificador aceito. Pode aparecer em qualquer coluna, mas deve ser precedido por um espaço em branco;

- Os compiladores Fortran, normalmente necessitam de uma opção de compilação que indica ao compilador para ativar e interpretar todas as diretivas OpenMP;

- Diversas diretivas do Fortran OpenMP identificam seções do código que serão executadas em paralelo. Essas diretivas trabalham em pares, iniciando e finalizando a seção.

!$OMP diretiva

[código Fortran]

!$OMP end diretiva

4.2-Formato C/C++

#pragma omp	Diretiva	[atributos,…]	Nova linha
Obrigatório para todas as diretivas OpenMP C/C++	Diretiva OpenMP válida.	Campo opcional. Os atributos podem aparecer em qualquer ordem.	Obrigatório. Precede uma estrutura de bloco.

- Exemplo:

```
#pragma omp parallel default(shared) private(beta,pi)
```

- Segue as regras de sintaxe do C/C++ ;

- “Case sensitive” ;

- Linhas de diretivas muito longas podem continuar em diversas linhas, acrescentando o caractere de nova linha (“\”) no final da diretiva.

4.3 – Construção PARALLEL

Identifica um bloco de código que será executa por múltiplos “threads”. É a construção fundamental do OpenMP.

- Formato:

Fortran	**!$OMP PARALLEL** ***[atributo ...]*** **IF** ***(expressão lógica)*** **PRIVATE** ***(lista)*** **SHARED** ***(lista)*** **DEFAULT (PRIVATE \| SHARED \| NONE)** **FIRSTPRIVATE** ***(lista)*** **REDUCTION** ***(operador: lista)*** **COPYIN** ***(lista)*** ***Bloco de código*** **!$OMP END PARALLEL**
C/C++	**#pragma omp parallel** ***[atributo ...] nova_linha*** **if** ***(expressão lógica)*** **private** ***(lista)*** **shared** ***(lista)*** **default (shared | none)** **firstprivate** ***(lista)*** **reduction** ***(operador: lista)*** **copyin** ***(lista)*** ***estrutura de bloco***

- **Atributos:**

IF

- Especifica uma condição para que o grupo de “threads” seja criado, se o resultado for verdadeiro. Se falso, a região definida como paralela, será executada somente pela “thread” mestre.

- **Observações**

1-“Fork/Join”

Quando uma “thread” chega a uma definição de região paralela, ela cria um conjunto de “threads” e passa a ser a “thread” mestre. A “thread” mestre faz parte do conjunto de “threads” e possui o número de identificação “0”.

A partir do início da região paralela, o código é duplicado e todas as “threads” executarão esse código.

Existe um ponto de sincronização (“barreira”) no final da região paralela, sincronizando o fim de execução de cada “thread”. Somente a “thread” mestre continua desse ponto.

2-O número de “trheads”

Em uma execução com o OpenMP, o número de “threads” é determinado pelos seguintes fatores, em ordem de precedência:

1. Utilização da função **omp_set_num_threads()** no código Fortran ou C/C++;
2. Definindo a variável de ambiente **OMP_NUM_THREADS,** antes da execução;
3. Implementação padrão do ambiente: número de processadores em um nó.

3-“Threads” dinâmico

Um programa com várias regiões paralelas, utilizara o mesmo número de “thread” para executar cada região. Isso pode ser alterado permitindo que durante a execução do programa, o número de “threads” seja ajustado dinamicamente para uma determinada região paralela. Dois métodos disponíveis:

1. Utilização da função **omp_set_dynamic()** no código Fortran ou C/C++;
2. Definindo a variável de ambiente **OMP_DYNAMIC,** antes da execução.

4-Regiões paralelas recursivas

Uma região paralela dentro de outra região paralela resulta na criação de um novo grupo de “threads” de apenas uma “thread”, por “default”. É possível definir um número maior de “threads”, utilizando um dos dois métodos disponíveis:

1. Utilização da função **omp_set_nested**()no código Fortran ou C/C++;
2. Definindo a variável de ambiente **OMP_NESTED,** antes da execução.

5-Restrições

Não é permitido caminhar para dentro ou fora (”branch”) de uma estrutura de blocos definida por uma diretiva OpenMP e somente um IF é permitido.

Exemplo:

- Programa “Hello World”;
- Todas as “threads” executarão o código da região paralela definida pela diretiva OpenMP;
- São utilizadas rotinas da biblioteca OpenMP para obter a identificação de cada “thread” e o número “threads” que participam da execução.

Fortran

```
      PROGRAM HELLO

      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
     +   OMP_GET_THREAD_NUM

C     Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL PRIVATE(NTHREADS, TID)

C     Obtain and print thread id
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

C     Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF

C     All threads join master thread and disband
!$OMP END PARALLEL
      END
```

C / C++

```
#include <omp.h>
main ()  {
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {
  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

Atributos Comuns

- Em programação OpenMP é importante entender como é utilizado o dado em uma “thread” (“Data scope”).

- Devido ao fato do OpenMP ser baseado no modelo de programação de memória compartilhada, a maioria das variáveis são compartilhadas entre as “threads” por definição (Variáveis Globais).

- Variáveis globais, são as variáveis definidas:

  Fortran: COMMON blocks, Atributo SAVE , variáveis MODULE
  C: Variáveis estáticas

- Variáveis não compartilhadas, locais, privadas:

  Variáveis de índice (“loops”)
  Variáveis locais de subrotinas

- Os atributos comuns são utilizados por diversas diretivas (PARALLEL, DO/for, e SECTIONS) para controlar o acesso as dados.

  Definem como os dados das variáveis serão transferidos das seções seriais para as seções paralelas.

  Definem quais as variáveis serão visíveis por todas as “threads” e quais serão locais.

- **Atributos:**

PRIVATE

Declara que as variáveis listadas serão de uso específico de cada “thread”. Essas variáveis não são iniciadas.

SHARED (“default”)

Declara que as variáveis listadas irão compartilhar o seu conteúdo com todas as “threads” de um grupo. As variáveis existem em apenas um endereço de memória, que pode ser lido e escrito por todas as “threads” do grupo.

DEFAULT

Permite que o programador defina o atributo “default” para as variáveis em uma região paralela (PRIVATE, SHARED ou NONE).

FIRSTPRIVATE

Define uma lista de variáveis com o atributo PRIVATE, mas sendo inicializadas automaticamente, de acordo com o valor que possuíam no programa (“thread” mestre) antes de uma região paralela.

LASTPRIVATE

Define uma lista de variáveis com o atributo PRIVATE e copia o valor da última iteração de um “loop” da última “thread” que finalizou.

REDUCTION

Efetua uma operação de redução em uma lista de variáveis. Essas variáveis devem ser escalares e terem sido definidas com o atributo SHARED, no contexto em que elas participarão.

Exemplo: REDUCTION – Produto de vetores

- As iterações do "loop" paralelo serão distribuídas em tamanhos iguais para cada "thread" do grupo (SCHEDULE STATIC).

- No final da construção paralela do "loop", todas as "threads" irão adicionar o seu valor do resultado para "thread" mestre com a variável global.

Fortran

```
      PROGRAM DOT_PRODUCT
      INTEGER N, CHUNK, I
      PARAMETER (N=100)
      PARAMETER (CHUNK=10)
      REAL A(N), B(N), RESULT
!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = I * 2.0
      ENDDO
      RESULT= 0.0
!$OMP  PARALLEL DO
!$OMP& DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)
!$OMP& REDUCTION(+:RESULT)
      DO I = 1, N
        RESULT = RESULT + (A(I) * B(I))
      ENDDO
!$OMP  END DO NOWAIT
      PRINT *, 'Final Result= ', RESULT
      END
```

C / C++

```
#include <omp.h>
main ()  {
int   I, n, chunk;
float a[100], b[100], result;
/* Some initializations */
n = 100;
chunk = 10;
result = 0.0;
for (I=0; I < n; I++)
  {
  a[I] = I * 1.0;
  b[I] = I * 2.0;
  }
#pragma omp parallel for      \
  default(shared) private(i)  \
  schedule(static,chunk)      \
  reduction(+:result)
  for (I=0; I < n; I++)
    result = result + (a[I] * b[I]);
printf("Final result= %f\n",result);

}
```

- Observações e restrições do atributo REDUCTION:

    1. Variáveis da lista devem ser do tipo escalar. Não são possíveis variáveis do tipo “array” (vetores);

    2. Uma cópia privada de cada variável da lista é criada e iniciada, dependendo da operação de redução;

    3. Devem ser declaradas como SHARED;

    4. As cópias são atualizadas localmente pelas “threads”;

    5. No final da construção, as cópias locais são combinadas através do operador em único valor, na variável compartilhada da “thread” mestre.

    6. A variável utilizada na operação de redução deve ser utilizada dentro de uma região paralela e em expressões da seguinte forma:

Fortran	**C / C++**
x = x operator expr	***x = x op expr***
x = expr operator x *(exceto subtração)*	***x = expr op x*** *(exceto subtração)*
x = intrinsic(x, expr)	***x binop = expr***
x = intrinsic(expr, x)	***x++***
	++x
	x—
	--x
x é uma variável escalar da lista ***expression*** é uma expressão escalar que não faz referência a x ***intrinsic*** pode ser: MAX, MIN, IAND, IOR, IEOR ***operator*** pode ser: +, *, -, .AND., .OR., .EQV., .NEQV.	***x*** é uma variável escalar da lista ***expression*** é uma expressão escalar que não faz referência a x ***op*** pode ser: +, *, -, /, &, ^, |, &&, || ***binop*** pode ser: +, *, -, /, &, ^, |

Exercício 1

O objetivo desse exercício é entender como se compila e executa um programa utilizando OpenMP. O programa HELLO, é o mais simples possível; em paralelo várias “threads” irão imprimir um “hello”.

1 – Caminhe para a pasta ~/curso/openmp/ex1

```
cd  ~/curso/openmp/ex1
```

2 – Se quiser, verifique o conteúdo do programa omp_hello.f ou omp_hello.c

```
more  omp_hello.f
```

3 – Compilação

Basicamente, as implementações de Fortran e C/C++ que utilizam OpenMP são adaptações a compiladores que já existem e dependem da utilização de opções de compilação.

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_hello.f  -qsmp=omp  -o  hello

xlc_r  omp_hello.c  -qsmp=omp  -o  hello
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_hello.f  -openmp  -o hello

icc omp_hello.c  -openmp  -o  hello
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. No caso desse exercício, a maneira mais fácil será definir a variável de ambiente OMP_NUM_THREADS.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

Exercício 2

O objetivo desse exercício é entender como se compila e executa um programa utilizando OpenMP. O programa omp_prod_vet, simula o produto entre vetores, utilizando o atributo REDUCTION da diretiva DO/for do OpenMP .

1 – Caminhe para a pasta ~/curso/openmp/ex2

```
cd  ~/curso/openmp/ex2
```

2 – Se quiser, verifique o conteúdo do programa omp_prod_vet.f ou omp_prod_vet.c

```
more  omp_prod_vet.f
```

3 – Compilação

Basicamente, as implementações de Fortran e C/C++ que utilizam OpenMP são adaptações a compiladores que já existem e dependem da utilização de opções de compilação.

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_prod_vet.f  -qsmp=omp  -o  prod

xlc_r  omp_prod_vet.c  -qsmp=omp  -o  prod
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_prod_vet.f  -openmp  -o prod

icc omp_prod_vet.c  -openmp  -o  prod
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. No caso desse exercício, a maneira mais fácil será definir a variável de ambiente OMP_NUM_THREADS.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

Exercício 3

O objetivo desse exercício é entender como se compila e executa um programa utilizando OpenMP. O programa omp_mm, simula a operação de multiplicação de matrizes, utilizando OpenMP para paralelizar a operação.

1 – Caminhe para a pasta ~/curso/openmp/ex3

```
cd  ~/curso/openmp/ex3
```

2 – Edite o programa omp_mm.f ou omp_mm.c e adicione a diretiva DO/for do OpenMP nas posições indicadas no programa.

```
vi  omp_mm.f    ou   vi  omp_mm.c
                     ou
pico omp_mm.f    ou   pico  omp_mm.c
```

3 – Compilação

Basicamente, as implementações de Fortran e C/C++ que utilizam OpenMP são adaptações a compiladores que já existem e dependem da utilização de opções de compilação.

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_mm.f  -qsmp=omp  -o  mm

xlc_r  omp_mm.c  -qsmp=omp  -o  mm
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_mm.f  -openmp  -o mm

icc omp_mm.c  -openmp  -o  mm
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. No caso desse exercício, a maneira mais fácil será definir a variável de ambiente OMP_NUM_THREADS.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

4.4–Construções de Trabalho Compartilhado

- Esse tipo de construção divide a execução de uma região paralela por entre os membros do grupo de “threads”;

- Esse tipo de construção deve estar dentro de uma região paralela definida pelo OpenMP, afim de executar em paralelo;

- Não existe sincronização (barreira) no início de uma construção compartilhada, mas existe no final.

- Três tipos de construções:

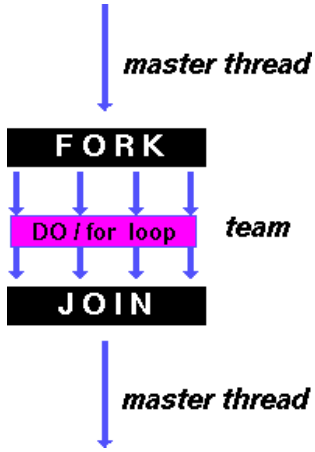
  DO / for – Compartilha as iterações de “loops” por entre o grupo de “threads”. Representa a implementação de paralelismo de dados.

  SECTIONS – Quebra o trabalho em seções separadas, aonde cada seção será executada por uma “thread” do grupo. Representa a implementação de paralelismo funcional, por código.

  SINGLE – Determina que uma seção da região paralela, seja executada por uma única “thread”.

Diretiva compartilhada DO/for

- A diretiva **DO / for** especifica que as iterações de um "loop" sejam distribuídas e executadas em paralelo pelo grupo de "threads". A região paralela tem que ter sido identificada antes.



- Formato:

Fortran

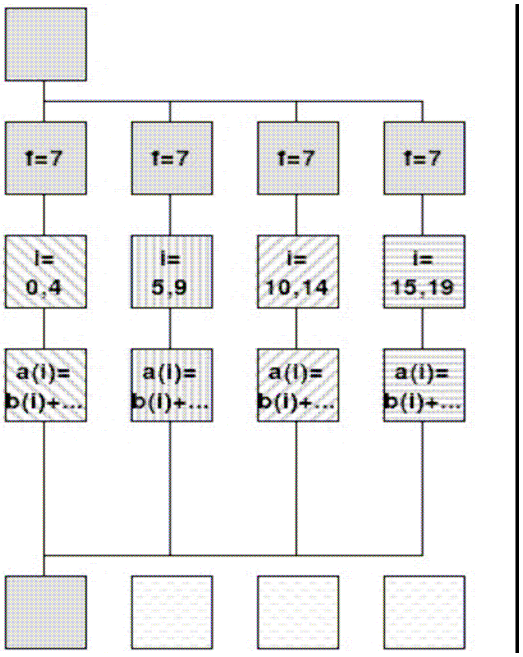
```
!$OMP DO [atributos ...]
         SCHEDULE (tipo [,chunk])
         ORDERED
         PRIVATE (lista)
         FIRSTPRIVATE (lista)
         LASTPRIVATE (lista)
         SHARED (lista)
         REDUCTION (operador | intrinsic : lista)

   do_loop

!$OMP END DO  [ NOWAIT ]
```

C/C++

```
#pragma omp for [atributo ...]  nova_linha
                schedule (tipo [,chunk])
                ordered
                private (lista)
                firstprivate (lista)
                lastprivate (lista)
                shared (lista)
                reduction (operador: lista)
                nowait

   for_loop
```

C / C++:

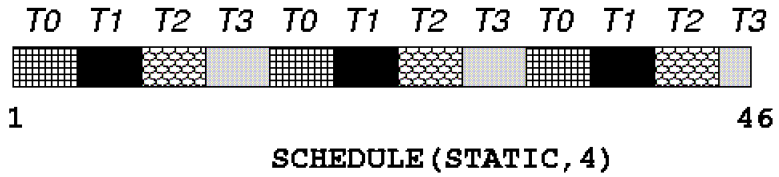
```
#pragma omp parallel private(f)
{
      f=7;

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



} /* omp end parallel */
```



- Atributos:

SCHEDULE Descreve como as iterações do "loop" serão divididas por entre os "threads":

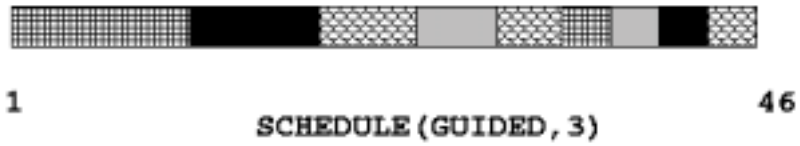
STATIC - O "loop" será dividido em pedaços de tamanho "chunk" e distribuído estaticamente para cada "thread" (por ordem de "trhead"); Se o parâmetro "chunk" não for definido, o compilador irá dividir o "loop" de maneira contínua e em partes iguais (quando possível), para cada "thread".



DYNAMIC - O "loop" será dividido em pedaços de tamanho "chunk" e distribuído dinamicamente para cada "thread" (pela "thread disponível); Quando uma "thread" finaliza um "chunk", outra e atribuída para processamento. O tamanho padrão de "chunk", neste caso, é 1.

GUIDED – O número de iterações para cada “thread” irá variar, começando com um valor grande e sendo reduzido exponencialmente, a medida que é enviado o tamanho do “loop” para cada “thread”. Neste caso, o valor de “chunk” é o tamanho mínimo permitido para o número de iterações, se não for especificado, será de 1.

1ªTh=46/4≈11 2ªTh=(46-11)/4≈8 3ªTh=(46-11-8)/4≈6 4ªTh=(46-11-8-6)≈5 ...



RUNTIME - A decisão do agendamento é feita durante a execução do programa, com a definição da variável de ambiente OMP_SCHEDULE. Neste caso o parâmetro “chunk” não é definido na diretiva.

export OMP_SCHEDULE = “GUIDED,4”

ORDERED – É utilizado quando existe num “loop” de uma região paralela, comandos que devem ser executados sequencialmente, na ordem do “loop”.

- **Observações**

1-Se o atributo **nowait** (C/C++) ou NO WAIT (Fortran) for especificado, as “threads” não serão sincronizadas no final do “loop” paralelo, continuando a execução para o próximo comando.

2-Não é possível desviar para for a de um “loop” associado a diretiva DO/for.

Exemplo:

- Programa Adição de Vetores;
- Os vetores A, B, C, e a variável N, serão compartilhados por entre todas as “threads”;
- A variável I será local para cada “Thread”;
- As iterações do “loop” serão distribuídas dinamicamente em tamanho CHUNK;
- As “threads” não serão sincronizadas ao completarem o trabalho (NOWAIT).

Fortran

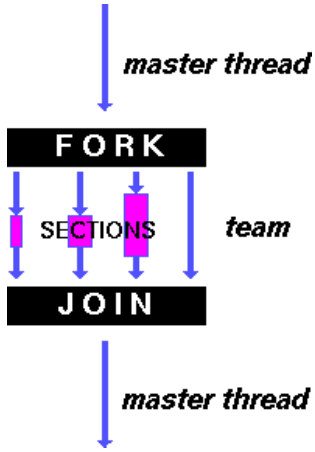
```
      PROGRAM VEC_ADD_DO
      INTEGER N, CHUNK, I
      PARAMETER (N=1000)
      PARAMETER (CHUNK=100)
      REAL A(N), B(N), C(N)
!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO
!$OMP PARALLEL SHARED(A,B,C,N) PRIVATE(I)

!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL
      END
```

C / C++

```
#include <omp.h>
#define CHUNK 100
#define N     1000
main ()
{
int i, n, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
n = N;
chunk = CHUNK;
#pragma omp parallel shared(a,b,c,n,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */

}
```

Diretiva Compartilhada SECTIONS

- Diretiva que divide o trabalho de forma não iterativa em seções separadas, aonde cada seção será executada por uma “thread” do grupo. Representa a implementação de paralelismo funcional, ou seja, por código.



- Formato:

Fortran

```
!$OMP SECTIONS [atributo ...]
               PRIVATE (lista)
               FIRSTPRIVATE (lista)
               LASTPRIVATE (lista)
               REDUCTION (operador | intrinsic : lista)

!$OMP  SECTION

   block

!$OMP  SECTION

    block

!$OMP END SECTIONS  [ NOWAIT ]
```

C/C++

```
#pragma omp sections [atributo ...]  nova_linha
                     private (lista)
                     firstprivate (lista)
                     lastprivate (lista)
                     reduction (operador: lista)
                     nowait
  {

  #pragma omp section   nova_linha

     estrutura de bloco

  #pragma omp section   nova_linha

     estrutura de bloco

  }
```

- **Observações**

  1-A diretiva SECTIONS define a seção do código sequêncial aonde será definida as seções independentes, através da diretiva SECTION;

  2-Cada SECTION é executada por uma “thread” do grupo;

  3-Existe um ponto de sincronização implícita no final da diretiva SECTIONS, a menos que, se especifique o atributo **nowait** (C/C++) ou **NOWAIT** (Fortran);

  4-Se existirem mais “threads” do que seções, o OpenMP decidirá, quais “threads” executarão os blocos de SECTION, e quais, não executarão.

Exemplo:

- Programa Adição de Vetores;
- O primeiro n/2 iterações do “loop” será distribuído para a primeira “thread” e o resto será distribuído para a segunda “thread”;
- As “threads” não possuem um ponto de sincronização devido ao atributo (NOWAIT).

Fortran

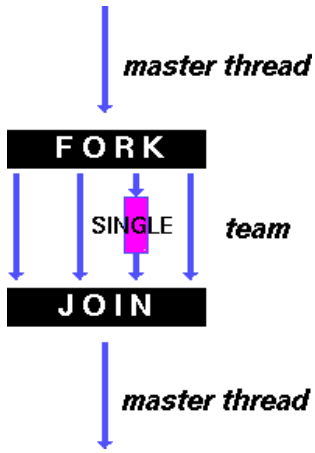
```
      PROGRAM VEC_ADD_SECTIONS

      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N)

!     Some initializations

      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO

!$OMP PARALLEL SHARED(A,B,C,N), PRIVATE(I)

!$OMP SECTIONS

!$OMP SECTION

      DO I = 1, N/2
         C(I) = A(I) + B(I)
      ENDDO

!$OMP SECTION

      DO I = 1+N/2, N
         C(I) = A(I) + B(I)
      ENDDO

!$OMP END SECTIONS NOWAIT

!$OMP END PARALLEL

      END
```

C / C++

```
#include <omp.h>
#define N     1000

main ()
{

int i, n;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
n = N;

#pragma omp parallel shared(a,b,c,n) private(i)
  {

  #pragma omp sections nowait
    {

    #pragma omp section
    for (i=0; i < n/2; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=n/2; i < n; i++)
      c[i] = a[i] + b[i];

    }  /* end of sections */

  }  /* end of parallel section */

}
```

Diretiva SINGLE

- Diretiva que determina que o código identificado seja executado por somente uma “thread” do grupo.



- Formato:

Fortran

```
!$OMP SINGLE [atributo ...]
             PRIVATE (lista)
             FIRSTPRIVATE (lista)

   código

!$OMP END SINGLE [ NOWAIT ]
```

C/C++

```
#pragma omp single [atributo ...]  nova_linha
                   private (lista)
                   firstprivate (lista)
                   nowait

     estrutura de bloco
```

- Os “threads” do grupo que não executam a diretiva SINGLE, esperam o fim do processamento da “thread” que executa a diretiva, a menos que se especifique o atributo **`nowait`** (C/C++) ou NOWAIT (Fortran).

Exercício 4

O objetivo desse exercício é entender como funciona a diretiva DO/for do OpenMP.

1 – Caminhe para a pasta ~/curso/openmp/ex4

cd ~/curso/openmp/ex4

2 – Edite o programa omp_do_for.f ou omp_do_for.c e codifique o que é solicitado nas linhas com as setas.

vi omp_do_for.f ou vi omp_do_for.c
ou
pico omp_do_for.f ou pico omp_do_for.c

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

xlf_r omp_do_for.f -qsmp=omp -o dofor

xlc_r omp_do_for.c -qsmp=omp -o dofor

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

ifort omp_do_for.f -openmp -o dofor

icc omp_do_for.c -openmp -o dofor

4 – Execução

Antes da execução do programa, será necessário especificar o número de "threads" que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

setenv OMP_NUM_THREADS <número de threads>

Ambiente LINUX/bash

export OMP_NUM_THREADS=<número de threads>

Exercício 5

Edite novamente o programa e faça as modificações necessárias para que o programa execute em paralelo, somente se, o número N for maior que 1000, e o loop paralelo do tipo estático sem tamanho definido. Varie o valor de N.

1 – Caminhe para a pasta ~/curso/openmp/ex5

```
cd  ~/curso/openmp/ex5
```

2 – Edite o programa omp_do_for.f ou omp_do_for.c e codifique o que é solicitado nas linhas com as setas.

```
vi  omp_do_for.f    ou   vi  omp_do_for.c
                    ou
pico omp_do_for.f    ou   pico  omp_do_for.c
```

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_do_for.f   -qsmp=omp  -o  dofor

xlc_r  omp_do_for.c  -qsmp=omp  -o  dofor
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_do_for.f   -openmp  -o dofor

icc omp_do_for.c  -openmp  -o  dofor
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

Exercício 6

Edite o programa omp_sections.f ou omp_sections.c e faça as modificações necessárias para que o programa execute em paralelo, e execute duas seções separadas, uma para cada "thread". Defina apenas 2 "threads" para a execução desse exercício.

1 – Caminhe para a pasta ~/curso/openmp/ex6

```
cd  ~/curso/openmp/ex6
```

2 – Edite o programa omp_sections.f ou omp_sections.c . Este programa possui na região paralela, duas seções que podem ser executadas em threads separadas. Identifique as seções e acrescente as diretivas threads necessárias.

```
vi  omp_sections.f    ou   vi  omp_sections.c
                      ou
pico omp_sections.f    ou   pico  omp_sections.c
```

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_sections.f   -qsmp=omp  -o  sections

xlc_r  omp_sections.c  -qsmp=omp  -o  sections
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_sections.f   -openmp  -o sections

icc omp_sections.c  -openmp  -o  sections
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de "threads" que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

4.5-Construções Combinadas e Compartilhadas PARALLEL

O objetivos das diretivas combinadas são as de reduzir o número de instruções OpenMP para determinadas regiões paralelas melhorando a compreensão do código. È uma construção mais conveniente.

Diretiva PARALLEL DO/for

- Determina uma região paralela e simultaneamente distribui as iterações do "loop" na região, por entre os "threads" de um grupo. A diretiva DO/for tem que vir imediatamente após a palavra PARALLEL.

- Formato:

Fortran

```
!$OMP PARALLEL DO [atributo ...]
                  IF (expressão lógica)
                  DEFAULT (PRIVATE | SHARED | NONE)
                  SCHEDULE (tipo [,chunk])
                  SHARED (lista)
                  PRIVATE (lista)
                  FIRSTPRIVATE (lista)
                  LASTPRIVATE (lista)
                  REDUCTION (operador | intrinsic : lista)
                  COPYIN (lista)

   "do loop"

!$OMP END PARALLEL DO
```

C/C++

```
#pragma omp parallel for [atributo ...]  nova_linha
                         if (expressão lógica)
                         default (shared | none)
                         schedule (tipo [,chunk])
                         shared (lista)
                         private (lista)
                         firstprivate (lista)
                         lastprivate (lista)
                         reduction (operador: lista)
                         copyin (lista)

   "for loop"
```

Exemplo

- As iterações serão distribuídas em blocos de mesmo tamanho por entre as threads.

Fortran

```
      PROGRAM VECTOR_ADD

      INTEGER N, I, CHUNK
      PARAMETER (N=1000)
      PARAMETER (CHUNK=100)
      REAL A(N), B(N), C(N)

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO

!$OMP PARALLEL DO
!$OMP& SHARED(A,B,C) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK)

      DO I = 1, N
         C(I) = A(I) + B(I)
      ENDDO

!$OMP END PARALLEL DO

      END
```

C / C++

```
#include <omp.h>
#define N       1000
#define CHUNK   100

main ()  {

int i, n, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
n = N;
chunk = CHUNK;

#pragma omp parallel for \
   shared(a,b,c) private(i) \
   schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

Diretiva PARALLEL SECTIONS

- Diretiva combinada que determina a região paralela e simultaneamente as seções que cada "thread" irá executar. A diretiva SECTIONS tem que vir imediatamente após a palavra PARALLEL.

- Formato:

Fortran

```
!$OMP PARALLEL SECTIONS [atributo ...]
                        DEFAULT (PRIVATE | SHARED | NONE)
                        SHARED (lista)
                        PRIVATE (lista)
                        FIRSTPRIVATE (lista)
                        LASTPRIVATE (lista)
                        REDUCTION (operador | intrinsic : lista)
                        COPYIN (lista)
                        ORDERED

   código

!$OMP END PARALLEL SECTIONS
```

C/C++

```
#pragma omp parallel sections [atributo ...]  nova_linha
                              default (shared | none)
                              shared (lista)
                              private (lista)
                              firstprivate (lista)
                              lastprivate (lista)
                              reduction (operador: lista)
                              copyin (lista)
                              ordered

   código
```

Exercício 7

O objetivo desse exercício é entender as características e os detalhes das diretivas combinadas PARALLEL DO do OpenMP.

1 – Caminhe para a pasta ~/curso/openmp/ex7

```
cd  ~/curso/openmp/ex7
```

2 – Edite o programa omp_combinado.f ou omp_combinado.c . Este programa possui na região paralela, um "loop" que pode ser executado em threads separadas, no entanto, alguns detalhes na codificação do programa e na regra de utilização do PARALLEL DO, ocasionam erros de compilação. Tente entender os erros e corrija-os; compare com o exemplo da apostila.

```
vi  omp_combinado.f  ou   vi  omp_combinado.c
                     ou
pico omp_combinado.f ou   pico  omp_combinado.c
```

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_combinado.f   -qsmp=omp  -o  comb

xlc_r  omp_combinado.c  -qsmp=omp  -o  comb
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_combinado -openmp  -o comb

icc omp_combinado -openmp  -o  comb
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de "threads" que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

4.6-Construções de Sincronização

Considere o exemplo abaixo no qual duas “threads” em dois processadores distintos, estão ambos tentando alterar a variável x ao mesmo tempo (x é inicialmente 0).

THREAD 1:
`increment(x)`
`{`
`  x = x + 1;`
`}`
THREAD 1:
`10  LOAD A, (x address)`
`20  ADD A, 1`
`30  STORE A, (x address)`
	THREAD 2:
	`increment(x)`
	`{`
	`  x = x + 1;`
	`}`
	THREAD 2:
	`10  LOAD A, (x address)`
	`20  ADD A, 1`
	`30  STORE A, (x address)`

- Para compreender o que acontece durante a execução, vamos visualizar as instruções em assembler. Uma possível execução seria:

  1-Thread 1 carrega o valor de x no registro A.
  2-Thread 2 carrega o valor de x no registro A.
  3-Thread 1 adiciona 1 ao registro A.
  4-Thread 2 adiciona 1 ao registro A.
  5-Thread 1 armazena o registro A no endereço x.
  6-Thread 2 armazena o registro A no endereço x.

- O resultado final será x=1, ao invés de 2.

- Para contornar essa situação, a alteração do valor de x deve ser sincronizado por entre as duas “threads”, para garantir o resultado correto. O OpenMP possui uma variedade de construções de sincronização que controla como sera a execução de cada “thread” em relação ao grupo de “threads”.

Diretiva MASTER

- Determina a região que será executada apenas pela “thread” mestre do grupo. Todas as outras “threads” do grupo pulam essa seção do código.

- Formato:

Fortran	**!$OMP MASTER** ***código*** **!$OMP END MASTER**
C/C++	**#pragma omp master *<nova_linha>*** ***código***

- Não existe sincronização implícita entre as “threads”.

Diretiva CRITICAL

- Determina que a região do código deva ser executada somente por uma “thread” de cada vez.

- Formato:

Fortran	**!$OMP CRITICAL *[nome]*** ***código*** **!$OMP END CRITICAL**
C/C++	**#pragma omp critical *[nome]* *<nova_linha>*** ***código***

- Se um “thread” estiver executando uma região CRITICAL as outras “threads” irão bloquear a execução quando alcançarem essa região, e cada uma executará a região por ordem de chegada.

- O nome de uma região CRITICAL é opcional.

Exemplo

- Ambas as seções irão executar em paralelo, em duas “threads” diferentes.
- Devido a diretiva CRITICAL em torno da expressão de adição de x, somente uma “thread” por vez irá ler, incrementar e escrever em x.

Fortran

```
      PROGRAM CRITICAL

      INTEGER X
      X = 0
!$OMP PARALLEL SHARED(X)
!$OMP SECTIONS

!$OMP SECTION
!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL

!$OMP SECTION
!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL

!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
      END
```

C / C++

```
#include <omp.h>

main()
{
int x;
x = 0;

#pragma omp parallel shared(x)
  {
  #pragma omp sections nowait
    {
    #pragma omp section
    #pragma omp critical
    x = x + 1;

    #pragma omp section
    #pragma omp critical
    x = x + 1;

    }  /* end of sections */
  }  /* end of parallel section */
}
```

Exercício 8

O objetivo desse exercício é entender a diretiva CRITICAL do OpenMP.

1 – Caminhe para a pasta ~/curso/openmp/ex8

```
cd  ~/curso/openmp/ex8
```

2 – Edite o programa omp_critical.f ou omp_critical.c . Este programa possui na região paralela, uma operação que deve ser executada serialmente pelas "threads", ou seja, uma de cada vez. Determine qual é a operação e adicione a diretiva apropriada para a execução correta do programa.

```
vi  omp_critical.f  ou   vi  omp_critical.c
                    ou
pico omp_critical.f ou   pico  omp_critical.c
```

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_critical.f   -qsmp=omp  -o  critical

xlc_r  omp_critical.c  -qsmp=omp  -o  critical
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_critical.f -openmp  -o critical

icc omp_critical.c -openmp  -o  critical
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de "threads" que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

Diretiva BARRIER

- Diretiva que sincroniza todas as “threads” de um grupo.
- Formato:

Fortran	`!$OMP BARRIER`
C/C++	`#pragma omp barrier` ***<nova_linha>***

- Quando uma “thread” executa essa diretiva, ela para de processar e espera por todas as “threads” chegarem nessa diretiva. Quando isso acontecer, as “threads” voltam a processar o restante do código.
- Todas as “threads” de uma região paralela devem executar a diretiva BARRIER.

Diretiva ATOMIC

- Diretiva que especifica, que imediatamente o próximo endereço de memória, seja atualizado “pontualmente” por cada “thread”, e não simultaneamente por todas as “threads”. Seria similar a diretiva CRITICAL.
- Formato:

Fortran	`!$OMP ATOMIC` ***expressião***
C/C++	`#pragma omp atomic` ***newline*** ***expressão***

- A “thread” executará, somente, imediatamente o próximo comando.

- Esse comando deverá possuir o seguinte formato:

Fortran	C / C++
x = x operador expr ***x = expr operador x*** ***x = intrinsica(x, expr)*** ***x = intrinsica(expr, x)***	***x binop = expr*** ***x++*** ***++x*** ***x--*** ***--x***
x é uma variável escalar ***expressão*** é uma expressão escalar que não faz referência a ***x*** ***intrinsica*** é uma função como: MAX, MIN, IAND, IOR, ou IEOR ***operador*** pode ser: +, *, -, /, .AND., .OR., .EQV., or .NEQV.	***x*** é uma variável escalar ***expressão*** é uma expressão escalar que não faz referência a ***x*** ***binop*** é um operador, podendo ser: +, *, -, /, &, ^, |, >>, or <<

- Importante: Somente a leitura e gravação da variável x será "atômica" ou pontual. A avaliação da expressão, não será, sendo feita simultaneamente por todas as "threads".

Diretiva FLUSH

- Determina a atualização dos dados compartilhados entre os "threads" em memória. Representa uma barreira, garantindo que todas as operações realizadas por todas as "threads" até o momento do flush foram realizadas.

- Formato:

Fortran	`!$OMP FLUSH (lista)`
C/C++	`#pragma omp flush (lista) <nova_linha>`

- Essa diretiva é necessária para instruir o compilador que as variáveis da lista, devem ser lidas ou escritas do sistema de memória, ou seja, as variáveis não podem permanecer nos registradores locais de cada cpu.

- É opcional fornecer as variáveis que se deseja atualizar os dados, mas se não for feito, todas as variáveis utilizadas no programa serão atualizadas na memória.

- A ação dessa diretiva é implícita em outras diretivas, desde que não seja utilizado o atributo NOWAIT.

Fortran	C / C++

BARRIER
CRITICAL e END CRITICAL
END DO
END PARALLEL
END SECTIONS
END SINGLE
ORDERED e END ORDERED
	`barrier`
	`critical`
	`ordered`
	`parallel`
	`for`
	`sections`
	`single`

Diretiva ORDERED

- Determina que as iterações do "loop", na região paralela, sejam executados na ordem sequêncial. Só pode ser utilizada junto com as diretivas, DO / for com o atributo ORDERED.

  DO ou PARALLEL DO (Fortran)
  for ou parallel for (C/C++)

- Formato:

Fortran

```
!$OMP DO ORDERED [atributos...]
   (loop)

!$OMP ORDERED

   (código)

!$OMP END ORDERED

   (fim do loop)
!$OMP END DO
```

C/C++

```
#pragma omp for ordered [atributos...]
   (loop)

#pragma omp ordered  <nova_linha>

   código

   (fim do loop)
```

- As "threads" necessitam esperar para executar a sua parte de um "loop", se as iterações anteriores, na ordem do "loop", ainda não tiverem terminado.

4.7-Regras de Operação das Diretivas

- As regras se aplicam a todas implementações de OpenMP, para os compiladores Fortran e C/C++;

- As diretivas DO/for, SECTIONS, SINGLE, MASTER e BARRIER só serão utilizadas se estiverem em uma região paralela definida pela diretiva PARALLEL. Se a região paralela não tiver sido definida, essas diretivas não terão nenhuma ação;

- A diretiva ORDERED só será utilizada junto com a diretiva DO/for;

- A diretiva ATOMIC tem ação em todas as “threads” utilizadas durante o processamento do programa;

- Uma diretiva PARALLEL dentro de outra diretiva PARALLEL, logicamente, estabelece um novo grupo, que será composto apenas pela “thread” que alcançar a nova região paralela, primeiro;

- A diretiva BARRIER não é permitida dentro das regiões definidas pelas diretivas DO/for, ORDERED, SECTIONS, SINGLE, MASTER e CRITICAL;

- Qualquer diretiva pode ser executada fora de uma região definida pela diretiva PARALLEL, no entanto será executado apenas pela “thread” MASTER.

Exercício 9

O objetivo desse exercício é entender a diretiva BARRIER do OpenMP.

1 – Caminhe para a pasta ~/curso/openmp/ex9

```
cd  ~/curso/openmp/ex9
```

2 – Edite o programa omp_barrier.f ou omp_barrier.c . Este programa possui um erro durante a execução. Tente entender o erro, verifique as regras de utilização das diretivas OpenMP. Corrija o problema.

```
vi  omp_barrier.f  ou   vi  omp_barrier.c
                    ou
pico omp_barrier.f ou   pico  omp_barrier.c
```

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

```
xlf_r  omp_barrier.f   -qsmp=omp  -o  barrier

xlc_r  omp_barrier.c  -qsmp=omp  -o  barrier
```

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

```
ifort  omp_barrier.f -openmp  -o barrier

icc omp_barrier.c -openmp  -o  barrier
```

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

```
setenv OMP_NUM_THREADS <número de threads>
```

Ambiente LINUX/bash

```
export OMP_NUM_THREADS=<número de threads>
```

5-Rotinas

O OpenMP padrão possui uma biblioteca com rotinas de API (Aplication Program Interface), que possibilitam uma grande variedade de funções:

- Verifica o número de “threads” ativas;
- Determina o número de “threads” que serão usadas;
- Função que bloqueia a execução;
- Função de cronometragem;
- Função que define ambiente de execução, como paralelismo recursivo e ajuste dinâmico de “threads”;
- Para o compilador C/C++ é necessário especificar o arquivo (include) “omp.h”;

5.1-OMP_SET_NUM_THREADS

- Esta subrotina/função determina o número de “threads” que serão utilizadas para a próxima região paralela;

- Formato:

Fortran	`SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)`
C/C++	`void omp_set_num_threads(int num_threads)`

- Esta rotina só pode ser executada na parte serial do código, antes definição de uma região paralela.

- Esta rotina tem precedência sobre a variável de ambiente OMP_NUM_THREADS.

5.2-OMP_GET_NUM_THREADS

- Esta subrotina/função retorna o número de “threads” que estão sendo utilizadas em uma região paralela.

Fortran	`INTEGER FUNCTION OMP_GET_NUM_THREADS()`
C/C++	`int omp_get_num_threads(void)`

- Esta função tem que ser executada dentro de uma região paralela.

5.3-OMP_GET_MAX_THREADS

- Esta subrotina/função retorna o número máximo de “threads” que o programa pode utilizar.

Fortran	`INTEGER FUNCTION OMP_GET_MAX_THREADS()`
C/C++	`int omp_get_max_threads(void)`

- Geralmente, reflete o número de “threads” definidos pela variável de ambiente OMP_NUM_THREADS ou pela função OMP_SET_NUM_THREADS().

- Esta função pode ser executada em qualquer parte do programa, na região serial ou na região paralela.

5.4-OMP_GET_THREAD_NUM

- Esta subrotina/função retorna a identificação da “thread” que está executando, dentro de um grupo. A identificação será um numero entre 0 e OMP_GET_NUM_THREADS-1. A “thread” mestre do grupo sempre será identificada como 0.

Fortran	`INTEGER FUNCTION OMP_GET_THREAD_NUM()`
C/C++	`int omp_get_thread_num(void)`

- Se a função for executada de uma região paralela recursiva ou da região serial, o valor retornado será 0.

Exemplos:

Fortran – Determinação do número de “threads” em uma região paralela.

Exemplo 1: Correto – Forma correta de determinar o número de “threads” numa região paralela.

```
      PROGRAM HELLO

      INTEGER TID, OMP_GET_THREAD_NUM

!$OMP PARALLEL PRIVATE(TID)

      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

      ...

!$OMP END PARALLEL

      END
```

Exemplo2 2: Incorreto – A variável TID, tem que ser PRIVATE (local).

```
      PROGRAM HELLO

      INTEGER TID, OMP_GET_THREAD_NUM

!$OMP PARALLEL

      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

      ...

!$OMP END PARALLEL

      END
```

Exemplo 3: Incorreto – A função **OMP_GET_THREAD_NUM** foi utilizada fora da região paralela.

```
      PROGRAM HELLO

      INTEGER TID, OMP_GET_THREAD_NUM

      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID

!$OMP PARALLEL

      ...

!$OMP END PARALLEL

      END
```

5.5-OMP_GET_NUM_PROCS

- Esta subrotina/função retorna o número de processadores disponíveis para a execução do programa.

Fortran	`INTEGER FUNCTION OMP_GET_NUM_PROCS()`
C/C++	`int omp_get_num_procs(void)`

5.6-OMP_IN_PARALLEL

- Esta subrotina/função pode ser chamada para determinar se uma seção do código está sendo executada em paralelo, ou não.

Fortran	`LOGICAL FUNCTION OMP_IN_PARALLEL()`
C/C++	`int omp_in_parallel(void)`

- Para o Fortran, a função retorna .TRUE. se for chamada de uma região que esteja executando em paralelo, caso contrário, retornará .FALSE. Para o C/C++, retornará um inteiro diferente de 0, se for paralelo, caso contrário, retornará 0.

5.7-OMP_SET_DYNAMIC

- Esta subrotina/função, ativa ou desativa a capacidade do programa de ajustar dinamicamente (durante a execução), o número de "threads" disponíveis para a execução de regiões paralelas.

Fortran	`SUBROUTINE OMP_SET_DYNAMIC(expressão lógica)`
C/C++	`void omp_set_dynamic(int dynamic_threads)`

- Para o Fortran, se a expressão resolver para .TRUE. permitirá o ajuste.
- Para o C/C++, se "dynamic_threads" tiver um valor diferente de 0, permitirá o ajuste.
- A subrotina/função OMP_SET_DYNAMIC tem precedência sobre a variável dinâmica OMP_DYNAMIC.
- Deve ser chamada da seção serial do programa.

5.8-OMP_GET_DYNAMIC

- Verifica se está disponível o ajuste dinâmico de “threads”.

Fortran	`LOGICAL FUNCTION OMP_GET_DYNAMIC()`
C/C++	`int omp_get_dynamic(void)`

- Para o Fortran, esta função retorna .TRUE. se ajuste dinâmico está disponível.
- Para o C/C++, esta função retorna um valor diferente de 0 se ajuste dinâmico está disponível.

5.9-OMP_SET_NESTED

- Esta função é utilizada para ativar ou desativar o paralelismo recursivo.

Fortran	`SUBROUTINE OMP_SET_NESTED(scalar_logical_expression)`
C/C++	`void omp_set_nested(int nested)`

- Fortran: .TRUE. – ativar .FALSE. – desativar.
- C/C++ : ≠ 0 – ativar = 0 – desativar.
- O paralelismo recursivo está ativo por “default”.

5.10-OMP_GET_NESTED

- Esta função verifica se o paralelismo recursivo está ativo ou não.

Fortran	`LOGICAL FUNCTION OMP_GET_NESTED()`
C/C++	`int omp_get_nested(void)`

- Fortran: .TRUE. – ativado .FALSE. – desativado.
- C/C++ : ≠ 0 – ativado = 0 – desativado.

5.11-OMP_INIT_LOCK

- Esta subrotina/função inicia para o OpenMP as variáveis de bloqueio, indicando para as "threads" as variáveis de bloqueio. A variável tem que ser definida com tipo LOCK

Fortran	`INTEGER(OMP_LOCK_KIND) var` `SUBROUTINE OMP_INIT_LOCK(var)`
C/C++	`void omp_init_lock(omp_lock_t *lock)`

5.12-OMP_DESTROY_LOCK

- Esta subrotina/função finaliza qualquer associação de bloqueio de uma determinada variável.

Fortran	`SUBROUTINE OMP_DESTROY_LOCK(var)`
C/C++	`void omp_destroy_lock(omp_lock_t *lock)` `void omp_destroy_nest__lock(omp_nest_lock_t *lock)`

5.13-OMP_SET_LOCK

- Esta subrotina/função solicita o bloqueio de uma variável, ou aguarda que uma variável bloqueada esteja disponível.

Fortran	`SUBROUTINE OMP_SET_LOCK(var)`
C/C++	`void omp_set_lock(omp_lock_t *lock)` `void omp_set_nest__lock(omp_nest_lock_t *lock)`

5.14-OMP_UNSET_LOCK

- Esta subrotina/função libera o bloqueio de uma determinada variável.

Fortran	`SUBROUTINE OMP_UNSET_LOCK(var)`
C/C++	`void omp_unset_lock(omp_lock_t *lock)` `void omp_unset_nest__lock(omp_nest_lock_t *lock)`

5.15-OMP_TEST_LOCK

- Esta subrotina/função testa e bloqueia, se uma variável de bloqueio, está disponível para ser bloqueada, mas sem parar a execução da "thread".

Fortran	`SUBROUTINE OMP_TEST_LOCK(var)`
C/C++	`void omp_test_lock(omp_lock_t *lock)` `void omp_test_nest__lock(omp_nest_lock_t *lock)`

- Fortran: .TRUE. – Ativado bloqueio .FALSE. – Não é possível o bloqueio.
- C/C++ : $\neq 0$ – Ativado bloqueio $= 0$ – Não é possível o bloqueio.

6-Variáveis de Ambiente

São parâmetros definidos no ambiente operacional antes da execução do programa. O OpenMP possui quatro definições de variáveis, que controlam a execução do código paralelo. O nome dessas variáveis deve vir sempre em letras maiúsculas.

6.1-OMP_SCHEDULE

- O valor dessa variável determina como as iterações de um "loop" são agendadas por entre os "threads". Essa variável de ambiente é utilizada somente para as diretivas DO, PARALLEL DO (Fortran) e `for, parallel for` (C/C++).

- Exemplos:

Ambiente csh (IBM/AIX)	Ambientes bsh/bash/ksh (Linux)
`setenv OMP_SCHEDULE "guided, 4"`	`export OMP_SCHEDULE="guided, 4"`
`setenv OMP_SCHEDULE "dynamic"`	`export OMP_SCHEDULE="dynamic"`

6.2-OMP_NUM_THREADS

- Define o número máximo de "threads" durante a execução.

- Exemplos:

Ambiente csh (IBM/AIX)	Ambientes bsh/bash/ksh (Linux)
`setenv OMP_NUM_THREADS 8`	`export OMP_NUM_THREADS=8`

6.3-OMP_DYNAMIC

- Habilita ou não, o ajuste dinâmico no número de "threads" para a execução de regiões paralelas. Valores possíveis TRUE ou FALSE.

- Exemplos:

Ambiente csh (IBM/AIX)	Ambientes bsh/bash/ksh (Linux)
`setenv OMP_DYNAMIC TRUE`	`export OMP_DYNAMIC=TRUE`

6.4-OMP_NESTED

- Habilita ou não, o paralelismo recursivo.

- Exemplos:

Ambiente csh (IBM/AIX)	Ambientes bsh/bash/ksh (Linux)
`setenv OMP_NESTED TRUE`	`export OMP_NESTED=TRUE`

Exercício 10

O objetivo desse exercício é entender a definição das variáveis nas diretivas do OpenMP.

1 – Caminhe para a pasta ~/curso/openmp/ex10

cd ~/curso/openmp/ex10

2 – Edite o programa omp_erro.f ou omp_erro.c . Este programa produz resultados incorretos. Verifique como foi definida e utilizada a variável de redução na região paralela.

vi omp_erro.f ou vi omp_erro.c
ou
pico omp_erro.f ou pico omp_erro.c

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

xlf_r omp_erro.f -qsmp=omp -o erro

xlc_r omp_erro.c -qsmp=omp -o erro

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

ifort omp_erro.f -openmp -o erro

icc omp_erro.c -openmp -o erro

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

setenv OMP_NUM_THREADS <número de threads>

Ambiente LINUX/bash

export OMP_NUM_THREADS=<número de threads>

Exercício 11

O objetivo desse exercício é tentar paralelizar um programa com “loops”.

1 – Caminhe para a pasta ~/curso/openmp/ex11

cd ~/curso/openmp/ex11

2 – Edite o programa loop.f ou omp_loop.c . Analise as regiões com “loops” e tente acrescentar as diretivas OpenMP, de maneira que, esses “loops” sejam executados pro “threads”.

vi loop.f ou vi loop.c
ou
pico loop.f ou pico loop.c

3 – Compilação

Ambiente IBM/AIX – Compilador: xlf_r e xlc_r – opção: -qsmp=omp

xlf_r loop.f -qsmp=omp -o loop

xlc_r loop.c -qsmp=omp -o loop

Ambiente INTEL/LINUX – Compilador: ifort e icc – opção: -openmp

ifort loop.f -openmp -o loop

icc loop.c -openmp -o loop

4 – Execução

Antes da execução do programa, será necessário especificar o número de “threads” que irão participar da execução. Defina um número que possa visualizar e entender o resultado, entre 2 e 4.

Ambiente IBM/csh

setenv OMP_NUM_THREADS <número de threads>

Ambiente LINUX/bash

export OMP_NUM_THREADS=<número de threads>

7-Referências

- **SP Parallel Programming II Workshop - "OpenMP".**
  Maui High Performance Computing Center. 1998.

- **The OpenMP web site. *www.openmp.org***

- **"OpenMP". Workshop Presentation. Lawrence Livermore National Laboratory**
  John Engle, October, 1998.

- **"Introduction to Parallel Programming ". Lawrence Livermore National Laboratory.**
  Blaise Barney, Livermore Computing.

- **Programação Paralela e Distribuída – OpenMP. DCC-IM / UFRJ**
  Gabriel P. Silva - Bacharelado em Ciência da Computação

- **Parallel Programming Using OpenMP**
  H. Birali Runesha and Shuxia Zhang

- **Ferramentas para Programação em Processadores Multi-Core**
  Prof. Dr. Gerson Geraldo H. Cavalheiro
  Departamento de Informática Universidade Federal de Pelotas

- **Introdução ao OpenMP**
  Fernando Silva
  DCC-FCUP

- **Operating System Concepts**
  Silberschatz, Galvin and Gagne

- **Multithread - Instituto de Matemática da UFRJ/NCE - Depto de Ciência da Computação**
  Luiz Paulo Maia