Programação em Memória Partilhada com o OpenMP

Ricardo Rocha

Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Po rto

Computação Paralela 2015/2016

OpenMP

O que significa OpenMP?

Open *specifications for* ***Multi Processing*** *via collaborative work between interested parties from the hardware and software industry, government and academia*

OpenMP

O que é o OpenMP:

- O OpenMP é um **modelo de programação em memória partilhada** que nasceu da cooperação (http://www.openmp.org) entre um grupo de grandes fabricantes de software e hardware (Sun, Intel, Fujitsu, IBM, AMD, HP, SGI, Compaq, ...)
- O OpenMP é uma **API para programação paralela de arquiteturas multiprocessor/multicore** definida inicialmente para ser usada em programas C/C++ (versão 1.0 publicada em 1998) ou Fortran (versão 1.0 publicada em 1997) sobre plataformas Unix/Linux ou Windows
- O OpenMP **não é a implementação, é apenas a especificação!**

OpenMP

Principais objetivos:

- Ser o standard de programação para arquiteturas de memória partilhada
- Estabelecer um conjunto muito simples e limitado de diretivas de programação
- Permitir a paralelização incremental de programas sequenciais
- Conseguir implementações eficientes em problemas de granularidade fina, média e grossa

Principais componentes:

- Diretivas de compilação
- Biblioteca de funções
- Variáveis de ambiente

Modelo de Programação do OpenMP

Paralelismo explícito: cabe ao programador anotar as tarefas para execução em paralelo e definir os pontos de sincronização. Essa anotação é feita por utilização de diretivas de compilação embebidas no código do programa.

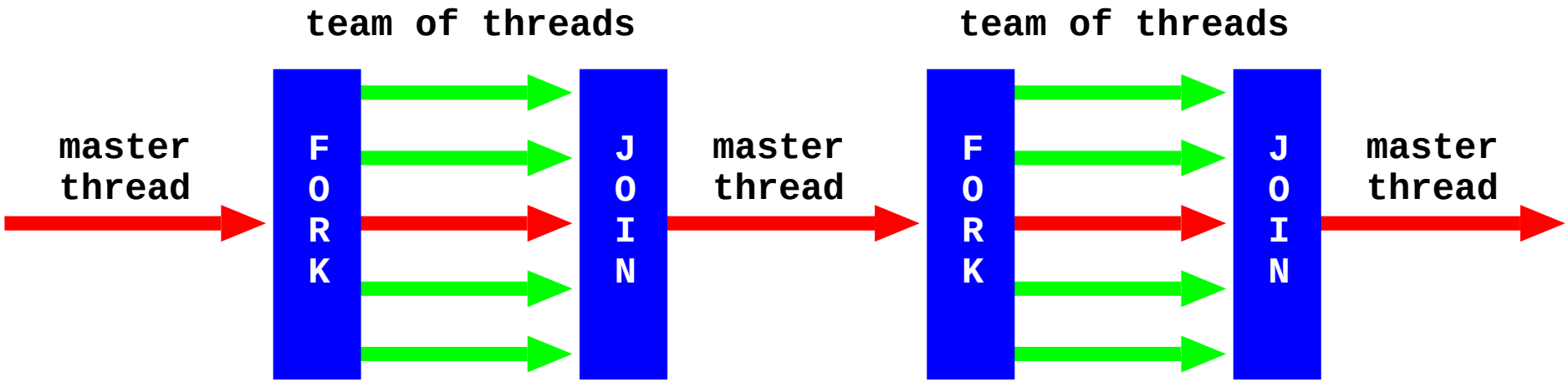
Multi-threaded implícito: um processo é visto como um conjunto de threads que comunicam por utilização de variáveis partilhadas. A criação, iniciação e terminação dos threads é feita de forma implícita pelo ambiente de execução, sem que o programador se tenha de preocupar com isso.

- O espaço de endereçamento global é partilhado por todos os threads
- As variáveis podem ser partilhadas ou privadas (duplicadas) para cada thread
- O controle, manuseamento e sincronização das variáveis envolvidas nas tarefas paralelas é transparente para o programador

Modelo de Execução Fork-Join do OpenMP

Todos os programas iniciam a sua execução com um processo, o **master thread**. O master thread executa sequencialmente até encontrar um **construtor paralelo**, altura em que cria um **team of threads**.

- O código delimitado pelo construtor paralelo é executado em paralelo pelo master thread e pelo team of threads
- Ao completarem a execução paralela, o team of threads sincroniza numa barreira implícita com o master thread
- O team of threads termina a sua execução e o master thread continua sequencialmente até encontrar um novo construtor paralelo



Estrutura Base de um Programa OpenMP

```
main() {
  ... // sequential region executed by master thread

  #pragma omp parallel // OpenMP parallel constructor
  { // master thread creates/launches the team of threads

  ... // parallel region executed by all threads

  } // team of threads sincronizes with master thread and terminates

  ... // sequential region executed by master thread
}
```

As definições da biblioteca OpenMP encontram-se em `omp.h` e a sua implementação em `libgomp.so`. Para compilar um programa com o OpenMP é necessário incluir o cabeçalho `#include <omp.h>` no início do programa e compilá-lo com a opção `-fopenmp`.

Diretivas de Compilação do OpenMP

```
#pragma omp directive [clause, ...]
```

`#pragma omp` é a funcionalidade básica de comunicar informação ao compilador de C/C++ de modo a este gerar código otimizado para o ambiente de execução do OpenMP.

- `directive` é uma das diretivas válidas do OpenMP
- `clause` permite especificar informação adicional sobre a diretiva

As diretivas do tipo `#pragma` (**pragmatic information**) permitem passar informação ao compilador para além do que está especificado na própria linguagem. No entanto, essa informação não deve ser essencial à execução do programa no sentido de que a sua não utilização deverá produzir igualmente um programa correto.

Compilação Condicional com o OpenMP

```
#ifdef _OPENMP
```

Para escrever código que funcione com e sem o OpenMP devemos ainda proteger as chamadas às funções do OpenMP com a diretiva `#ifdef _OPENMP`. A macro `_OPENMP` apenas é conhecida se a biblioteca do OpenMP estiver disponível.

```
// conditional compilation
#ifdef _OPENMP
  n = omp_get_num_threads(); // OpenMP function
#else
  n = 1;
#endif
```

Funções Básicas do OpenMP

```
int omp_get_num_threads(void)
```

`omp_get_num_threads()` retorna o número de threads momentaneamente ativos. Se for chamada a partir duma região sequencial (executada apenas pelo master thread) retorna 1.

```
omp_set_num_threads(int num_threads)
```

`omp_set_num_threads()` especifica o número máximo de threads que o ambiente de execução pode utilizar nas próximas regiões paralelas. Esta função só pode ser chamada a partir duma região sequencial.

Funções Básicas do OpenMP

```
int omp_get_max_threads(void)
```

`omp_get_max_threads()` retorna o número máximo de threads que o ambiente de execução pode utilizar numa região paralela.

```
int omp_get_num_procs(void)
```

`omp_get_num_procs()` retorna o número máximo de processadores que podem ser utilizados pelo programa.

Funções Básicas do OpenMP

```
int omp_get_thread_num(void)
```

`omp_get_thread_num()` retorna o identificador do thread corrente. Os N threads a executar numa região paralela são numerados de 0 a N-1 e o master thread é sempre identificado pelo número 0.

```
int tid;
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  printf("Thread id: %d\n", tid);
  ...
}
```

Funções Básicas do OpenMP

```
int omp_in_parallel(void)
```

`omp_in_parallel()` retorna 1 se for chamada a partir duma região paralela e 0 caso contrário.

```
double omp_get_wtime(void)
```

`omp_get_wtime()` retorna o tempo em segundos que passou desde um determinado ponto arbitrário no passado.

```
double omp_get_wtick(void)
```

`omp_get_wtick()` retorna a precisão da função `omp_get_wtime()`. Por exemplo, se `omp_get_wtime()` for incrementado a cada microsegundo então `omp_get_wtick()` retorna 0.000001.

Diretiva 'omp parallel'

```
#pragma omp parallel [clause, ...]
```

A diretiva `omp parallel` é o construtor fundamental do OpenMP. Indica que o bloco de código que se segue deve ser executado em paralelo. O número de threads que devem executar a região paralela é determinado pelos seguintes fatores, por ordem de precedência:

- Apenas o master thread se existir uma cláusula do tipo `if(expr)` em que a expressão `expr` é falsa
- Número definido pela expressão `expr` numa cláusula do tipo `num_threads(expr)`
- Número definido na última chamada a `omp_set_num_threads()`
- Número definido pela variável de ambiente `OMP_NUM_THREADS`
- Dependente da implementação (normalmente, o número de processadores disponíveis)

Cláusulas da Diretiva 'omp parallel'

```
#pragma omp parallel if(expr)
```

Executa em paralelo se a expressão **expr** for avaliada como verdade. Caso contrário, a execução é sequencial (apenas o master thread).

```
#pragma omp parallel num_threads(expr)
```

Executa em paralelo com um número de threads igual ao resultado da avaliação da expressão **expr**.

Cláusulas da Diretiva 'omp parallel'

```
#pragma omp parallel shared(list)
```

As variáveis definidas em `list` são partilhadas por todos os threads ficando à responsabilidade do programador garantir o seu correto manuseamento. Por omissão, as variáveis para as quais não é definido qualquer tipo são consideradas variáveis partilhadas.

```
#pragma omp parallel private(list)
```

As variáveis definidas em `list` são duplicadas em cada thread e o seu acesso passa a ser local (privado) em cada thread. O valor inicial das variáveis privadas é indefinido (não é iniciado) e o valor final das variáveis originais (depois da região paralela) também é indefinido.

Cláusulas da Diretiva 'omp parallel'

```
#pragma omp parallel firstprivate(list)
```

Igual a `private()` mas as variáveis privadas são iniciadas com o valor que as variáveis originais têm antes da região paralela.

```
#pragma omp parallel copyin(list)
```

Igual a `firstprivate()` mas para variáveis do tipo `threadprivate` (que persistem entre diferentes regiões paralelas).

```
#pragma omp parallel default(none)
```

Define que o tipo de todas as variáveis envolvidas na região paralela deve ser declarado explicitamente (sobrepõe-se à definição de que, por omissão, as variáveis são consideradas partilhadas).

Hello! (omp-hello.c)

```
main() {
  int n = NTHREADS, tid = -1;
  #pragma omp parallel if(n >= 1) num_threads(n) \
          default(none) private(tid) shared(n)
  {
    tid = omp_get_thread_num();
    printf("Thread %d: Hello!\n", tid);
    if (n != omp_get_num_threads())
      printf("Error: NTHREADS\n");
  }
  printf("Thread %d: Bye!\n", tid);
}
```

Se executarmos com 3 threads (NTHREADS) obtemos o seguinte output:

```
Thread 1: Hello!
Thread 0: Hello!
Thread 2: Hello!
Thread -1: Bye!
```

Atualização Exclusiva com o OpenMP

```
main() {
  ...
  #pragma omp parallel num_threads(NTHREADS) \
          private(tid) shared(result)
  {
     tid = omp_get_thread_num();
     result += thread_main(tid); // can be run in parallel?
  }
  // do something with the result
  ...
}

long thread_main(int tid) {
  ...
  return result;
}
```

Como a variável **result** é partilhada por todos os threads, é necessário sincronizar as operações de escrita, pois caso contrário, o resultado final do programa pode não ser o correto.

Cláusulas da Diretiva 'omp parallel'

```
#pragma omp parallel reduction(operator: list)
```

Define a lista de variáveis a serem utilizadas em operações de redução de informação. As variáveis definidas em `list` são duplicadas em cada thread e o seu acesso passa a ser privado. No final da região paralela, as variáveis originais ficam com o valor do resultado de aplicar o operador de redução `operator` a todas as cópias privadas em cada thread.

As variáveis definidas em `list` devem ser partilhadas e escalares (não podem representar vetores nem estruturas de dados).

Operações de redução sobre números em vírgula flutuante não são associativas e por isso podem originar resultados não determinísticos.

Cláusulas da Diretiva 'omp parallel'

```
#pragma omp parallel reduction(operator: list)
```

As expressões/operações de redução válidas são:

Tipo de Expressão	Operações Válidas
`var = var operator expr`	`+, *, -, /, &, ^, |, &&, ||`
`var = expr operator var`	`+, *, /, &, ^, |, &&, ||`
`var operator = expr`	`+, *, -, /, &, ^, |`
`var ++`	
`++ var`	
`var --`	
`-- var`	

Atualização Exclusiva com o OpenMP

```
main() {
  ...
  result = 0;
  #pragma omp parallel num_threads(NTHREADS) \
          private(tid) reduction(+: result)
  {
    tid = omp_get_thread_num();
    result += thread_main(tid); // reduction only done at the end
  }
  // do something with the result
  ...
}

long thread_main(int tid) {
  ...
  return result;
}
```

Construtores de Work-Sharing

Os construtores de work-sharing **permitem definir o modo de dividir trabalho entre os threads a executar numa região paralela**.

Os construtores de work-sharing não criam novos threads, apenas definem o modo de execução de blocos específicos de código dentro duma região paralela (por este motivo, só faz sentido utilizá-los dentro de regiões paralelas).

Todos os threads numa região paralela devem encontrar os mesmos construtores de work-sharing e pela mesma ordem.

Construtores de Work-Sharing

Ao encontrarem um construtor de work-sharing, todos os threads podem desde logo começar a executar a parte de código que lhes diz respeito, i.e., **à entrada** dos construtores de work-sharing **não existe qualquer barreira implícita de sincronização entre os threads**.

Ao completarem a região delimitada pelo construtor de work-sharing, por omissão, **todos os threads sincronizam numa barreira implícita**.

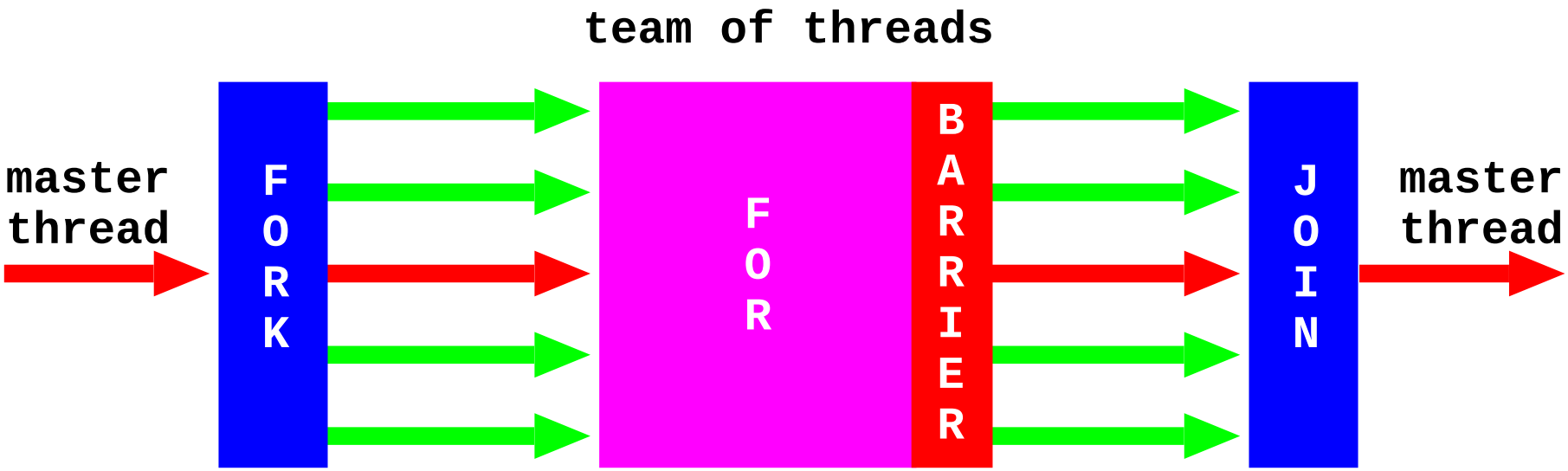
Existem 3 tipos de construtores de work-sharing:

- `#pragma omp for`
- `#pragma omp sections`
- `#pragma omp single`

Diretiva 'omp for'

```
#pragma omp for [clause, ...]
```

A diretiva `omp for` define que as iterações de um ciclo `for` devem ser divididas pelos threads na região paralela (**paralelismo nos dados**).



Diretiva 'omp for'

```
#pragma omp for [clause, ...]
```

A diretiva só pode ser utilizada quando o ciclo `for` está na forma canónica, i.e., quando **é possível determinar o número de iterações do ciclo**:

```
for (iter = start; iter { < | <= | >= | > } end; { iter ++ | ++ iter | iter - | - iter | iter += inc | iter -= inc | iter = iter + inc | iter = inc + iter | iter = iter - inc } )
```

Diretiva 'omp for'

```
#pragma omp for [clause, ...]
```

Para além disso, o ciclo **for** não pode conter instruções que terminem a sua execução prematuramente, i.e., não pode conter instruções do tipo **break**, **return**, **exit** e/ou **goto**.

Por omissão, todas as variáveis são consideradas variáveis partilhadas, com a excepção da variável utilizada para fazer a iteração do ciclo **for** que é considerada privada a cada thread.

```
#pragma omp for
for (iter = start; iter < end; iter++) {
  ... // by default, iter is considered private to each thread
}
```

Work-Sharing com o OpenMP (omp-worksharing.c)

```
main() {
  int i, j, start, end, a[N], b[N];

  for (i = 0; i < N; i++)
    a[i] = b[i] = i;
  end = N;
  #pragma omp parallel private(i,j) firstprivate(end) shared(start,a,b)
  for (i = 0; i < N; i++) { // i is private since it is incremented
    start = a[i]; // start can be shared since threads synchronize
    end -= start; // end is private since it is decremented
    if (start >= end) {
      printf("Exiting in iteration %d\n", i);
      break; // invalidates the usage of directive ‘omp for’
    }
    #pragma omp for // threads synhcronize at the end of the directive
    for (j = start; j < end; j++) // j is private by default
      b[j] += start;
  }
  ...
}
```

Cláusulas da Diretiva 'omp for'

```
#pragma omp for private(list)
```

Igual a `private()` da diretiva `omp parallel`.

```
#pragma omp for firstprivate(list)
```

Igual a `firstprivate()` da diretiva `omp parallel`.

```
#pragma omp for lastprivate(list)
```

Igual a `private()` mas, no final da região delimitada pelo construtor, as variáveis originais ficam com o valor da cópia privada que executa a última iteração do ciclo.

Cláusulas da Diretiva 'omp for'

```
#pragma omp for reduction(operator: list)
```

Igual a `reduction()` da diretiva `omp parallel`.

```
#pragma omp for ordered
```

Define que o ciclo pode conter blocos de código `#pragma omp ordered` em que as iterações devem ser executadas por ordem, i.e., pela ordem que seriam executadas num programa sequencial.

```
#pragma omp for nowait
```

Define que no final da região delimitada pelo construtor, o team of threads não necessita de sincronizar com o master thread.

Cláusulas da Diretiva 'omp for'

```
main() {
  ...
  #pragma omp parallel private(i,j,start) firstprivate(end) shared(a,b)
  for (i = 0; i < N; i++) {
    start = a[i]; // start is now private since threads not synchronize
    end -= start;
    if (start >= end) {
      printf("Exiting in iteration %d\n", i);
      break;
    }
    #pragma omp for nowait // now threads do not synchronize
    for (j = start; j < end; j++)
      b[j] += start; // can b[] still be shared?
  }
  ...
}
```

Como o número de iterações do ciclo associado a `#pragma omp for` não é constante, pode existir concorrência entre os threads nas atualizações do vetor `b[]`, o que fará com que o resultado final não seja o correto.

Cláusulas da Diretiva 'omp for'

```
#pragma omp for schedule(type [,chunk])
```

Define como é que as N iterações do ciclo `for` devem ser divididas pelos T threads da região paralela (**balanceamento de carga**). Os esquemas possíveis são:

- `static`: as iterações são divididas em conjuntos de N/T iterações contínuas que são depois atribuídos estaticamente a cada thread
- `static, C`: as iterações são divididas em conjuntos de `C` iterações contínuas que são depois atribuídos estaticamente a cada thread em round-robin
- `dynamic, C`: as iterações são divididas em conjuntos de `C` iterações contínuas que são depois atribuídos dinamicamente a cada thread à medida que estes terminam de executar o conjunto anterior (por omissão, `C` é 1)

Cláusulas da Diretiva 'omp for'

```
#pragma omp for schedule(type [,chunk])
```

- `guided, C`: idêntico a `dynamic` mas as iterações são divididas em conjuntos proporcionais ao número de threads e ao número de iterações por executar, decrescendo de forma exponencial até um mínimo de `C` iterações contínuas por conjunto (por omissão, `C` é 1). A ideia é ter um esquema adaptativo em que os conjuntos de iterações começam por ser maiores no início e menores no fim, diminuindo assim a probabilidade de um thread ficar sem nada para fazer prematuramente
- `runtime`: o esquema a utilizar é escolhido em tempo de execução a partir da variável de ambiente `OMP_SCHEDULE`

Por omissão, quando não é definido nenhum esquema, a maior parte das implementações utiliza o esquema `static`.

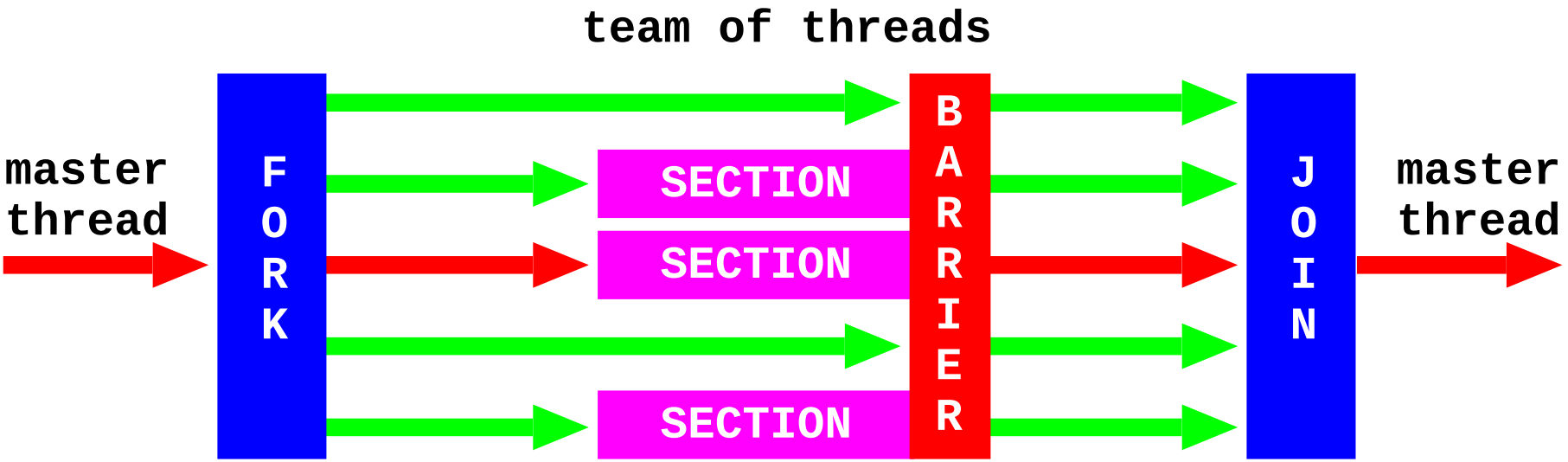
Cláusulas da Diretiva 'omp for'

```
chunk = N / 100;
#pragma omp parallel private(i,j) shared(a,b,c,d)
{
  #pragma omp for schedule(static,chunk) nowait
  for (i = 0; i < N; i++)
    b[i] = f(a[i]); // f() complexity is constant
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i = 0; i < N; i++)
    c[i] = g(a[i]); // g() complexity is not constant
  #pragma omp for private(j) schedule(guided) nowait
  for (i = 0; i < N; i++)
    for (j = i; j < N; j++)
      d[i] += f(a[j]); // the number of iterations is not constant
}
```

Diretiva 'omp sections'

```
#pragma omp sections [clause, ...]
```

A diretiva `omp sections` define um conjunto de secções independentes de código `#pragma omp section` que devem ser divididas pelos threads da região paralela (**paralelismo funcional**).



Diretiva 'omp sections'

```
#pragma omp sections [clause, ...]
```

Diferentes secções podem ser executadas por diferentes threads mas cada secção é executada por apenas um thread.

Se existirem mais threads do que secções, então alguns threads não executam nenhuma secção.

Se existirem mais secções do que threads, então alguns threads podem executar mais secções do que outros.

Diretiva 'omp sections'

```
#pragma omp parallel private(i) shared(a,b,c,d)
{
  #pragma omp sections
  {
    #pragma omp section
    for (i = 0; i < N; i++)
      c[i] = a[i] + b[i];
    #pragma omp section
    for (i = 0; i < N; i++)
      d[i] = a[i] * b[i];
  }
}
```

Cláusulas da Diretiva 'omp sections'

```
#pragma omp sections private(list)
```

Igual a `private()` da diretiva `omp for`.

```
#pragma omp sections firstprivate(list)
```

Igual a `firstprivate()` da diretiva `omp for`.

```
#pragma omp sections lastprivate(list)
```

Igual a `lastprivate()` da diretiva `omp for`.

Cláusulas da Diretiva 'omp sections'

```
#pragma omp sections reduction(operator: list)
```

Igual a `reduction()` da diretiva `omp for`.

```
#pragma omp sections nowait
```

Igual a `nowait()` da diretiva `omp for`.

Paralelismo Funcional com a Diretiva 'omp sections'

Consideremos o seguinte bloco de código:

```
a = alpha();
b = beta(a);
c = gamma(a);
d = delta(a);
e = epsilon(b,c);
f = zeta(d,e);
```

Como é que podemos paralelizar este bloco de código?

- O cálculo do valor de `a` não depende de outras variáveis
- O cálculo dos valores de `b`, `c` e `d` depende do valor de `a`
- O cálculo do valor de `e` depende dos valores de `b` e `c`
- O cálculo do valor de `f` depende dos valores de `d` e `e`

Paralelismo Funcional com a Diretiva 'omp sections'

Uma possível alternativa de paralelização seria:

```
a = alpha();
#pragma omp parallel shared(a,b,c,d)
{
  #pragma omp sections
  {
    #pragma omp section
    b = beta(a);
    #pragma omp section
    c = gamma(a);
    #pragma omp section
    d = delta(a);
  }
}
e = epsilon(b,c);
f = zeta(d,e);
```

Paralelismo Funcional com a Diretiva 'omp sections'

Outra possível alternativa de paralelização seria:

```
a = alpha();
#pragma omp parallel shared(a,b,c,d,e)
{
  #pragma omp sections
  {
    #pragma omp section
    b = beta(a);
    #pragma omp section
    c = gamma(a);
  }
  #pragma omp sections
  {
    #pragma omp section
    d = delta(a);
    #pragma omp section
    e = epsilon(b,c);
  }
}
f = zeta(d,e);
```

Paralelismo Funcional com a Diretiva 'omp sections'

Qual das duas alternativas de paralelização é melhor?

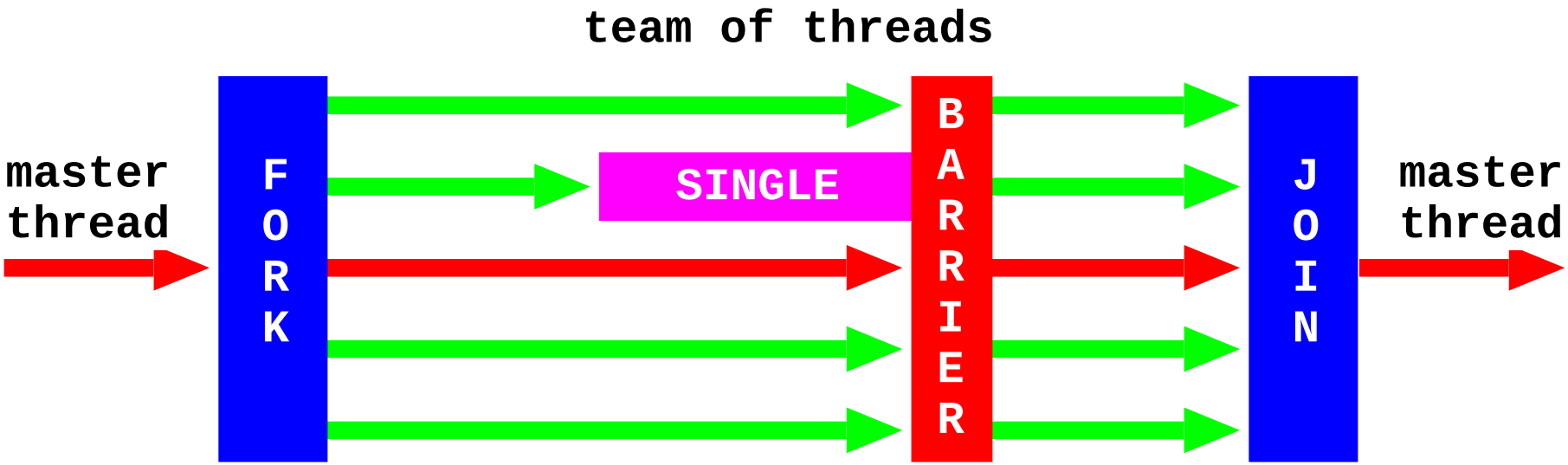
- A primeira alternativa define uma secção paralela de código para ser executada com 3 threads
  $T = T_{alpha} + max(T_{beta} + T_{gamma} + T_{delta}) + T_{epsilon} + T_{zeta}$
- A segunda alternativa define duas secções paralelas de código para serem executadas com 2 threads cada
  $T = T_{alpha} + max(T_{beta} + T_{gamma}) + max(T_{delta} + T_{epsilon}) + T_{zeta}$

Se o número de processadores disponíveis for de apenas 2, então a segunda alternativa pode resultar numa melhor eficiência. No entanto, essa eficiência depende do tempo de execução de cada uma das funções.

Diretiva 'omp single'

```
#pragma omp single [clause, ...]
```

A diretiva `omp single` define um bloco de código que deve ser executado por apenas um thread (**útil para sincronizar operações de I/O ou executar código thread-unsafe**).



Cláusulas da Diretiva 'omp single'

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

Igual a `private()` da diretiva `omp for`.

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

Igual a `firstprivate()` da diretiva `omp for`.

```
#pragma omp single nowait
```

Igual a `nowait()` da diretiva `omp for`.

Work-Sharing com o OpenMP (omp-worksharing.c)

```
main() {
  ...
  #pragma omp parallel private(i,j) firstprivate(end) shared(start,a,b)
  for (i = 0; i < N; i++) {
    start = a[i];
    end -= start;
    if (start >= end) {
      #pragma omp single nowait // only one thread prints the message
      printf("Exiting in iteration %d\n", i);
      break;
    }
    #pragma omp for
    for (j = start; j < end; j++)
      b[j] += start;
  }
  ...
}
```

Simplificação de Sintaxe

Quando uma região paralela é apenas definida por um construtor de work-sharing do tipo `omp for` ou `omp sections` então é possível estender a sintaxe da diretiva `omp parallel` de forma a definir desde logo o construtor de work-sharing.

```
#pragma omp parallel for [clause, ...]
```

```
#pragma omp parallel sections [clause, ...]
```

As diretivas estendidas herdam o conjunto das cláusulas das diretivas base. A única excepção é a cláusula `nowait` que não faz sentido numa diretiva estendida.

Simplificação de Sintaxe

```
#pragma omp parallel for [clause, ...]
```

```
chunk = N / 100;
#pragma omp parallel for private(i) shared(a,b) schedule(static,chunk)
for (i = 0; i < N; i++)
  b[i] = f(a[i]);
#pragma omp parallel for private(i) shared(a,c) schedule(dynamic,chunk)
for (i = 0; i < N; i++)
  c[i] = g(a[i]);
#pragma omp parallel for private(i,j) shared(a,d) schedule(guided)
for (i = 0; i < N; i++)
  for (j = i; j < N; j++)
    d[i] += f(a[j]);
```

Simplificação de Sintaxe

```
#pragma omp parallel sections [clause, ...]
```

```
a = alpha();
#pragma omp parallel sections shared(a,b,c,d)
{
  #pragma omp section
  b = beta(a);
  #pragma omp section
  c = gamma(a);
  #pragma omp section
  d = delta(a);
}
e = epsilon(b,c);
f = zeta(d,e);
```

Resumo das Cláusulas das Diretivas

Cláusula	Diretiva					
	`parallel`	`for`	`sections`	`single`	`parallel for`	`parallel sections`
`if`	*				*	*
`default`	*				*	*
`private`	*	*	*	*	*	*
`firstprivate`	*	*	*	*	*	*
`lastprivate`		*	*		*	*
`shared`	*	*			*	*
`reduction`	*	*	*		*	*
`copyin`	*				*	*
`nowait`		*	*	*		
`ordered`		*			*	
`schedule`		*			*	

Construtores de Sincronização

Os construtores de sincronização **permitem definir secções críticas de código e formas de sincronizar a execução numa região paralela**.

As diretivas associadas aos construtores de sincronização não aceitam cláusulas.

Construtores de Sincronização

```
#pragma omp barrier
```

A diretiva `omp barrier` define um ponto de sincronização explícito entre todos os threads numa região paralela. Cada thread bloqueia até que todos os threads atinjam a barreira.

```
#pragma omp master
```

A diretiva `omp master` define um bloco de código que deve ser executado apenas pelo master thread. Todos os restantes threads ignoram esse bloco de código. Não existe qualquer barreira implícita de sincronização entre os threads.

Construtores de Sincronização

```
#pragma omp critical[(name)]
```

A diretiva `omp critical` define um bloco de código (**zona crítica**) que deve ser executado de forma atómica (um thread de cada vez).

- A opção `name` permite definir zonas críticas múltiplas ou zonas críticas comuns
- Blocos de código com nomes diferentes definem zonas críticas diferentes
- Blocos de código sem nome ou com o mesmo nome definem a mesma zona crítica

Construtores de Sincronização

```
#pragma omp atomic
```

A diretiva `omp atomic` define que a próxima instrução de código, que deve ser uma atribuição a uma variável (**atribuição crítica**), deve ser executada de forma atómica (um thread de cada vez).

Sincroniza a escrita sobre a variável com base no seu endereço de memória. Isto permite que sejam efetuadas múltiplas atribuições concorrentes sobre variáveis diferentes.

Construtores de Sincronização

Os construtores `omp critical` e `omp atomic` podem ser usados para substituir a cláusula `reduction`, mas a sua eficiência é inferior.

```
#pragma omp parallel private(tid,partial_result) shared(result)
{
  tid = omp_get_thread_num();
  partial_result = thread_main(tid);
  #pragma omp critical
  result += partial_result; // critical section
}
```

```
#pragma omp parallel private(tid,partial_result) shared(result)
{
  tid = omp_get_thread_num();
  partial_result = thread_main(tid);
  #pragma omp atomic
  result += partial_result; // critical assignment
}
```

Construtores de Sincronização

A definição de atribuições críticas nem sempre é suficiente para garantir que o resultado final seja o correto.

```
max = a[0];
#pragma omp parallel for private(i) shared(a,max)
for (i = 1; i < N; i++)
  if (a[i] > max)
    #pragma omp atomic
    max = a[i];
```

Apesar da atribuição à variável `max` ser realizada de forma atómica, isso não garante que entre o testar da condição `if` e a atualização da variável `max` não existam atribuições concorrentes realizadas por outros threads.

É necessário definir uma zona crítica que inclua a condição de teste.

Construtores de Sincronização

Por outro lado, a definição de zonas críticas deve ser feita com algum cuidado, pois caso contrário podemos estar a sequencializar a execução.

```
max = a[0];
#pragma omp parallel for private(i) shared(a,max)
for (i = 1; i < N; i++)
  #pragma omp critical
  if (a[i] > max)
    max = a[i];
```

Todos os threads executam a instrução `if` de forma sequencial mesmo quando não existe nenhum efeito colateral na variável `max`.

Questão: Será que podemos reescrever este código de modo a evitar isso?

Construtores de Sincronização

No exemplo anterior, a zona crítica não está na execução da instrução `if` mas sim na eventual atribuição à variável `max` que daí possa resultar. Isso poderia ser resolvido da seguinte forma:

```
max = a[0];
#pragma omp parallel for private(i) shared(a,max)
for (i = 1; i < N; i++)
  if (a[i] > max) // non-critical section
    #pragma omp critical // critical section
    if (a[i] > max) // we need to test it again ...
      max = a[i]; // ... before updating max
```

Construtores de Sincronização

```
max = a[0];
min = a[0];
#pragma omp parallel for private(i) shared(a,min,max)
for (i = 1; i < N; i++) {
  if (a[i] > max)
    #pragma omp critical(critical_max)
    if (a[i] > max)
      max = a[i];
  if (a[i] < min)
    #pragma omp critical(critical_min)
    if (a[i] < min)
      min = a[i];
}
```

Construtores de Sincronização

```
#pragma omp ordered
```

A diretiva `omp ordered` define um bloco de código em que as iterações dum ciclo `for` devem ser executadas por ordem, i.e., pela ordem que seriam executadas num programa sequencial.

- Os threads só executam o seu conjunto de iterações quando todas as iterações anteriores estiverem completas
- A utilização deste construtor obriga a definir a cláusula `ordered` na diretiva `omp parallel for` correspondente

```
#pragma omp parallel for private(i) shared(a) ordered
for (i = 0; i < N; i++) {
  a[i] = complex_function(i);
  #pragma omp ordered
  printf("a[%d] = %d\n", i, a[i]); // ordered output
}
```

Construtores de Sincronização

```
#pragma omp threadprivate(list)
```

A diretiva `omp threadprivate()` declara que as variáveis em `list` são **variáveis globais privadas a cada thread**. Esta diretiva deve aparecer logo após a declaração das variáveis globais definidas em `list`.

- As variáveis definidas em `list` são duplicadas em cada thread e o seu acesso passa a ser local em cada thread
- Como são variáveis globais, o seu valor persiste entre diferentes regiões paralelas e, por omissão, o valor destas variáveis à entrada da primeira região paralela é indefinido a menos que se utilize a cláusula `copyin()` para iniciá-las com o valor das variáveis globais originais
- Em algumas implementações, para que o valor das variáveis `threadprivate` persista entre regiões paralelas não é possível ter threads dinâmicos (i.e., utilizar a cláusula `num_threads()` e/ou a função `omp_set_num_threads()`)

Variáveis Globais Privadas (omp-threadprivate.c)

```
int tid, a, b, c; float x;
#pragma omp threadprivate(a,b,x)

main() {
  a = b = c = 10;
  printf("Parallel Region I:\n");
  #pragma omp parallel private(c,tid) copyin(a,b) num_threads(4)
  {
    tid = omp_get_thread_num();
    a += tid;
    b += tid;
    c = tid;
    x = tid * 1.0;
    printf("Thread %d: (a b c x) = %d %d %d %f\n", tid, a, b, c, x);
  }
  printf("\nParallel region II:\n");
  #pragma omp parallel private(tid) copyin(a) num_threads(6)
  {
    tid = omp_get_thread_num();
    printf("Thread %d: (a b c x) = %d %d %d %f\n", tid, a, b, c, x);
  }
}
```

Variáveis Globais Privadas (omp-threadprivate.c)

Se executarmos o program obtemos o seguinte output:

```
Parallel Region I:
Thread 0: (a b c x) = 10 10 0 0.000000
Thread 1: (a b c x) = 11 11 1 1.000000
Thread 2: (a b c x) = 12 12 2 2.000000
Thread 3: (a b c x) = 13 13 3 3.000000

Parallel Region II:
Thread 0: (a b c x) = 10 10 10 0.000000
Thread 1: (a b c x) = 10 11 10 1.000000
Thread 2: (a b c x) = 10 12 10 2.000000
Thread 3: (a b c x) = 10 13 10 3.000000
Thread 4: (a b c x) = 10 0 10 0.000000
Thread 5: (a b c x) = 10 0 10 0.000000
```

Construtores de Sincronização

```
#pragma omp flush(list)
```

A diretiva `omp flush()` define um **ponto de sincronização** no qual os **valores das variáveis** definidas em `list` são **sincronizados em memória**. Por omissão, `list` é o conjunto de todas as variáveis partilhadas.

Isto significa que as variáveis definidas em `list` que possam ter valores atualizados apenas em registos da máquina, passam a ter esses valores sincronizados e atualizados com os valores em memória (semelhante à declaração `volatile` de variáveis).

Note que os protocolos de coerência das caches por si só não solucionam este problema quando o compilador otimiza partes do código através da alocação temporária duma variável a um registo, o que pode atrasar a atualização do valor da variável na linha de cache e respetivo endereço de memória associado à variável.

Construtores de Sincronização

```
#pragma omp flush(list)
```

Por si só, a diretiva `omp flush()` não implementa qualquer esquema de sincronização da computação. Ela apenas providencia um mecanismo de obter uma visão consistente da memória ao thread que executa a diretiva.

- Se uma variável partilhada é **modificada dentro de uma zona crítica** então o seu novo valor deve ser escrito para memória antes do fim da zona crítica de forma a ser visível por todos os threads
- Se uma variável partilhada é utilizada como **ponto de sincronização** então o seu valor deve ser lido a partir da memória de modo a garantir que o último valor escrito é o valor lido

Construtores de Sincronização

```
// producer thread
#pragma omp critical
{
  ...
  flag_is_set = TRUE;
  #pragma omp flush(flag_is_set)
}  // end of critical section


// consumer thread
do {
  #pragma omp flush(flag_is_set)
} while (flag_is_set == FALSE); // synchronization point
#pragma omp critical
{
  #pragma omp flush(flag_is_set)
  if (flag_is_set == TRUE) { // synchronization point
    ...
  }
}
```

Funções de Locking do OpenMP

As funções de locking do OpenMP permitem controlar de forma explícita o acesso a zonas críticas de código.

```
omp_init_lock(omp_lock_t *lock)
```

`omp_init_lock()` inicia um lock associado à variável `lock`.

```
omp_destroy_lock(omp_lock_t *lock)
```

`omp_destroy_lock()` elimina o lock associado à variável `lock`.

Funções de Locking do OpenMP

```
omp_set_lock(omp_lock_t *lock)
```

`omp_set_lock()` espera até conseguir obter o lock associado à variável `lock`.

```
omp_unset_lock(omp_lock_t *lock)
```

`omp_unset_lock()` liberta o lock associado à variável `lock`.

```
int omp_test_lock(omp_lock_t *lock)
```

`omp_test_lock()` faz tentativa de obter o lock associado à variável `lock` mas não bloqueia caso não seja possível.

Funções de Locking do OpenMP

```
omp_lock_t *lock;

max = 0;
omp_init_lock(lock);
#pragma omp parallel for private(i) shared(a,max,lock)
for (i = 0; i < N; i++)
  if (a[i] > max) {
    omp_set_lock(lock);
    if (a[i] > max)
      max = a[i];
    omp_unset_lock(lock);
  }
omp_destroy_lock(lock);
```

Funções de Locking do OpenMP

```
do_work () {
  int tid;
  omp_lock_t *lock;

  omp_init_lock(lock);
  #pragma omp parallel private(tid) shared(lock)
  {
    tid = omp_get_thread_num();
    while (omp_test_lock(lock) == 0)
      do_safe_code(tid);
    do_critical_code(tid);
    omp_unset_lock(lock);
  }
  omp_destroy_lock(lock);
}
```

Variáveis de Ambiente do OpenMP

O OpenMP define variáveis de ambiente que podem ser utilizadas para passar informação ao ambiente de execução do OpenMP:

- `OMP_NUM_THREADS` – define o número máximo de threads que o ambiente de execução pode utilizar numa região paralela
- `OMP_SCHEDULE` – define como é que as iterações de um ciclo `for` duma região paralela devem ser escalonadas para execução quando o tipo de execução definido pela cláusula `schedule` é `runtime`

```
export OMP_NUM_THREADS=10

export OMP_SCHEDULE="static"
export OMP_SCHEDULE="static,10"
export OMP_SCHEDULE="dynamic"
export OMP_SCHEDULE="dynamic,10"
export OMP_SCHEDULE="guided"
export OMP_SCHEDULE="guided,10"
```

Variáveis em Funções da Região Paralela

Quando uma **região paralela contém chamadas a funções**, qual é o âmbito das variáveis que aparecem nessas funções?

- Variáveis globais são consideradas `shared`
- Variáveis locais são consideradas `private`, exceto se forem do tipo `static`

Variáveis em Funções da Região Paralela

```
int w;

f(int a[], int n) {
  int i, j;
  #pragma omp parallel for private(w)
  for (i = 0; i < n; i++) {
    int q = w;
    w += i;
    for (j = 1; j <= 5; j++)
      g(&a[i], &q, j);
  }
}

g(int *x, int *y, int z) {
  static int s = 0;
  int k;
  s++;
  for (k = 0; k < z; k++)
    *x = *y + w;
}
```

Variáveis em Funções da Região Paralela

Fun	Var	Âmbito	Explicação	OK?
f()	`a[]`	shared	declarada fora do construtor `parallel`	
	`n`	shared	declarada fora do construtor `parallel`	
	`i`	private	iterador do ciclo do construtor `for`	
	`j`	shared	declarada fora do construtor `parallel`	Não
	`q`	private	declarada dentro do construtor `parallel`	
	`w`	private	cláusula private	Não
g()	`x`	private	argumento da função	
	`*x`	shared	referência à variável `a`	
	`y`	private	argumento da função	
	`*y`	private	referência à variável `q`	
	`z`	private	argumento da função	
	`s`	shared	variável static	Não
	`k`	private	variável local à função	
	`w`	shared	variável global	Não

Construtores Órfãos

Os construtores do OpenMP podem aparecer em regiões de código não protegidas diretamente pelo construtor `parallel`. Quando isso acontece, esses construtores são designados como **construtores órfãos**.

```
do_loop() {
  #pragma omp for // orphaned directive
  for ( )
    ...
}
```

Um construtor órfão pode no entanto ser chamado a partir duma região sequencial ou a partir duma região paralela.

```
do_something() {
  do_loop(); // called from a sequential region
  #pragma omp parallel
  do_loop(); // called from a parallel region
}
```

Construtores Órfãos

O que acontece então nesses dois casos?

- Quando chamado a partir duma região sequencial, o construtor órfão é simplesmente ignorado
- Quando chamado a partir duma região paralela, o comportamento do construtor órfão é o mesmo que nos casos em que aparece protegido diretamente pelo construtor `parallel`

```
do_loop() {
  #pragma omp for // orphaned directive
  for ( )
    ...
}

do_something() {
  do_loop(); // ignored
  #pragma omp parallel
  do_loop(); // treated as part of the parallel region
}
```

Construtores Órfãos

Para os construtores `ordered`, `critical` e `atomic` aplicam-se também as seguintes regras:

- Um construtor órfão do tipo `ordered` deve ser chamado a partir duma região protegida pela diretiva `omp for`
- Um construtor órfão do tipo `critical` define uma zona crítica relativamente a todos os threads em execução
- Um construtor órfão do tipo `atomic` define uma atribuição atómica relativamente a todos os threads em execução

Encadeamento de Construtores

No caso de existirem vários níveis de encadeamento de construtores órfãos ou não órfãos aplicam-se ainda as seguintes regras:

- Um construtor `parallel` encadeado com (dentro de) outro construtor `parallel` dá origem a um novo team of threads. No caso da implementação não suportar **nested parallelism** ou este não estar ativo, o novo team of workers é composto apenas pelo thread corrente que passa a ser o seu master thread
- Os construtores `for`, `sections` e `single` não podem ser encadeados entre si nem encadeados com os construtores `master`, `critical` e `ordered`
- O construtor `barrier` não pode ser encadeado com os construtores `for`, `sections`, `single`, `master`, `critical` e `ordered`
- O construtor `master` não pode ser encadeado com os construtores `for`, `sections` e `single`
- Os construtores `ordered` e `critical` não podem ser encadeados

Dependências nos Dados

Se uma instrução escreve/lê uma posição de memória e uma outra instrução escreve/lê a mesma posição, e pelo menos uma das duas instruções escreve nessa posição, então quando isso acontece diz-se que existe uma **dependência nos dados** nessa posição de memória entre as duas instruções.

Quando essas instruções fazem parte de um ciclo e a dependência nos dados ocorre entre iterações diferentes do ciclo, então tal dependência é designada como **loop-carried**. Geralmente, os ciclos sem dependências **loop-carried** podem ser diretamente paralelizáveis.

Dependências nos Dados

```
for (i = 1; i < N; i++)
  a[i] = a[i] + a[i - 1]; // with loop-carried dependencies
...
#pragma omp parallel for
for (i = 1; i < N; i += 2)
  a[i] = a[i] + a[i - 1]; // without loop-carried dependencies
...
#pragma omp parallel for
for (i = 0; i < N/2; i++)
  a[i] = a[i] + a[i + N/2]; // without loop-carried dependencies
...
for (i = 0; i < N/2 + 1; i++)
  a[i] = a[i] + a[i + N/2]; // with loop-carried dependencies
```

Dependências nos Dados

Um caso especial de dependências loop-carried acontece quando existem **atribuições condicionais durante a execução dum ciclo** (note que no caso da atribuição não ser condicional a dependência pode não existir).

```
x = 0;
for (i = 0; i < N; i++) {
  if (some_condition(i))
    x = new_value(i);
  a[i] = x; // x's value can be propagated to several iterations
}
```

Dependências nos Dados

No caso de existirem vários ciclos encadeados, normalmente queremos paralelizar o ciclo exterior. Quando assim acontece, podemos permitir a existência de dependências nos dados entre sub-iterações dos ciclos interiores para a mesma iteração do ciclo exterior.

```
#pragma omp parallel for
for (i = 0; i < N; i++) // without loop-carried dependencies for i
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++) // with loop-carried dependencies for k
      c[i,j] = c[i,j] + a[i,k] * b[k,j];
```

Dependências nos Dados

Outra forma de determinar dependências nos dados é classificar o fluxo de dados entre duas instruções dependentes:

- **Flow dependence** – acontece quando uma instrução S1 escreve para uma posição de memória e uma outra instrução S2 lê posteriormente essa posição. Nestes casos, a execução de S2 está condicionada pelo resultado de S1 (S1 pode ser visto como um produtor e S2 como um consumidor)
- **Anti dependence** – acontece quando uma instrução S1 lê uma posição de memória e uma outra instrução S2 escreve posteriormente para essa posição. Nestes casos, a execução prematura de S2 pode resultar numa leitura incorreta em S1
- **Output dependence** – acontece quando ambas as instruções S1 e S2 escrevem para a mesma posição de memória. Nestes casos, a execução prematura de S2 pode resultar num valor final incorreto na posição de memória

Dependências nos Dados

O exemplo seguinte resume os diferentes tipos de dependências nos dados.

```
for (i = 1; i < N - 1; i++) {
  x = d[i] + i;                          // instruction 1
  a[i] = a[i + 1] + x;                   // instruction 2
  b[i] = b[i] + b[i - 1] + d[i - 1]; // instruction 3
}
```

	Instrução S1			Instrução S2			Loop-	
Var	Inst	Iter	Acesso	Inst	Iter	Acesso	Carried	Dep
x	1	i	write	2	i	read	Não	flow
x	1	i	write	1	i + 1	write	Sim	output
x	2	i	read	1	i + 1	write	Sim	anti
a[i+1]	2	i	read	2	i + 1	write	Sim	anti
b[i]	3	i	write	3	i + 1	read	Sim	flow

Remoção de Dependências nos Dados

Anti dependence: S1 lê e S2 escreve.

```
for (i = 0; i < N - 1; i++) {
  x = d[i] + i;          // S2(x:i+1:w)
  a[i] = a[i + 1] + x; // S1(x:i:r) S1(a[i+1]:i:r) S2(a[i+1]:i+1:w)
}
```

Versão paralela com as dependências removidas:

```
#pragma omp parallel for shared(a,a2)
for (i = 0; i < N - 1; i++)
  a2[i] = a[i + 1];
#pragma omp parallel for private(x) shared(a,a2)
for (i = 0; i < N - 1; i++) {
  x = d[i] + i;      // x -> private
  a[i] = a2[i] + x; // a[i+1] -> a2[i]
}
```

Remoção de Dependências nos Dados

Output dependence: S1 escreve e S2 escreve.

```
for (i = 0; i < N; i++) {
  x = d[i] + i;       // S1(x:i:w) S2(x:i+1:w)
  a[i] = a[i] + x;
  c[1] = 2 * x;       // S1(c[1]:i:w) S2(c[1]:i+1:w)
}
y = x + c[1] + c[2]; // x and c[1] final values are read here
```

Versão paralela com as dependências removidas:

```
#pragma omp parallel for lastprivate(x,c1) shared(a)
for (i = 0; i < N; i++) {
  x = d[i] + i; // x -> lastprivate
  a[i] = a[i] + x;
  c1 = 2 * x;   // c[1] -> c1 -> lastprivate
}
c[1] = c1;
y = x + c[1] + c[2]; // c cannot be lastprivate because of c[2]
```

Remoção de Dependências nos Dados

Flow dependence: S1 escreve e S2 lê.

```
x = 0;
for (i = 0; i < N; i++)
  x = x + a[i]; // S1(x:i:w) S2(x:i+1:r)
```

Versão paralela com as dependências removidas:

```
x = 0;
#pragma omp parallel for reduction(+: x)
for (i = 0; i < N; i++)
  x = x + a[i]; // x -> reduction
```

Remoção de Dependências nos Dados

Flow dependence: S1 escreve e S2 lê.

```
sum = 0; exp = 1; j = N/2;
for (i = 0; i < N/2; i++) {
  a[i] = a[i] + a[j]; // S2(j:i+1:r)
  b[i] = sum;         // S2(sum:i+1:r)
  c[i] = exp;         // S2(exp:i+1:r)
  j = j + 1;          // S1(j:i:w) S2(j:i+1:r)
  sum = sum + i;      // S1(sum:i:w) S2(sum:i+1:r)
  exp = exp * 2;      // S1(exp:i:w) S2(exp:i+1:r)
}
```

Versão paralela com as dependências removidas:

```
#pragma omp parallel for shared(a,b,c)
for (i = 0; i < N/2; i++) {
  a[i] = a[i] + a[i + N/2]; // j -> i + N/2
  b[i] = i * (i + 1) / 2;   // sum -> i * (i + 1) /2
  c[i] = 2 ^ i;             // exp -> 2 ^ i
}
```

Remoção de Dependências nos Dados

Flow dependence: S1 escreve e S2 lê.

```
for (i = 1; i < N; i++) {
  b[i] = b[i] + a[i - 1]; // S2(a[i]:i+1:r)
  a[i] = a[i] + c[i];     // S1(a[i]:i:w)
}
```

Versão paralela com as dependências removidas:

```
b[1] = b[1] + a[0];
#pragma omp parallel for shared(a,b,c)
for (i = 1; i < N - 1; i++) { // without loop-carried dependencies
  a[i] = a[i] + c[i];         // S1(a[i]:i:w)
  b[i + 1] = b[i + 1] + a[i]; // S2(a[i]:i:r)
}
a[N - 1] = a[N - 1] + c[N - 1];
```

Remoção de Dependências nos Dados

Flow dependence: S1 escreve e S2 lê:

```
for (j = 1; j < N; j++) // with loop-carried dependencies for j
  for (i = 0; i < N; i++)
    a[i,j] = a[i,j] + a[i,j - 1]; // S1(a[i,j]:j:w) S2(a[i,j]:j+1:r)
```

Versão paralela com as dependências removidas:

```
#pragma omp parallel for shared(a)
for (i = 0; i < N; i++) // without loop-carried dependencies for i
  for (j = 1; j < N; j++)
    a[i,j] = a[i,j] + a[i,j - 1];
```

Remoção de Dependências nos Dados

Flow dependence: S1 escreve e S2 lê:

```
for (i = 1; i < N; i++) {
  a[i] = a[i] + a[i - 1]; // S1(a[i]:i:w) S2(a[i]:i+1:r)
  y = y + c[i];           // S1(y:i:w) S2(y:i+1:r)
}
```

Versão paralela com as dependências removidas:

```
for (i = 1; i < N; i++) // a[i] -> difficult to parallelize !?
  a[i] = a[i] + a[i - 1];
#pragma omp parallel for reduction(+: y)
for (i = 1; i < N; i++)
  y = y + c[i]; // y -> reduction
```

Remoção de Dependências nos Dados

Flow dependence: S1 escreve e S2 lê:

```
for (i = 0; i < N; i++) {
  y = y + a[i];                 // S1(y:i:w) S2(y:i+1:r)
  b[i] = (b[i] + c[i]) * y; // S2(y:i:r)
}
```

Versão paralela com as dependências removidas:

```
y2[0] = y + a[0];
for (i = 1; i < N; i++)
  y2[i] = y2[i - 1] + a[i];
y = y2[N - 1];
#pragma omp parallel for shared(b,c,y2)
for (i = 0; i < N; i++) // without loop-carried dependencies
  b[i] = (b[i] + c[i]) * y2[i]; // y -> y2[i]
```