Programação Paralela com OpenMP (Parte 1)

Ricardo Rocha e Eduardo R. B. Marques

Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Porto

Computação Paralela 2018/2019

OpenMP – Parte 1

Conteúdo:

- Introdução ao OpenMP.
- Directiva `omp parallel`, funções básicas do OpenMP.
- Variáveis públicas e privadas.
- Directivas `omp for` e `omp parallel for` (introdução, a revisitar na próxima aula).
- Cláusula `reduction`.
- Exclusão mútua com `omp atomic`, `omp critical`, e uso de locks.
- Sincronização com directivas `omp barrier` / `omp master` / `omp single` e cláusulas `nowait`, `ordered` e `lastprivate`.
- Paralelismo functional com `omp sections`.

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 (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!** (da mesma forma que a MPI ou a Pthreads).

OpenMP

Principais objetivos:

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

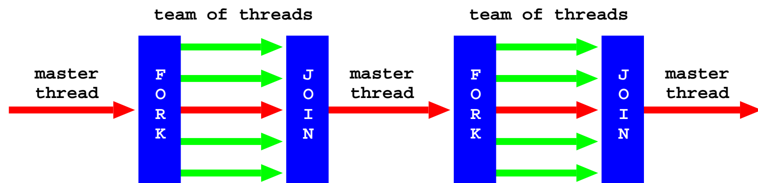
Modelo de Programação do OpenMP

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-threading 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, a partir de código especial gerado pelo **compilador**, 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



Um programa inicia a sua execução no contexto de uma thread, chamada **"master thread"**. Esta executa sequencialmente até encontrar uma **região paralela**, definida por um **construtor OpenMP**. Nessa altura:

1. O "master thread" cria (faz "fork") de um "team of threads".
2. O código da região paralela é executado de forma concorrente por todas as threads, incluindo as "master thread" e outras criadas.
3. No fim da região paralela, as threads sincronizam numa barreira implícita.
4. O "team of threads" termina a sua execução e o "master thread" continua sequencialmente até encontrar um novo construtor paralelo.

OpenMP API

A API OpenMP contempla:

- diretivas de compilação na forma `#pragma omp <directiva>`
- conjunto de funções C declaradas em `omp.h`
- algumas variáveis de ambiente (`OMP_NUM_THREADS`, ...).

Código OpenMP:

- deve incluir o "header file" `omp.h`

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

- e ser compilado com a opção **`-fopenmp`**

```
gcc  myCode.c -fopenmp ...
```

Diretivas de Compilação OpenMP

Diretivas de compilação `#pragma ...` podem ser usadas em C (ou C++) para passar informação ao compilador para além do que está especificado no código.

O OpenMP faz uso de directivas de compilação para indicar ao compilador a forma de execução de código em paralela, e assim este poder gerar código correspondente. Têm a forma geral:

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

- `directive` é uma diretiva válida do OpenMP
- `clause, ...` são cláusulas (opcionais) de parameterização

A directiva `omp parallel`

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

A diretiva `omp parallel` é o construtor fundamental do OpenMP, definindo uma região paralela. O número de threads que participam na região paralela é é determinado pelos seguintes fatores, por ordem de precedência:

- Apenas o "master thread" se existir uma cláusula `if(expr)` em que a expressão `expr` é falsa.
- Número definido por uma cláusula `num_threads(expr)`, em que `expr` define o número de threads.
- Número definido pela última chamada a `omp_set_num_threads(n)`
- Número definido pela variável de ambiente `OMP_NUM_THREADS`
- Dependente da implementação (normalmente, o número de processadores disponíveis).

Cláusulas `if` e `num_threads`

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

- executa em paralelo se a expressão **expr** for avaliada como verdade
- caso contrário, a execução é sequencial (envolve 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**

“Hello OpenMP!”

```
#include <omp.h>
...
int main(int argc, char** argv) {
  ...
  int n = atoi(argv[0]);
  omp_set_num_threads(n); // define number of threads
#pragma omp parallel
  {
    printf("Hello from thread %d / %d.\n",
           omp_get_thread_num(), omp_get_num_threads());
  }
#pragma omp parallel if(n % 2 == 0) num_threads(n-1) // use clauses
  {
    printf("Hello again from thread %d / %d.\n",
           omp_get_thread_num(), omp_get_num_threads());
  }
  return 0;
}
```

“Hello OpenMP!” (2)

```
[edrdo@vinicius hello_omp]$ ./hello_omp.bin 4
Hello from thread 3 / 4.
Hello from thread 0 / 4.
Hello from thread 1 / 4.
Hello from thread 2 / 4.
Hello again from thread 2 / 3.
Hello again from thread 1 / 3.
Hello again from thread 0 / 3.
```

- Execução entre threads na mesma região paralela é concurrente.
- Regiões paralelas executam sequencialmente.
- # threads definido por `omp_set_thread_num(n)` (4) na primeira região e pela cláusula `num_threads(n-1)` (3) na segunda.

“Hello OpenMP!” (3)

```
[edrdo@vinicius hello_omp]$ ./hello_omp.bin 5
Hello from thread 4 / 5.
Hello from thread 1 / 5.
Hello from thread 3 / 5.
Hello from thread 0 / 5.
Hello from thread 2 / 5.
Hello again from thread 0 / 1.
```

- Nesta execução, cláusula `if(n % 2 == 0)` determina que a segunda região seja executada apenas pela “master thread”.

Funções Básicas do OpenMP

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
void omp_set_num_threads(int num_threads);
```

- **omp_get_thread_num()** retorna o identificador do thread corrente, um valor de 0 a **n-1**, onde **n** é o número de threads activas. A "master thread" tem sempre o identificador 0.
- **omp_get_num_threads()** retorna o número de threads ativos (1 para uma região sequencial).
- **omp_set_num_threads(n)** especifica o número máximo **n** 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 (2)

```
int omp_in_parallel(void)
int omp_get_max_threads(void);
int omp_get_num_procs(void);
```

- `omp_in_parallel()` retorna 1 se chamada a partir duma região paralela e 0 caso contrário.
- `omp_get_max_threads()` retorna o número máximo de threads que o ambiente de execução pode utilizar numa região paralela.
- `omp_get_num_procs()` retorna o número máximo de processadores que podem ser utilizados pelo programa.

Funções Básicas do OpenMP (3)

```
double omp_get_wtime(void);
double omp_get_wtick(void);
```

- omp_get_wtime() retorna o tempo em segundos que passou desde um determinado ponto arbitrário no passado (“wall time”).
- omp_get_wtick() retorna a precisão da função omp_get_wtime() (exemplo 0.000001 para 1 microsegundo).

omp_get_wtime() é util para medir tempos de execução:

```
double time = - omp_get_wtime();
#pragma omp parallel
{
   ...
}
time += omp_get_wtime();
printf("Elapsed time: %f seconds\n", time);
```

Variáveis partilhadas e privadas

```
#pragma omp parallel shared(list1) private(list2) default(mod)
```

- **Variáveis partilhadas**, identificadas na cláusula `shared`, são partilhada por todas as threads. O programador deve garantir o seu correto manuseamento (ex. evitar "race conditions").
- **Variáveis privadas**, identificadas na cláusula `private` são privadas a/duplicadas por cada thread. Á entrada e saída da região paralela, os seus valores são indefinidos.
- Por omissão, variáveis são consideradas `shared`, a não ser que exista uma cláusula `default(mod)` que define o modo de partilha "default": `private`, `shared`, `none` — `none` obriga a declarar explicitamente o tipo de partilha por variável — ou ainda `firsprivate` (ver a seguir).
- Variáveis declaradas no interior da região paralela são por definição privadas. Podemos poupar um bom trabalho de anotação, declarando variáveis (em C99) a partir do âmbito em que são usadas.

Variáveis partilhadas e privadas – exemplo

```
int n = ...;
int *v = (int *) malloc(sizeof(int) * n);
int i, j;
#pragma omp parallel num_threads(n) default(none) \
                     shared(v,n) private(i,j)
{
   int tid = omp_get_thread_num();
   j = 0;
   for (i = 0; i < n * 100000; i++) j += tid;
   v[tid] = j;
}
```

- Variáveis privadas: `i`, `j` e `tid`. Note que `tid` é declarada dentro da região paralela.
- Variáveis partilhadas: `v` e `n`.
- Após a região paralela, qual o valor esperado de `v[tid]` para `tid=0, ..., n-1`?
- O que poderá acontecer se tornarmos `i` e `j` partilhadas?

Variáveis partilhadas e privadas – exemplo (2)

```
[edrdo@vinicius vars]$ ./vars.bin 5
=> Correct handling ... -- i + j are private
v[3] = 1500000 | addr(i)=0x7fc0f89f2e8c addr(j)=0x7fc0f89f2e88
v[1] =  500000 | addr(i)=0x7fc0f99f4e8c addr(j)=0x7fc0f99f4e88
v[2] = 1000000 | addr(i)=0x7fc0f91f3e8c addr(j)=0x7fc0f91f3e88
v[4] = 2000000 | addr(i)=0x7fc0f81f1e8c addr(j)=0x7fc0f81f1e88
v[0] =       0 | addr(i)=0x7fffd166a79c addr(j)=0x7fffd166a798
=> Incorrect handling -- i + j are shared
v[0] = 1203342 | addr(i)=0x7fffd166a7e8 addr(j)=0x7fffd166a7ec
v[4] = 1203344 | addr(i)=0x7fffd166a7e8 addr(j)=0x7fffd166a7ec
v[3] = 1203367 | addr(i)=0x7fffd166a7e8 addr(j)=0x7fffd166a7ec
v[1] = 1203361 | addr(i)=0x7fffd166a7e8 addr(j)=0x7fffd166a7ec
```

- Se `i` e `j` forem privadas teremos `v[tid] = tid * n * 100000`.
- Caso contrário o resultado é não-determinístico devido a condições de corrida sobre `i` e `j`.
- Note: o programa de teste imprime os endereços em memória de `i` e `j` em ambos os casos: distintos nas várias threads quando `private` e iguais quanto `shared`.

Cláusula firstprivate

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

- As variáveis são **private** mas os seu valores iniciais, ao invés de indefinidos, são dados pelos valores que têm à entrada da região paralela.
- Além da passagem de valores, a cláusula também pode ser útil por questões de eficiência, quando tornamos privadas variáveis partilhadas que são recorrentemente lidas (mas não escritas) na região paralela. A latência de leitura de variáveis privadas pode ser substancialmente menor.

“Dot product” revisitado

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  for (int i = 0; i < n; i++) {
     r += u[i] * v[i];
  }
  return r;
}
```

Vamos re-visitar o exemplo da aula sobre pthreads. A paralelisação usando OpenMP seguirá uma abordagem similar:

- partição das iterações / vector pelas várias threads
- redução dos resultados parciais de cada thread tendo o cuidado de evitar condições de corrida

"Dot product" – primeira aproximação

Esquecendo para já as condições de corrida, no que toca à partição podemos adaptar o código que vimos para a implementação Pthreads:

```
int dot_product(int* u, int* v, int n) {
   int r = 0;
   #pragma omp parallel
   {
     int rank = omp_get_thread_num();
     int my_n = n / omp_get_num_threads();
     int* my_u = &u[rank * my_n], *my_v = &v[rank * my_n];
     for (int i = 0; i < my_n; i++) r += my_u[i] * my_v[i];
   }
   return r;
}
```

A partição é feita "manualmente" pelo programador. O código tem um baixo nível de abstracção e é propenso a erros (ex. o que acontece se `n` não for divisível pelo número de threads?)

“Dot product” – uso da directiva `omp for`

A divisão de trabalho pode ser automatizada usando a directiva `#pragma omp for`.

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel
  {
    #pragma omp for reduction(+:r)
    for (int i = 0; i < n; i++)
       r += u[i] * v[i];
  }
  return r;
}
```

- A partição é feita automaticamente. Além disso, a cláusula `reduction(+:r)` especifica a redução necessária sobre `r`.
- Por omissão, maior parte das implementações atribuem N/T iterações contíguas a cada thread. O escalonamento é configurável com a cláusula `schedule`, a discutir mais tarde.

“Dot product” – desempenho

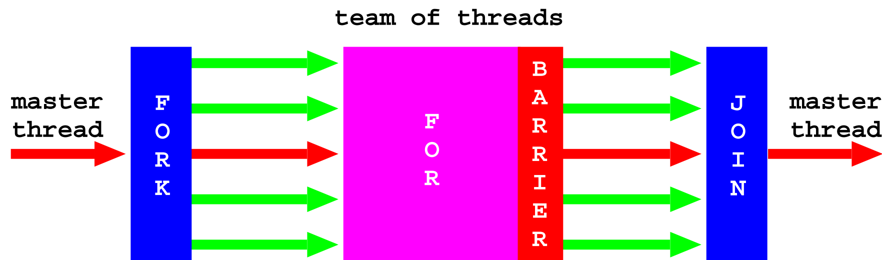
Threads	1	2	4	8	16	32
Tempo (s)	38.8	20.7	11.4	6.4	3.9	2.6
Speed-up	–	1.9	3.4	6.0	10.1	15.0

- Testes executados na máquina `sibila2.dcc.fc.up.pt` sob condições ideais (sem outra carga no sistema).
- Tempos médios de 5 execuções, cada uma com 50 000 chamadas a **`dot_product`** para um vector de tamanho 512 000.
- Observação geral: comportamento escalável, aproximadamente linear para < 8 threads. “Overheads” de sincronização / inicialização fork-join / hierarquia de memória tornam-se mais significativos à medida que o número de threads aumenta.

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` (2)

```
#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**:

$$\texttt{for}\left(\texttt{iter} = \texttt{start}; \texttt{iter} \begin{Bmatrix} \texttt{<} \\ \texttt{<=} \\ \texttt{>=} \\ \texttt{>} \end{Bmatrix} \texttt{end}; \begin{Bmatrix} \texttt{iter ++} \\ \texttt{++ iter} \\ \texttt{iter –} \\ \texttt{– iter} \\ \texttt{iter += inc} \\ \texttt{iter -= inc} \\ \texttt{iter = iter + inc} \\ \texttt{iter = inc + iter} \\ \texttt{iter = iter - inc} \end{Bmatrix}\right)$$

Diretiva omp for (3)

```
#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
}
```

Diretiva omp parallel for

```
#pragma omp parallel for <omp_parallel_clauses> <omp_for_clauses>
for (...) { code}
```

é equivalente a

```
#pragma omp parallel <omp_parallel_clauses>
{
  #pragma omp for <omp_for_clauses>
  for (...) { code }
}
```

A directiva `omp parallel for` (2)

O código de "dot product" pode ser então ser simplificado:

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel for reduction(+:r)
  for (int i = 0; i < n; i++)
     r += u[i] * v[i];
  return r;
}
```

... mas convém ter em atenção que neste caso queremos apenas executar o ciclo for na região paralela.

A directiva `omp parallel for` (3)

O seguinte código define **uma** região paralela, e portanto apenas um passo "fork-join". As threads envolvidas mantêm-se na mesma região paralela.

```
#pragma omp parallel
{
  #pragma omp for
  for (...) { ... }
  #pragma omp for
  for (...) { ... }
}
```

Por sua vez, o código a seguir define **duas** região paralelas / dois passos "fork-join", o que implicará maior latência

```
#pragma omp parallel for
for (...) { ... }
#pragma omp parallel for
for (...) { ... }
```

Cláusula reduction

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

- Aplicável a directivas `omp parallel` e `omp for`, entre outras.
- Define uma lista de variáveis `list` a serem utilizadas em operações de redução de informação, via operador `operator`. As variáveis devem ser partilhadas (`shared`) e de tipo primitivo (`int`, `double`, etc)[1]
- As variáveis em `list` são duplicadas em cada thread e o seu acesso passa a ser privado.
- No final da secção paralela em causa, **sem condições de corrida**, as variáveis originais ficam com o valor do resultado de aplicar o operador de redução `operator` a todas as cópias privadas em cada thread, e **ainda** ao valor inicial da variável.

[1] Operações de redução sobre números em vírgula flutuante não são associativas e por isso o resultado pode ter (normalmente ligeiras) variações não-determinísticas.

Cláusula reduction (2)

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

- Aplicável a directivas omp parallel e omp for, entre outras.
- Define uma lista de variáveis list a serem utilizadas em operações de redução de informação, via operador operator. As variáveis devem ser partilhadas (shared) e de tipo primitivo (int, double, etc)[2]
- As variáveis em list são duplicadas em cada thread e o seu acesso passa a ser privado.
- No final da secção paralela em causa, **sem condições de corrida**, as variáveis originais ficam com o valor do resultado de aplicar o operador de redução operator a todas as cópias privadas em cada thread, e **ainda** ao valor inicial da variável.

[2] Operações de redução sobre números em vírgula flutuante não são associativas e por isso o resultado pode ter (normalmente ligeiras) variações não-determinísticas.

Cláusula `reduction` (3)

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

Operadores válidos:

- Operadores aritméticos associativos: `+ * - min max`.
- Operadores lógicos e de manipulação de bits : `&& || & | ^`.
- É possível definir reduções customizadas usando a directiva `#pragma omp declare reduction`[3]

[3] Ver §2.15.3.6 da especificação OpenMP.

Cláusula `reduction` (4)

```
int a = 0, b = 5, c = -2, d = 3;
#pragma omp parallel num_threads(3) \
        reduction(+:a,b) reduction(min:c,d)
{
   a = b = c = d = omp_get_thread_num();
}
```

- Neste caso é feito uso da cláusula em associação a uma região paralela (construtor `omp parallel`).
- Após a região paralela teremos `a = 3`, `b = 8`, `c = -2` e `d = 0`
- Convém não esquecer o valor das variáveis à entrada da região paralela, que explicam os valores finais para `b` e `c`!

Suporte para exclusão mútua em OpenMP

No exemplo do "dot product" vimos que era possível usar a cláusula `reduction` para obter o resultado final sem condições de corrida, mas:

- Nem sempre se adequa ou é praticável uma redução para derivar o valor de uma variável partilhada.
- De forma mais geral, temos de lidar com a necessidade de garantir exclusão mútua numa região crítica arbitrária.

Vamos ver 3 formas de garantir exclusão mútua entre threads:

- Directiva `omp atomic`: aplicável a actualizações atómicas de variáveis.
- Directiva `omp critical`: aplicável a código arbitrário.
- Uso explícito de locks.

“Dot product” – versão com `omp atomic`

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel
  {
    int private_r = 0;
    #pragma omp for
    for (int i = 0; i < n; i++)
      private_r += u[i] * v[i];
    #pragma omp atomic
    r += private_r;
  }
  return r;
}
```

- `omp atomic` especifica que a próxima instrução de código deve ser uma atribuição a uma variável executa de forma atómica (**atribuição crítica**), i.e., uma thread de cada vez.
- Sincroniza escrita com base no endereço de memória, permitindo no caso geral múltiplas atribuições concorrentes.

"Dot product" – versão com `omp critical`

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel
  {
    int private_r = 0;
    #pragma omp for
    for (int i = 0; i < n; i++)
      private_r += u[i] * v[i];
    #pragma omp critical
    r += private_r;
  }
  return r;
}
```

- A diretiva `omp critical` define um bloco de código (**região crítica**) que deve ser executado de forma atómica (um thread de cada vez).
- Exclusão estrita (sem atender aos endereços de variáveis envolvidas), mas válida sobre código arbitrário.

“Dot product” – versão com locks

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  omp_lock_t lock;
  omp_init_lock(&lock); // initialize lock
  #pragma omp parallel
  {
    int private_r = 0;
    #pragma omp for
    for (int i = 0; i < n; i++)
      private_r += u[i] * v[i];
    omp_set_lock(&lock); // acquire lock
    r += private_r;
    omp_unset_lock(&lock); // release lock
  }
  omp_destroy_lock(&lock); // destroy lock
  return r;
}
```

- Operações envolvendo explicitamente locks, de forma análoga ao que já vimos em programas com PThreads.

“Dot product” – mau uso de exclusão mútua

```
int dot_product(int* u, int* v, int n) {
  int r = 0;
  #pragma omp parallel
  {
    #pragma omp for
    for (int i = 0; i < n; i++) {
      #pragma omp atomic
      r += u[i] * v[i];
    }
  }
  return r;
}
```

Esta variação do uso de `omp atomic` (e de forma análoga `omp critical` ou locks), embora aparentemente mais simples, inibe o potencial de paralelismo do código, pois força a uma sincronização a cada iteração do ciclo entre threads. Mesmo com poucas threads (ou apenas uma!) o custo da actualização atómica a cada iteração pode ser significativo (ver próximos slides).

"Dot product" – comparação de esquemas de exclusão mútua

Threads	1	2	4	8	16	32
`reduction`	4.1	2.2	1.2	0.8	0.4	0.3
`omp atomic`	3.8	2.0	1.0	0.6	0.4	0.3
`omp critical`	3.8	2.0	1.0	0.6	0.5	0.7
Locks	3.8	2.0	1.1	0.7	0.5	0.7

- Tempo médio (s) de 5 execuções na máquina `sibila2`, cada uma com 5000 chamadas a **`dot_product`** para vector de tamanho 512000.
- Redução e `omp atomic` escalam melhor para 16/32 threads do que `omp critical` e uso de locks (em que os tempos aliás pioram).
- `omp atomic` melhor para esta configuração e programa; no entanto é em geral recomendado o uso da redução para comportamento mais eficiente e escalável.

"Dot product" – comparação de esquemas de exclusão mútua (2)

Threads	1	2	4	8	16	32
`omp atomic` – mau uso	0.9	3.8	4.0	4.5	4.5	4.9

- Tempo médio (s) de 5 execuções na máquina `sibila2` da versão com mau uso de `omp atomic`, cada uma com **100** chamadas a **`dot_product`** (**em vez de 5000**) para vector de tamanho 512000.
- É visível que o program não escala, aliás o desempenho degrada-se à medida que aumenta o número de threads!
- Comparando com a versão que faz bom uso de `omp for`, em proporção a 50 000 iterações, o programa será cerca de 850 vezes mais lento para 32 threads!
- Para 1 thread não se paga o tempo de sincronização. Mas ainda assim o programa é aprox. 10 vezes mais lento em proporção às outras implementações!

Outras primitivas de sincronização

```
#pragma omp master
{  ... code ... }
#pragma omp single [nowait]
{ ... code ... }
#pragma omp barrier
```

- `omp master` identifica um bloco código que deve ser apenas executado pela "master thread". As outras threads ignoram o bloco e **não precisam de sincronizar** com a "master thread".
- `omp single` identifica um bloco código que deve ser apenas executado por uma thread. **Todas as threads sincronizam numa barreira implícita** até que o bloco seja executado, excepto se a cláusula `nowait` for definida.
- `omp barrier` define uma barreira explícita entre todas as threads.
- Vamos ver como podem ser empregues, re-visitando o exemplo do "Heat Diffusion".

“Heat diffusion” – passo de computação

```
#pragma omp for reduction(max:error)
for (int i=1; i <= cfg.N; i++) {
  for (int j=1; j <= cfg.N; j++) {
     x2[i][j] = x1[i][j] + cfg.alpha *
             (  x1[i][j-1] + x1[i][j+1] +
                x1[i-1][j] + x1[i+1][j]
                - 4.0 * x1[i][j] );
     double diff = fabs(x2[i][j] - x1[i][j]);
     if (diff > error) {
       error = diff;
     }
  }
}
```

- Constructor `omp for` no ciclo mais externo, com redução sobre a variável `error` com operador `max`.

“Heat diffusion” – esqueleto global

```
#pragma omp parallel firstprivate(cfg) num_threads(cfg.threads)
{
  double **x1 = Xold, **x2 = Xnew;
  for (int iter = 0; iter < cfg.max_iter; iter++) {
    // reset global error
    #pragma omp single
    error = 0.0;
    // computation step (previous slide)
    ...
    // master thread updates GUI (if enabled)
    #pragma omp master
    { ... }
    // break out of the loop if error lower than threshold
    if (error < cfg.error_threshold) break;
    // swap buffers for next iteration
    double** tmp = x1; x1 = x2; x2 = tmp;
    // synchronize all threads before next iteration
    #pragma omp barrier
  }
}
```

"Heat diffusion" – desempenho

Para grelha 512 x 512 e 10000 iterações:

Threads	1	2	4	8	16	32
Tempo (s)	23.1	12.4	5.3	3.5	1.8	1.5
Speed-up	–	1.9	4.4	6.6	12.9	15.6

Para grelha 5120 x 5120 e 100 iterações:

Threads	1	2	4	8	16	32
Tempo (s)	20.3	12.5	7.5	3.8	3.2	2.9
Speed-up	–	1.6	2.7	5.3	6.3	7.1

- Tempos médios de 5 execuções na máquina `sibila2`.
- Desempenho pior no segundo caso, explicável provavelmente por mais "overheads" na hierarquia de memória, em particular localidade de memória. Em aulas futuras, iremos ver como optimizar código tendo isso em conta.

Cláusulas nowait, ordered e lastprivate

```
#pragma omp for nowait
for (...)
#pragma omp for ordered ...
for(...)
   #pragma ordered
   { ... code ... }
}
#pragma omp for lastprivate(list)
for(...)
```

- `nowait`: indica as threads não precisam de sincronizar no fim do ciclo, podendo iniciar a computação que vem a seguir.
- `ordered`: indica que podem haver blocos `#pragma ordered` a executar na ordem de iteração do ciclo.
- `lastprivate(list)`: torna as variáveis em `list` privadas, transportando para fora do ciclo os valores da última iteração.

Cláusulas nowait e ordered

```
#pragma omp for nowait
for (int i = 0; i < n; i++) a[i] = ++x;
#pragma omp for
for (int i = 0; i < n; i++) b[i] = i*i;
#pragma omp for
for (int i = 0; i < n; i++) c[i] = a[i] + b[i];
#pragma omp for ordered lastprivate(x)
for (int i = 1; i < n; i++)
   #pragma omp ordered
   {
     c[i] += c[i-1]; x = c[i];
     printf("c[%d] = %d\n", c[i]);
   }
```

- Com `nowait` uma thread passa do 1º para o 2º ciclo (não há dependências de dados, o comportamento será correcto).
- No último ciclo a cláusula `ordered` implica que cada thread só executa o seu conjunto de iterações depois de concluídas as anteriores, e `lastprivate(x)` leva a que `x` fique com o valor `c[n-1]` no final.

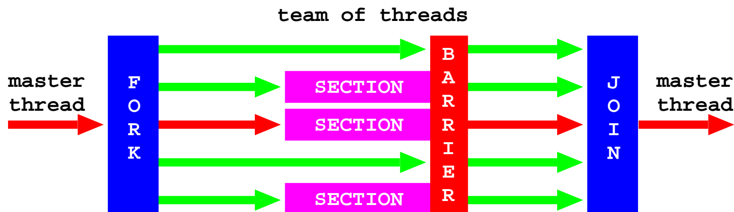
Directiva `omp sections`

```
#pragma omp sections [clause, ...]
{
   #pragma omp section
   { ... }
   ...
   #pragma omp section
   { ... }
}
```

- A directiva `omp sections` define um conjunto de secções de código que podem executar de forma concorrente, permitindo paralelismo functional entre as secções.
- Cada secção é identificadas por uma directiva `omp section` e é executada por apenas **uma** thread.
- Todas as threads, incluindo as que não se envolvam em nenhuma secção, sincronizam no final da região `omp sections`, a menos que a cláusula `nowait` seja especificada.

Directiva `omp sections` (2)

```
#pragma omp sections [clause, ...]
{
   #pragma omp section
   { ... }
   ...
   #pragma omp section
   { ... }
}
```



Obs.: Região `omp single` é equivalente a uma região `omp sections` com apenas uma secção.

Paralelismo functional com `omp sections`

```
x = f1();
a = f2(x);
b = f3(x,1);
c = f4(x,a);
d = f5(a,b);
e = f6(d);
```

Assumindo que `f1` a `f6` podem executar concorrentemente de forma correcta, como paralelisar o código? As dependências permitem a seguinte ordem de processamento:

1. `x = f1()` por apenas uma thread.
2. `a = f2(x)` e `b = f3(x,1)` por 2 threads em paralelo.
3. `c = f4(x,a)` e `d = f5(a,b)` por 2 threads em paralelo.
4. `e = f6(d)` por apenas uma thread.

Paralelismo functional com `omp sections` (2)

```
#pragma omp parallel
{
  #pragma omp single
  x = f1();
  #pragma omp sections
  {
    #pragma omp section
    a = f2(x);
    #pragma omp section
    b = f3(x,1);
  }
  #pragma omp sections
  {
    #pragma omp section
    c = f4(x, a);
    #pragma omp section
    d = f5(a,b);
  }
  #pragma omp single
  e = f6(d);
}
```