
# Módulo 12

# OpenMP – escalonamento

Considere o programa apresentado abaixo:

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

#define S 50000
double  a[S];

double fibonacci (int i) {
  double Fn = 0.0L, Fn1=1.0L, Fn2=0.0L;
  int j;

  for (j=1 ; j <= i ; j++) {
    Fn = Fn1 + Fn2; Fn2 = Fn1; Fn1 = Fn;
  }
  return (Fn);
}

main () {
  double T1, T2;
  int i;

  T1 = omp_get_wtime ();
  #pragma omp parallel for schedule(static)
  for (i=0 ; i<S ; i++) {
    a[i] = fibonacci(i);
  }

  T2 = omp_get_wtime ();
  printf ("Tempo = %.1lf secs\n", T2-T1);
}
```

Crie um ficheiro `pl12.c` com este código e compile-o usando o comando `gcc -O3 -fopenmp pl12.c -o pl12-static`

Note que a clásula `schedule(static)` indica ao OpenMP que o intervalo de valores que o índice do ciclo pode tomar deve ser dividido em tantos subintervalos consecutivos quanto o número de *threads* e que estes subintervalos terão a mesma cardinalidade. No exemplo acima, e para 2 threads, uma irá iterar para i=0 até 24999, e outra iterará para i=25000 até 49999.

**Exercício 1** – Execute pl12-static para 1 e 2 threads (`export OMP_NUM_THREADS=...`). Anote os respectivos tempos de execução. Qual o ganho ao passar de uma para duas *threads*? Porquê?

**Exercício 2** – Vamos medir o tempo que cada *thread* está activa, isto é, o tempo que cada *thread* passa a executar o ciclo `for`. Modifique o bloco paralelo de acordo com o código abaixo.
Execute para 2 *threads* e verifique o tempo de actividade da *thread* 0 e 1. A que se deve esta disparidade? Que impacto terá no tempo de execução total da aplicação?
**NOTA**: a cláusula `nowait` indica que as *threads* não devem sincronizar na barreira implícita no fim do bloco `for`. Isto é, em vez de a execução só prosseguir quando todas as *threads* terminarem o bloco `for`, as *threads* são autorizadas a prosseguir independentemente após este bloco. As *threads* sincronizarão no fim do bloco `parallel`.

```
#pragma omp parallel
{
double T2l, T1l;

T1l = omp_get_wtime();
#pragma omp for schedule(static) nowait
for (i=0 ; i<S ; i++) {
  a[i] = fibonacci(i);
  }
T2l = omp_get_wtime();
printf("Thread %d = %.1lf secs\n",omp_get_thread_num(),T2l-T1l);
}
```

**Exercício 3** – O Sistema Operativo da sua máquina disponibiliza uma aplicação para visualizar a ocupação dos CPUs. Nas máquinas do Laboratório, com Fedora13, clique no símbolo Fedora no canto inferior esquerdo do êcran, e seleccione "System" -> "System Monitor". Seleccione o Tab "System Load". Execute o programa para 2 *threads* e comente o que observa em termos da ocupação de cada um dos CPUs.

**Exercício 4** – O tempo de execução poderá ser melhorado reordenando a sequencia de Fibonacci, de forma a que as grandes sequencias fiquem distribuídas por todas as threads. Sugira algumas alternativas para o fazer. Ficam aqui duas sugestões (alíneas (a) e (b)). Experimente-as.

**NOTA**: Mantenha presente que uma invariante será "a soma do tempo activo de todas as *threads* é maior ou igual ao tempo de execução da versão sequencial". Se for maior então tal dever-se-á a custos adicionais associados ao paralelismo ou à ordem pela qual os dados são acedidos (por exemplo, menor localidade no acesso à cache). Se for menor, então terão que ter intervido outras optimizações que não apenas a utilização de

múltiplos *cores*, tais como por exemplo o código vectorizar na versa paralela e não na versão sequencial.

**a)** Garantir que cada thread calcula sequencias grandes e pequenas. Por exemplo para 2 *threads*:

```
const int S2 = S/2;
#pragma omp for schedule(static) nowait
    for (i=0 ; i<S2 ; i++) {
      a[i] = fibonacci(i);
      a[i+S2] = fibonacci(i+S2);
    }
```

**b)** Garantir que cada thread calcula as sequencias dos índices múltiplos do seu tid:

```
int NT = omp_get_num_threads(), ndx = omp_get_thread_num();
#pragma omp for schedule(static) nowait
for (i=0 ; i<S ; i++) {
   a[ndx] = fibonacci(ndx);
   ndx += NT;
}
```

**Exercício 5** – O OpenMP inclui a cláusula `dynamic,chunk_size`. Esta indica que o intervalo de iterações deve ser dividido em subintervalos com o tamanho especificado em `chunk_size`. Cada um destes subintervalos é atribuído a uma *thread* sempre que esta termina o subintervalo anterior. Se `chunk_size` não for especificado, o seu valor por omissão é 1.
Altere o código inicial para usar `schedule(dynamic,chunk_size)` e experimente para diferentes valores de `chunk_size`.