

1. Considere código C + OpenMP apresentado abaixo. Note que:

- a cláusula `#pragma omp single` garante que o bloco que se segue é executado apenas por **uma thread**, podendo esta ser qualquer *thread* do *team*; note ainda que **esta cláusula implica uma barreira no fim**, isto é as *threads* só prosseguem para as instruções seguintes quando todas as *threads* atingirem o fim deste bloco.

```
#pragma omp parallel
{ int i, first=false, tid = omp_get_thread_num ();
  double T;
  printf ("Thread %d starting\n", tid);
  #pragma omp for
    for (i=0; i < 300000 ; i++) do_work(i);
  #pragma omp single
  { T = omp_get_time ();
    first = true;
    printf ("Thread %d work done\n", tid);
  }
  if (first) printf ("1st finished in %.01f us\n", (omp_get_wtime()-T)*1e6);
  printf ("Thread %d finishing\n", tid);
}
```

Para uma execução com 3 *threads* indique qual dos *outputs* abaixo é possível.

<input type="checkbox"/>	Thread 1 starting Thread 0 starting Thread 0 work done 1st finished in 7 us Thread 2 starting Thread 2 finishing Thread 1 finishing Thread 0 finishing		<input type="checkbox"/>	Thread 1 starting Thread 0 starting Thread 2 starting Thread 0 work done Thread 2 finishing 1st finished in 7 us Thread 1 finishing Thread 0 finishing
<input type="checkbox"/>	Thread 1 starting Thread 0 starting Thread 2 starting Thread 2 finishing Thread 0 work done 1st finished in 7 us Thread 1 finishing Thread 0 finishing		<input type="checkbox"/>	Thread 1 starting Thread 0 starting Thread 2 starting Thread 0 work done Thread 0 finishing 1st finished in 7 us Thread 1 finishing Thread 2 finishing

2. Complete a afirmação abaixo :

“O ganho de desempenho obtido com a vectorização de código, relativamente à respectiva versão escalar, deve-se

- ☐ à diminuição do número médio de ciclos por instrução (CPI).”
- ☐ à diminuição do número total de operações matemáticas executadas sobre os dados.”
- ☐ a acessos mais rápidos à memória, devidos à maior localidade espacial imposta pelas intruções de `mov` vectoriais.”
- ☐ à diminuição do número total de instruções executadas (#I).”

3. O *loop unrolling* tem potencial para disponibilizar mais instruções para execução em paralelo num contexto de superescalaridade. Para o código abaixo seleccione a opção de *unrolling* que disponibiliza potencialmente mais *instruction level parallelism*.

```
int a[SIZE], i, sum=0;
for (i=0; i < SIZE ; i++) sum +=a[i];
```

<input type="checkbox"/>	<pre>int a[SIZE], i, sum=0; for (i=0; i &lt; SIZE ; i+=2) { sum +=a[i];   sum +=a[i+1]; }</pre>	<input type="checkbox"/>	<pre>int a[SIZE], i, sum=0, sum_a=0; for (i=0; i &lt; SIZE ; i+=2) { sum_a +=a[i];   sum +=a[i+1]; } sum += sum_a;</pre>
<input type="checkbox"/>	<pre>int a[SIZE], i, sum=0; for (i=0; i &lt; SIZE ; i+=4) { sum +=a[i];   sum +=a[i+1];   sum +=a[i+2];   sum +=a[i+3]; }</pre>	<input type="checkbox"/>	<pre>int a[SIZE], i, sum=0; for (i=0; i &lt; SIZE ; i+=4) { sum += a[i] + a[i+1];   sum += a[i+2] + a[i+3]; }</pre>

4. Considere um processador superescalar com 2 unidades funcionais (UF):

**UF1 (Op)** – realiza operações lógicas e aritméticas sobre inteiros;

**UF2 (LS + B)** – realiza acessos à memória (*Load/Store*) e saltos (*branches*).

Considere que cada uma destas unidades funcionais executa **uma instrução por ciclo do relógio** (isto é, não há nenhuma operação que exija mais do que um ciclo do relógio na respectiva UF). Considere ainda o seguinte excerto de código:

```
I1:  movl (%ebx, %edx, 4), %esi
I2:  addl %esi, %eax
I3:  incl %edx
I4:  decl %ecx
I5:  jnz I1
```

Preencha, para a primeira iteração do ciclo, a tabela 1 considerando um escalonamento *static in-order scheduling* e a tabela 2 considerando um escalonamento *dynamic out-of-order*:

- Para preencher as tabelas indique, para cada instrução, qual a unidade funcional em que é escalonada e em que ciclo do relógio. Calcule também o CPI exibido por essa primeira iteração.
- sugere-se que numa folha à parte preencha primeiro uma tabela com o layout apresentado a seguir e apenas depois converta para o formato pedido (note que nesta tabela nas células indica qual a instrução correspondente):

Ciclo	U1	U2
1		
2		
⋮		

Tabela 1 - <i>static in-order</i>	
UF (1 ou 2)	Ciclo
CPI =	

I1  
I2  
I3  
I4  
I5

Tabela 2 - <i>dynamic out-of-order</i>	
UF (1 ou 2)	Ciclo
CPI =	

Use o espaço abaixo para alguma justificação que lhe pareça pertinente.

Nome: \_\_\_\_\_

Número: \_\_\_\_\_

5. O código apresentado abaixo, que explora *Thread Level Parallelism* recorrendo ao OpenMP, pretende calcular a soma de alguns elementos de cada linha  $i$  de uma matriz (elementos das colunas 1 a  $i$ ) e armazenar o resultado no primeiro elemento dessa linha ( $a[i][0]$ ) :

<pre>#define W 400000 int a[W][W]; int sum, i, j; ...</pre>	<pre>#pragma omp parallel for for (i=0 ; i &lt; W ; i++) {     sum = 0;     for (j=1 ; j &lt;= i ; j++) sum += a[i][j];     a[i][0] = sum; }</pre>
---	--

- a) O resultado da execução deste programa com múltiplas *threads* é indeterminado, pois contém alguns erros semânticos. Identifique esses erros e diga como os corrigiria.

---

**NOTA:** Os erros semânticos não estão relacionados com o desempenho, mas sim com a correcção do programa.

---

- b) Associada à directiva `#pragma omp parallel for` usaria para o escalonamento a cláusula `schedule(static)` ou `schedule(dynamic)` ? Justifique