Questionnaire

Teaching Parallel Programming in the First Year of an Undergraduate Computer Science Curriculum

Author: Leonardo B. A. Vasconcelos

Collaborators: Max V. Machado, Luís Fabrício W. Góes, Carlos Augusto P. S. Martins and Henrique C. Freitas

This activity was planned to students of the course called Algorithms and Data Structures II of the undergraduate Computer Science curriculum at PUC Minas and it can be ported to any course related to OpenMP programming.

The goal is to evaluate the previous knowledge related to parallelism in order to check the student learning during the teaching approach.

Group 1/8

Note: Next two questions are related to parallelism perception.

1) A processor executes three arithmetic instructions. The time for processing each instruction is 1 second. There is one single core available. What is the minimum time to execute the following code? [Rague, B. 2011]

```
1. a1 = a2 + a3
2. a4 = a5 + a6
3. a7 = a8 + a9
```

a) ( ) 2 seconds

b) ( ) 3 seconds

c) ( ) 4 seconds

d) ( ) 6 seconds

2) Based on the same example from the previous question, there are two cores available. What is the minimum time to execute the following code? [Rague, B. 2011]

a) ( ) 2 seconds

b) ( ) 3 seconds

c) ( ) 4 seconds

d) ( ) 6 seconds

Group 2/8

Note: Next two questions are related to computer architecture.

3) A student has developed a code in C with OpenMP. As the first experiment, he executed the code onto two different machines. The first one has a **single-core processor** with no Simultaneous Multithreading (SMT) support and the second one is a **single-core processor** with SMT support. Choose the correct affirmative based on the processing for each processor, respectively:

a) ( ) parallel and sequential processing

b) ( ) sequential and sequential processing

c) ( ) sequential and parallel processing

d) ( ) parallel and parallel processing

4) A student has developed a code in C with OpenMP. As the first experiment, he executed the code onto two different machines. The first one has a **multi-core processor** with no Simultaneous Multithreading (SMT) support and the second one is a **multi-core processor** with SMT support. Choose the correct affirmative based on the processing for each processor, respectively:

a) ( ) parallel and sequential processing

b) ( ) sequential and sequential processing

c) ( ) sequential and parallel processing

d) ( ) parallel and parallel processing

Group 3/8

Note: Next two questions are related to OpenMP variable scope. In parallel programming, we have shared and private variables. Multiple threads can access them simultaneously.

5) In the following OpenMP code, the x, y and z variables are used in the region of the **omp parallel** directive. Choose the correct option about each variable scope:

```
int x, y;
#pragma omp parallel shared(x) private(y)
{
  int z;
  z = x + y;
}
```

a) ( ) shared, private and private

b) ( ) private, private and shared

c) ( ) shared, shared, shared

d) ( ) shared, private and shared

6) In the following OpenMP code, the x, y and z variables are used in the region of **omp parallel** directive. Choose the correct option about each variable scope:

```
int x, y ;
#pragma omp parallel
{
  int z;
  z = x + y;
}
```

a) ( ) private, private and private

b) ( ) shared, shared and private

c) ( ) shared, shared, shared

d) ( ) shared, private and shared

Group 4/8

Note: Next two questions have two native OpenMP functions called **omp_get_thread_num()** and **omp_get_num_threads()**.

7) About the following code, x and y variables receive an integer. Choose the correct answer based on the **omp_get_thread_num()** and **omp_get_num_threads()** behaviors:

```
#pragma omp parallel
{
  y = omp_get_thread_num();
  x = omp_get_num_threads();
}
```

a) ( ) Total number of threads in execution and the all thread IDs, respectively.

b) ( ) The local thread ID and the total number of threads in execution, respectively.

c) ( ) The local thread ID and the all thread IDs, respectively.

d) ( ) All thread IDs and the total number of threads in execution, respectively.

8) We can use **omp_get_thread_num()** and **omp_get_num_threads()** to divide tasks among threads. The following code is executed on a 6-core processor. Choose the incorrect answer:

```
int global_input_size = 120;
#pragma omp parallel shared (global_input_size)
{
  int local_size = global_input_size/omp_get_num_threads();
  int start = local_size * omp_get_thread_num();
  int end = start + local_size;
  doanything(start, end, array)
}
```

a) ( ) For the thread ID number 3, the variable values are: start = 60 and end = 80.

b) ( ) For the thread ID number 5, the variable values are: start = 100 and local_size = 20.

c) ( ) The local_size value is equal for all threads while star and end are different.

d) ( ) For the thread ID number 0, the variable values are: start = 20 and end = 40.

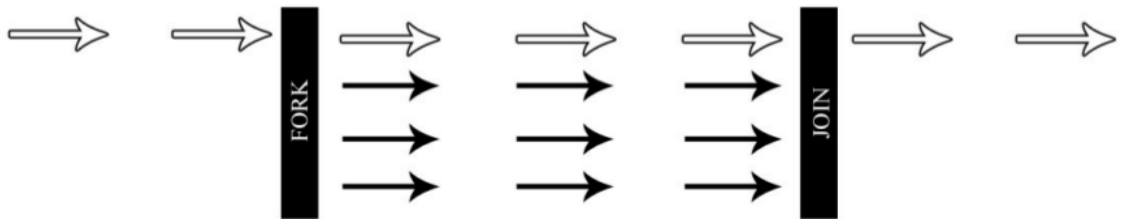
Group 5/8

Note: Next four questions have codes and figures to show the instruction flow (thread) based on fork and join. In each code, a set of any instructions is represented by "…". In each figure, an arrow represents a thread and its instructions ("…"). The threads can be executed in parallel or serial. The master thread is the white arrow.

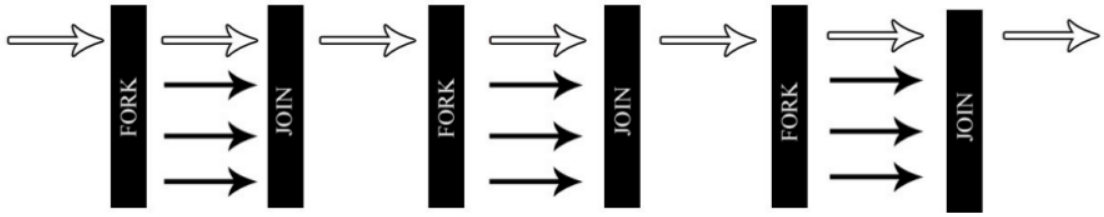
9) The following code has parallel and serial regions. From serial to parallel execution, there is a **fork**. From parallel to serial execution, there is a **join**. Choose the figure that represents the execution of the code:

```
…
for(i=0; i<3; i++){
  #pragma omp parallel
  {
    …
  }
  …
}
```

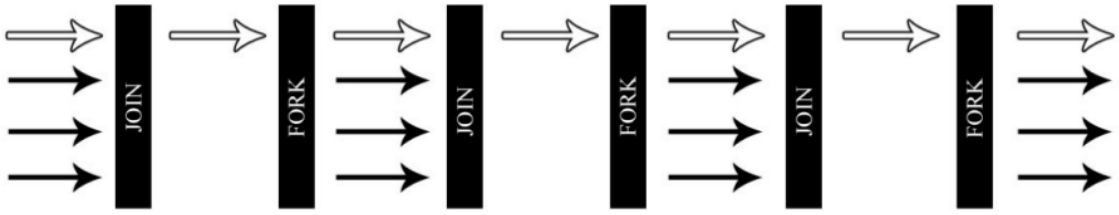
a) ( )



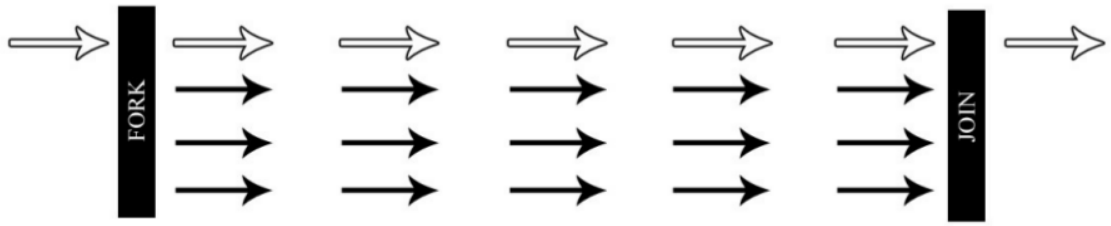
b) ( )



c) ( )



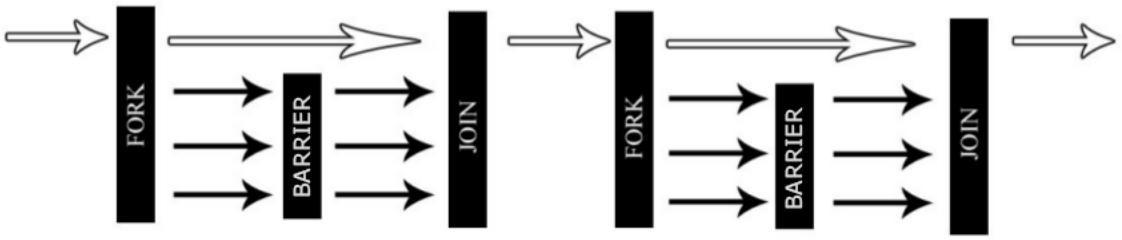
d) ( )



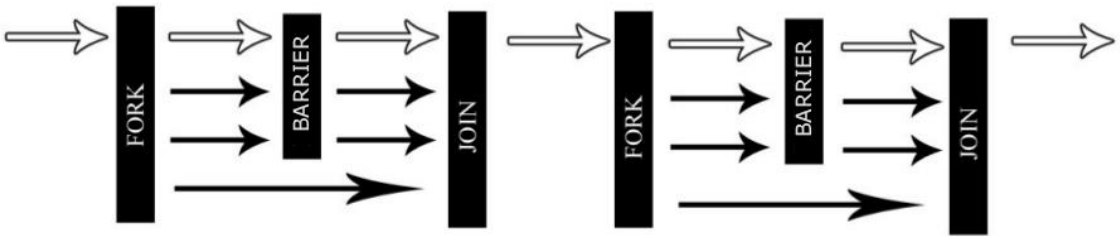
10) The following code uses the **barrier** directive for synchronization. Choose the figure that represents the execution of the code:

```
…
for(i=0; i<2; i++){
  #pragma omp parallel
  {
    …
   #pragma omp barrier
   …
  }
  …
}
```

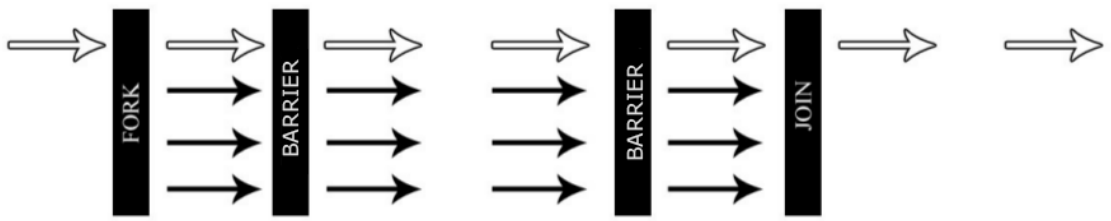
a) ( )



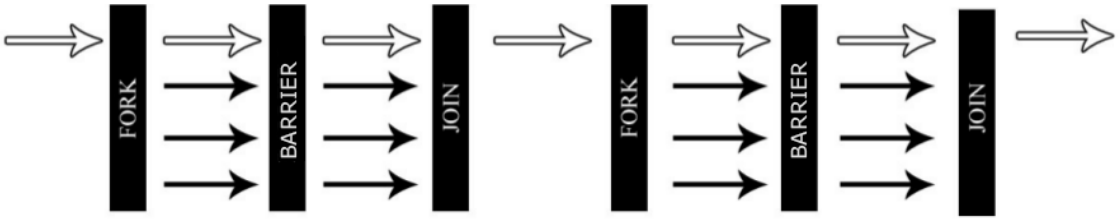
b) ( )



c) ( )



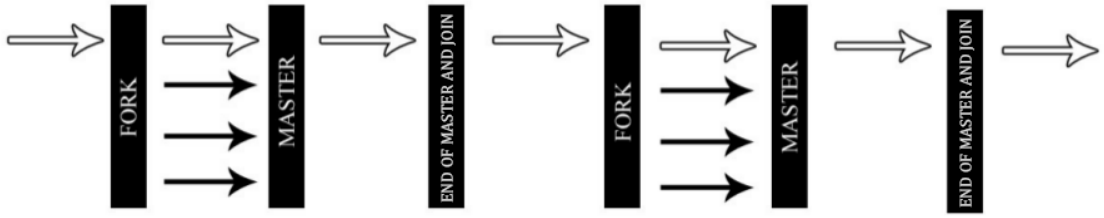
d) ( )



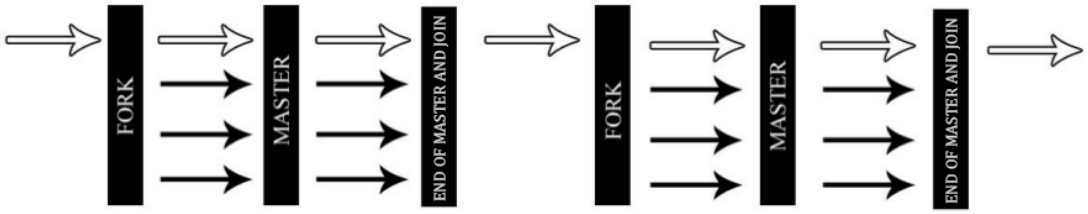
11) The following code uses **master** directive. Choose the figure that represents the execution of the code:

```
…
for(i=0; i<2; i++){
  #pragma omp parallel
  {
    …
   #pragma omp master
   …
  }
  …
}
```

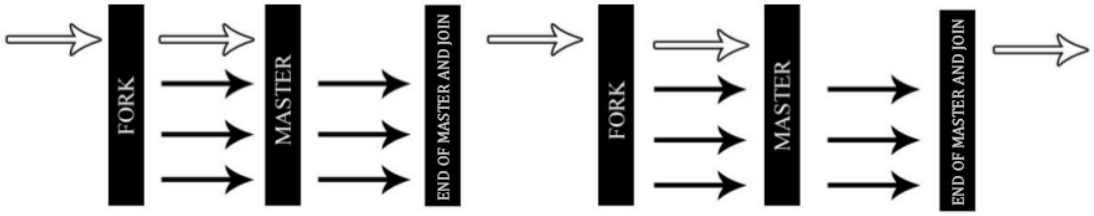
a) ( )



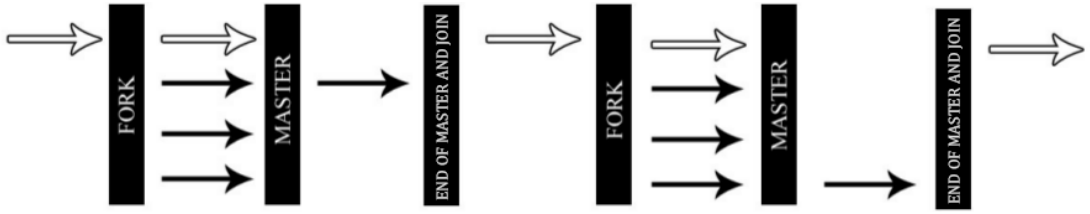
b) ( )



c) ( )



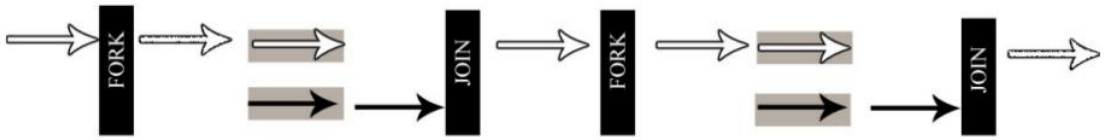
d) ( )



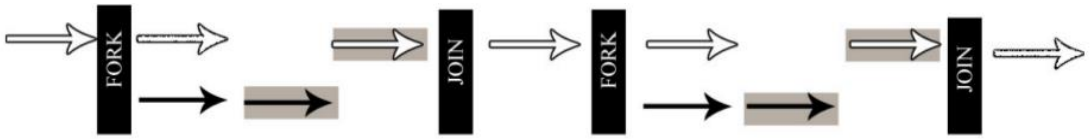
12) The following code uses **critical** directive. The gray background means a computation inside a critical region. Choose the figure that represents the execution of the code:

```
…
for(i=0; i<2; i++){
  #pragma omp parallel num_threads(2)
  {
   #pragma omp critical{
   …
   }
   …
  }
  …
}
```

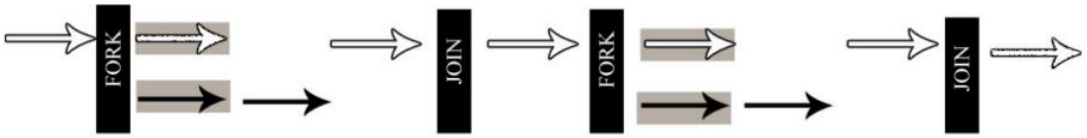
a) ( )



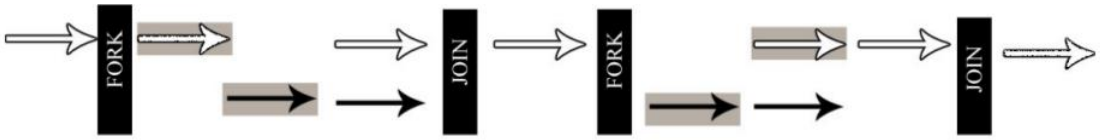
b) ( )



c) ( )



d) ( )



Group 6/8

Note: Next three questions have *printf()* instructions and each answer has a different execution result. Try to visualize the execution flow for each thread.

13) The following code is executed on 4-core processor. Choose the correct execution flow:

```
for(i=0; i<2; i++){
  #pragma omp parallel
  {
    printf("Thread %d\n", omp_get_thread_num());
  }
  printf("####");
}
```

○ Thread 0
Thread 0
####
Thread 1
Thread 1
####

○ Thread 0
Thread 1
####
Thread 2
Thread 3
####

○ Thread 3
Thread 0
Thread 2
Thread 1
####
Thread 0
Thread 1
Thread 2
Thread 3
####

○ Thread 0
Thread 1
Thread 2
Thread 3
####
Thread 0
Thread 1
Thread 2
Thread 2
####

14) The following code is executed on 4-core processor. Choose the correct execution flow:

```
for(i=0; i<2; i++){
  #pragma omp parallel
  {
    #pragma omp master
    printf("Thread %d\n", omp_get_thread_num());
  }
  printf("####");
}
```

○ Thread 3
Thread 0
Thread 2
Thread 1
####
Thread 0
Thread 1
Thread 2
Thread 3
####

○ Thread 0
Thread 0
####
Thread 0
Thread 0
####

○ Thread 0
####
Thread 1
####

○ Thread 0
####
Thread 0
####

15) The following code is executed on 4-core processor. Choose the correct execution flow:

```
for(i=0; i<2; i++){
  #pragma omp parallel
  {
    #pragma omp critical{
    printf("Thread %d\n", omp_get_thread_num());
   }
  }
  printf("####");
}
```

○ Thread 0
####
Thread 0
####

○ Thread 0
####
Thread 1
####

○ Thread 3
Thread 1
Thread 2
Thread 0
####
Thread 0
Thread 3
Thread 2
Thread 1
####

○ Thread 0
Thread 0
Thread 1
Thread 1
####
Thread 2
Thread 2
Thread 3
Thread 3
####

Group 7/8

Note: For the next three questions, each one has one serial code and three parallel versions. Analyze the parallel versions and answer the question.

16) A serial algorithm to find the largest number has a complexity O(n). Its parallel version executed on a n-core processor has the complexity O(1). Look at the following serial and parallel codes and choose the correct answer relative to the three parallel versions:

Serial

```
int arr[n], i, max = 0;
...
for(i=0; i<n; i++){
  if (max<arr[i])
    max=arr[i]; }
```

Parallel I

```
int arr[n], i, max = 0;
...
#pragma omp parallel for shared(max)
for(i=0; i<n; i++){
  if (max<arr[i])
    max=arr[i]; }
```

Parallel II

```
int arr[n], i, max = 0;
...
#pragma omp parallel for
for(i=0; i<n; i++){
  #pragma omp critical{
    if (max<arr[i])
      max=arr[i];
}}
```

Parallel III

```
int arr[n], i, max = 0;
...
#pragma omp parallel {
int max_local = 0;
  #pragma omp for
  for(i=0; i<n; i++){
    if (max<arr[i])
    max=arr[i]; }
  #pragma omp critical{
    if(max<max_local)
      max=max_local;
}}
```

a) ( ) Just the Parallel III and I codes are correct.

b) ( ) Just the Parallel III code is correct.

c) ( ) Just the Parallel II and III codes are correct.

d) ( ) Parallel I, II and III codes are correct.

17) The following serial and parallel codes generate the Pi number. It is important to remark that for parallel versions there is a race condition for the Pi variable. Look at the following serial and parallel codes and choose the correct answer relative to the three parallel versions:

Serial

```
double pi = 0;
int i;
for (i = 0; i < 20000000; i++){
  pi += CONST4/(const4*i + const1);
  pi -= CONST4/(const4*i + const3);
}
```

Parallel I

```
double pi = 0;
int i;
#pragma omp parallel for
for (i = 0; i < 20000000; i++){
  pi += CONST4/(const4*i + const1);
  pi -= CONST4/(const4*i + const3);
}
```

Parallel II	***Parallel III***

`double pi = 0;`
`int i;`
`#pragma omp parallel for`
`for (i = 0; i < 20000000; i++){`
`  #pragma omp critical{`
`  pi += CONST4/(const4*i + const1);`
`  pi -= CONST4/(const4*i + const3);`
`  }`
`}`
	`double pi = 0;`
	`int i;`
	`#pragma omp parallel for reduction(+:pi)`
	`for (i = 0; i < 20000000; i++){`
	`  pi += CONST4/(const4*i + const1);`
	`  pi -= CONST4/(const4*i + const3);`
	`}`

a) ( ) Parallel I, II and III codes are correct.

b) ( ) Just the Parallel I code is correct.

c) ( ) Just the Parallel I and II codes are correct.

d) ( ) Just the Parallel II and III codes are correct.

18) The serial and parallel codes for iterative Fibonacci is presented for this question. Look at the following serial and parallel codes and choose the correct answer relative to the three parallel versions:

Fibonacci numbers definition: the next number of the list is a sum of the two previous numbers. For instance: "0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …"

Serial	***Parallel I***

`long Fibonacci(int n){`
`  int i = 0;`
`  int j = 1;`
`  int k;`
`  for (k = 1; k<n; k++){`
`  int t = i + j;`
`  i = j;`
`  j = t;}`
`  return j;`
`}`
	`long Fibonacci(int n){`
	`  int i = 0;`
	`  int j = 1;`
	`  int k;`
	`  #pragma omp parallel for`
	`  for (k = 1; k<n; k++){`
	`  int t = i + j;`
	`  i = j;`
	`  j = t; }`
	`  return j;`
	`}`

Parallel II	***Parallel III***

`long Fibonacci(int n){`
`  int i = 0;`
`  int j = 1;`
`  int k;`
`  #pragma omp parallel{`
`  #pragma omp for`
`  for (k = 1; k<n; k++){`
`  int t = i + j;`
`  i = j;`
`  j = t; }}`
`  return j;`
`}`
	`long Fibonacci(int n){`
	`  int i = 0;`
	`  int j = 1;`
	`  int k;`
	`  for (k = 1; k<n; k++){`
	`  #pragma omp parallel for{`
	`  int t = i + j;`
	`  i = j;`
	`  j = t; }}`
	`  return j;`
	`}`

a) ( ) Just the Parallel I and III codes are correct.

b) ( ) Just the Parallel I and II codes are correct.

c) ( ) Just the Parallel I code is correct.

d) ( ) Parallel I, II and III codes are wrong. The iterative Fibonacci loop has data dependency.

Group 8/8

Note: The next two questions are relative to high-performance computing (application size, granularity, efficiency, overhead, memory, etc…).

19) A programmer achieved a performance speedup (relative to a serial program) running a parallel version with 4 threads onto 4 cores on a 8-core processor. With 8 threads and 8 cores (same processor) the speedup is lower than the first execution. Choose the correct answer.

a) ( ) Increasing the number of cores, the application efficiency increases, there is an increase of computation time per thread, and a lower final processing time.

b) ( ) Increasing the number of cores, the programmer increases the number of threads, reducing the computation time per thread, increasing the communication time by shared memory, and consequently, a higher final processing time.

c) ( ) Increasing the number of cores, the programmer executed 4 threads on 8 cores, such as one thread per two cores to reduce the workload per core, reducing the final processing time.

d) ( ) Increasing the number of cores, the programmer increases the number of threads, reducing the competition and the communication time by shared memory and consequently increase of final processing time.

20) A programmer achieved a performance speedup (relative to a serial program) running a parallel version with 8 threads onto 8 cores on a 8-core processor. With 4 threads and 4 cores (same processor) the speedup is lower than the first execution. Choose the correct answer.

a) ( ) With 4 cores, the computation time per thread reduces since there is a reduction on the number of instructions to process. Consequently, there is an increase on the final processing time.

b) ( ) With 4 cores, the number of memory accesses reduces the communication time and, consequently, the final processing time.

c) ( ) With 4 cores, there is an increase in the thread size and a higher computation time per thread. Consequently, there is an increase in the final processing time.

d) ( ) With 4 cores, there is an increase on the shared-memory access time and a reduction of computation time per thread. Consequently, there is a reduction on the final processing time.