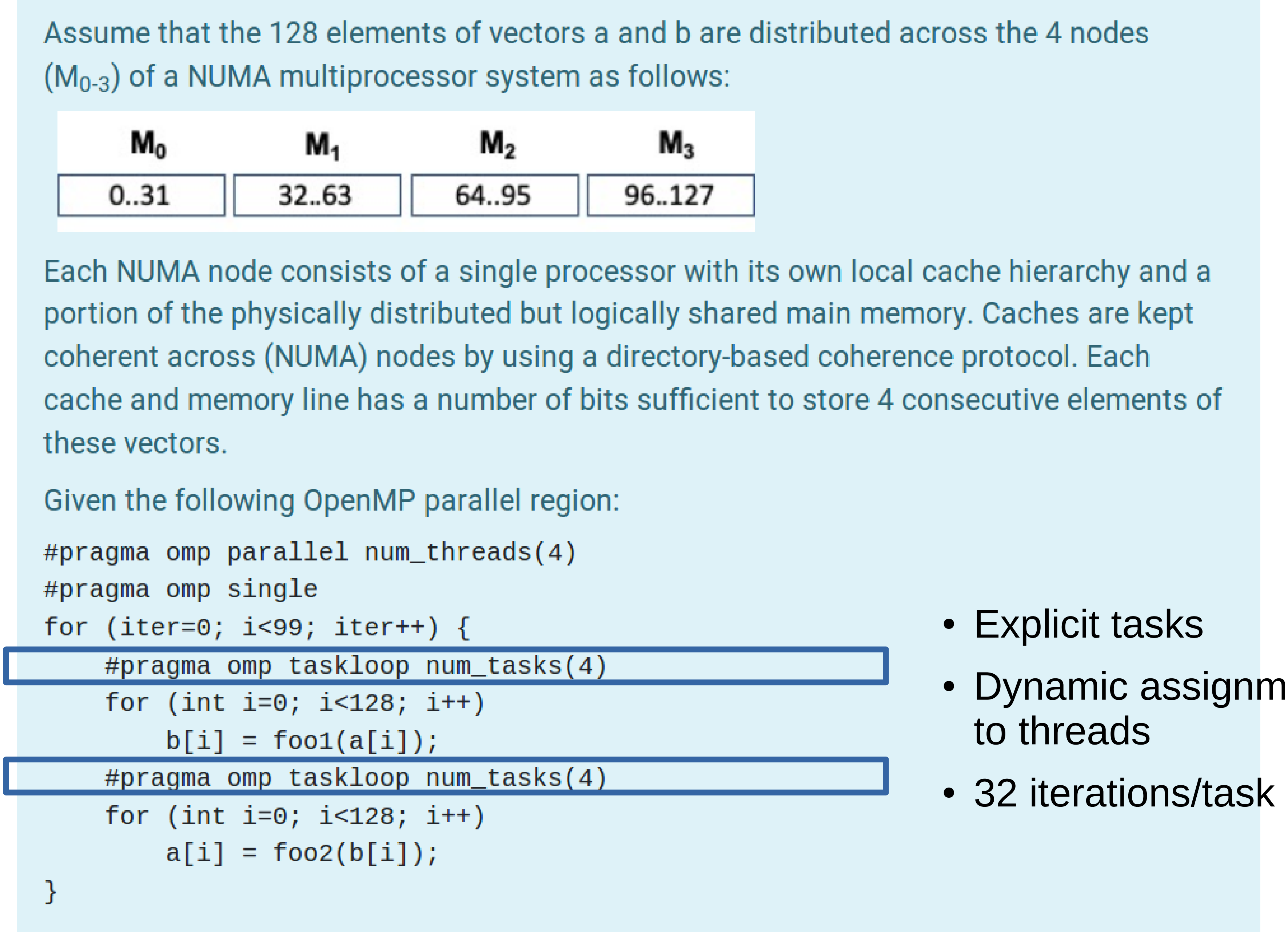
Programming with NUMA systems

Where is data (Wally)?

Is it out there?

The truth is out there!

Statement



Assume that the 128 elements of vectors a and b are distributed across the 4 nodes (M_{0-3}) of a NUMA multiprocessor system as follows:

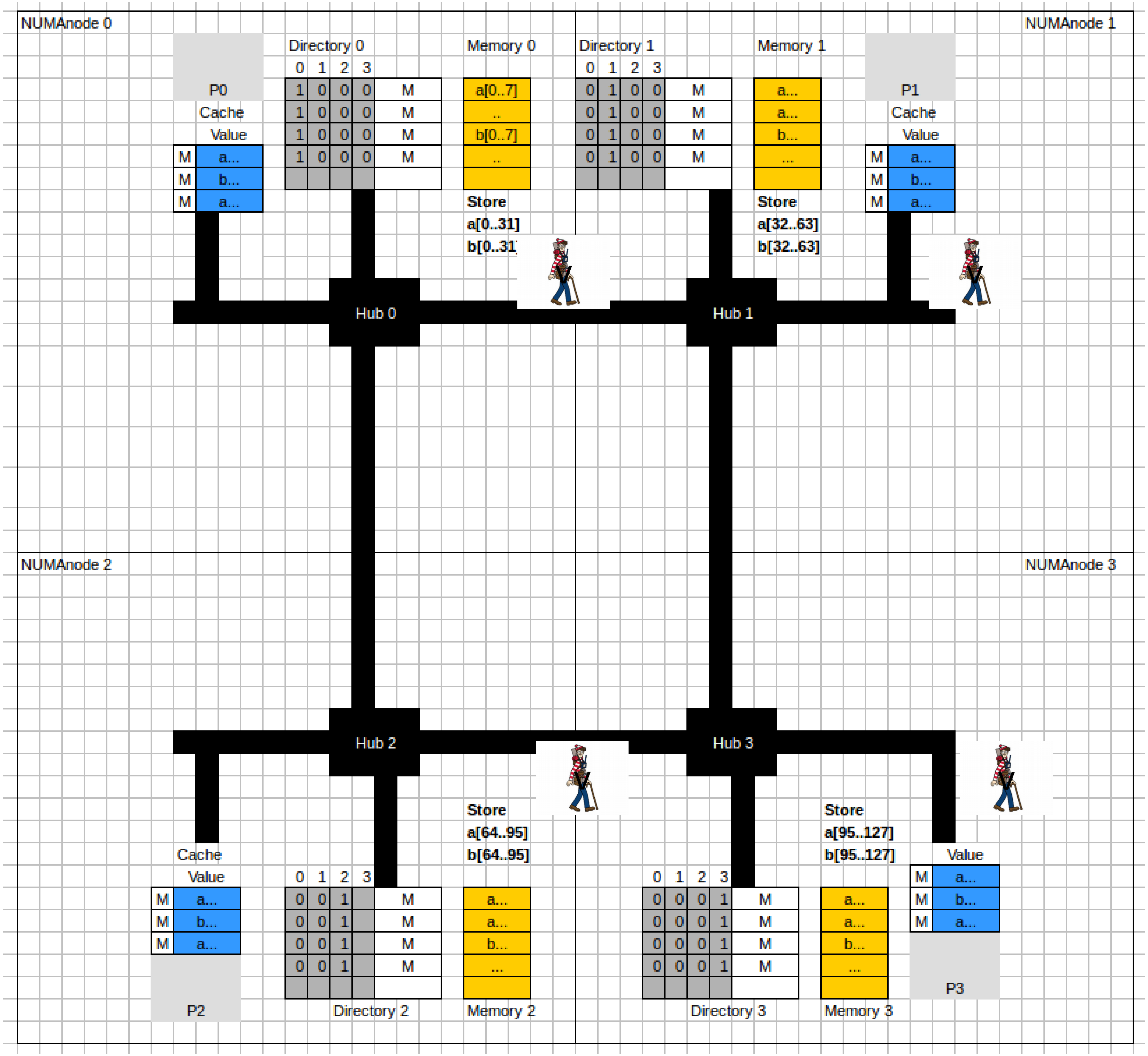
M_0	M_1	M_2	M_3
0..31	32..63	64..95	96..127

Each NUMA node consists of a single processor with its own local cache hierarchy and a portion of the physically distributed but logically shared main memory. Caches are kept coherent across (NUMA) nodes by using a directory-based coherence protocol. Each cache and memory line has a number of bits sufficient to store 4 consecutive elements of these vectors.

Given the following OpenMP parallel region:

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Explicit tasks
- Dynamic assignment to threads
- 32 iterations/task



NUMAnode 0

P0

Cache

	Value
M	a...
M	b...
M	a...

Directory 0

0	1	2	3	
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M

Memory 0
a[0..7]
..
b[0..7]
..

Store
a[0..31]
b[0..31]

Hub 0

NUMAnode 1

Directory 1

0	1	2	3	
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M

Memory 1
a...
a...
b...
...

Store
a[32..63]
b[32..63]

P1

Cache

	Value
M	a...
M	b...
M	a...

Hub 1

NUMAnode 2

Hub 2

Store
a[64..95]
b[64..95]

Cache

	Value
M	a...
M	b...
M	a...

P2

0	1	2	3	
0	0	1		M
0	0	1		M
0	0	1		M
0	0	1		M

Directory 2

a...
a...
b...
...

Memory 2

NUMAnode 3

Hub 3

Store
a[95..127]
b[95..127]

0	1	2	3	
0	0	0	1	M
0	0	0	1	M
0	0	0	1	M
0	0	0	1	M

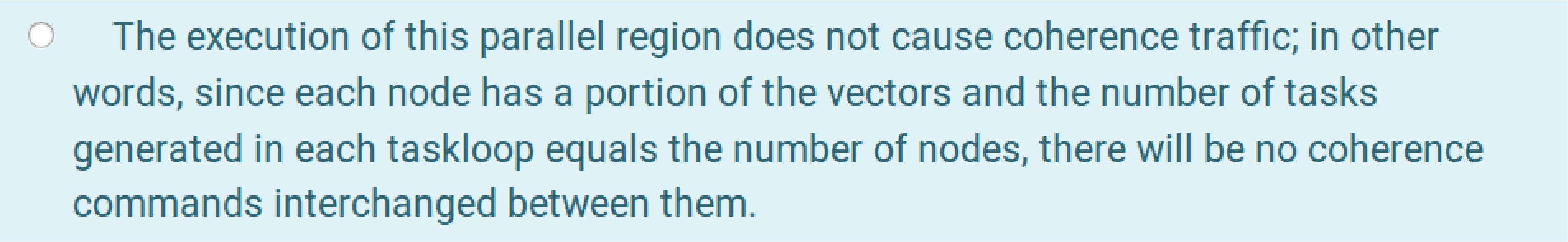
Directory 3

a...
a...
b...
...

Memory 3

	Value
M	a...
M	b...
M	a...

P3



○ The execution of this parallel region does not cause coherence traffic; in other words, since each node has a portion of the vectors and the number of tasks generated in each taskloop equals the number of nodes, there will be no coherence commands interchanged between them.

Let’s analyze:

1) There is not control which data will be accessed by tasks in a taskloop

2) There is not control which data will be accessed by task between taskloops neither iterations

Program Analysis: First Taskloop

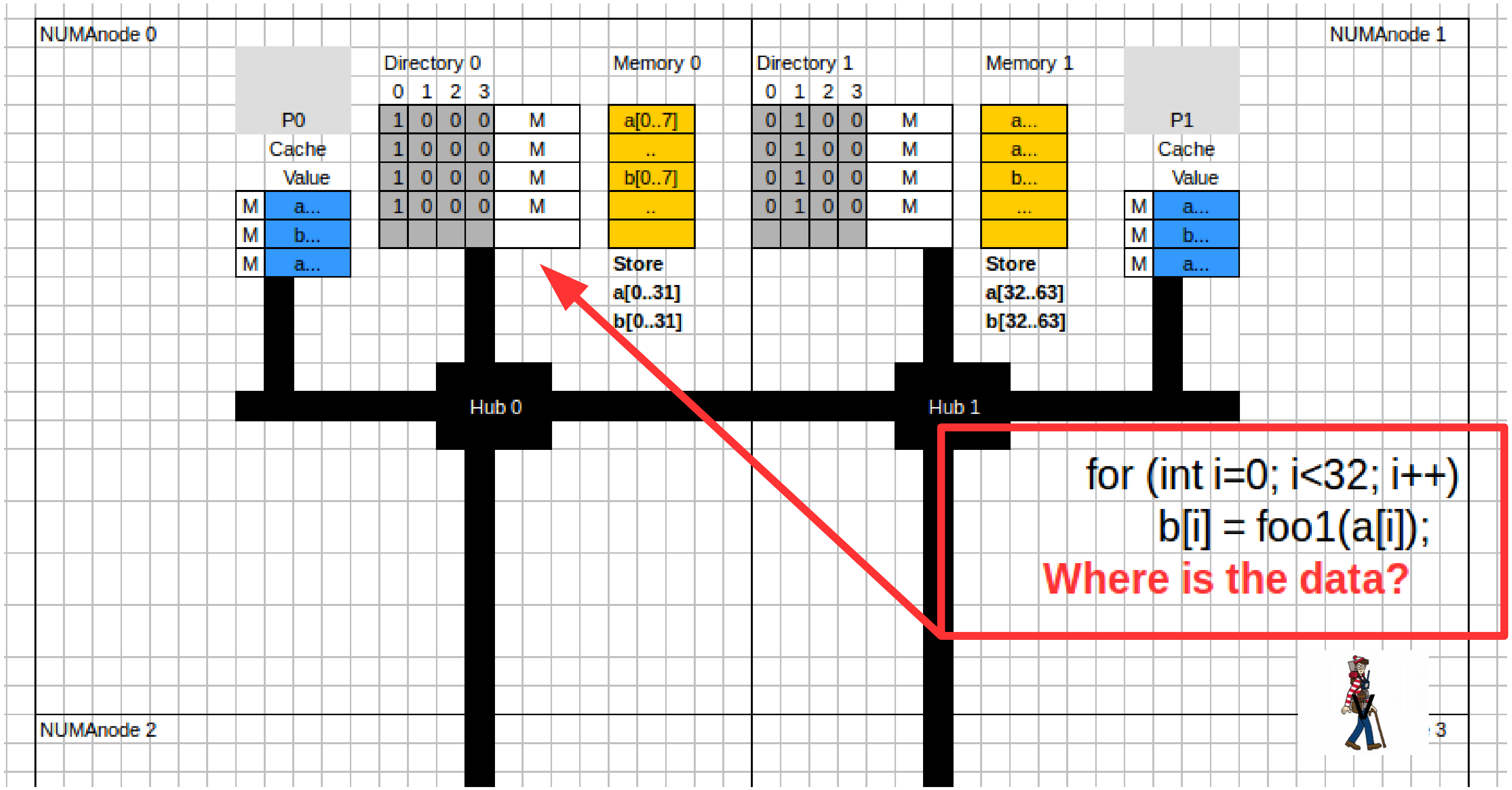
```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- First taskloop creates 4 explicit tasks

Program Analysis: First Taskloop

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- First taskloop creates 4 explicit tasks
- Each task executes 32 iterations
  - **However we don't know which thread will run it**
  - **In addition we don't know where this thread will be executed**



Directory 0

0	1	2	3	
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M

Memory 0

a[0..7]
..
b[0..7]
..

Directory 1

0	1	2	3	
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M

Memory 1

a...
a...
b...
...

P0 Cache

	Value
M	a...
M	b...
M	a...

P1 Cache

	Value
M	a...
M	b...
M	a...

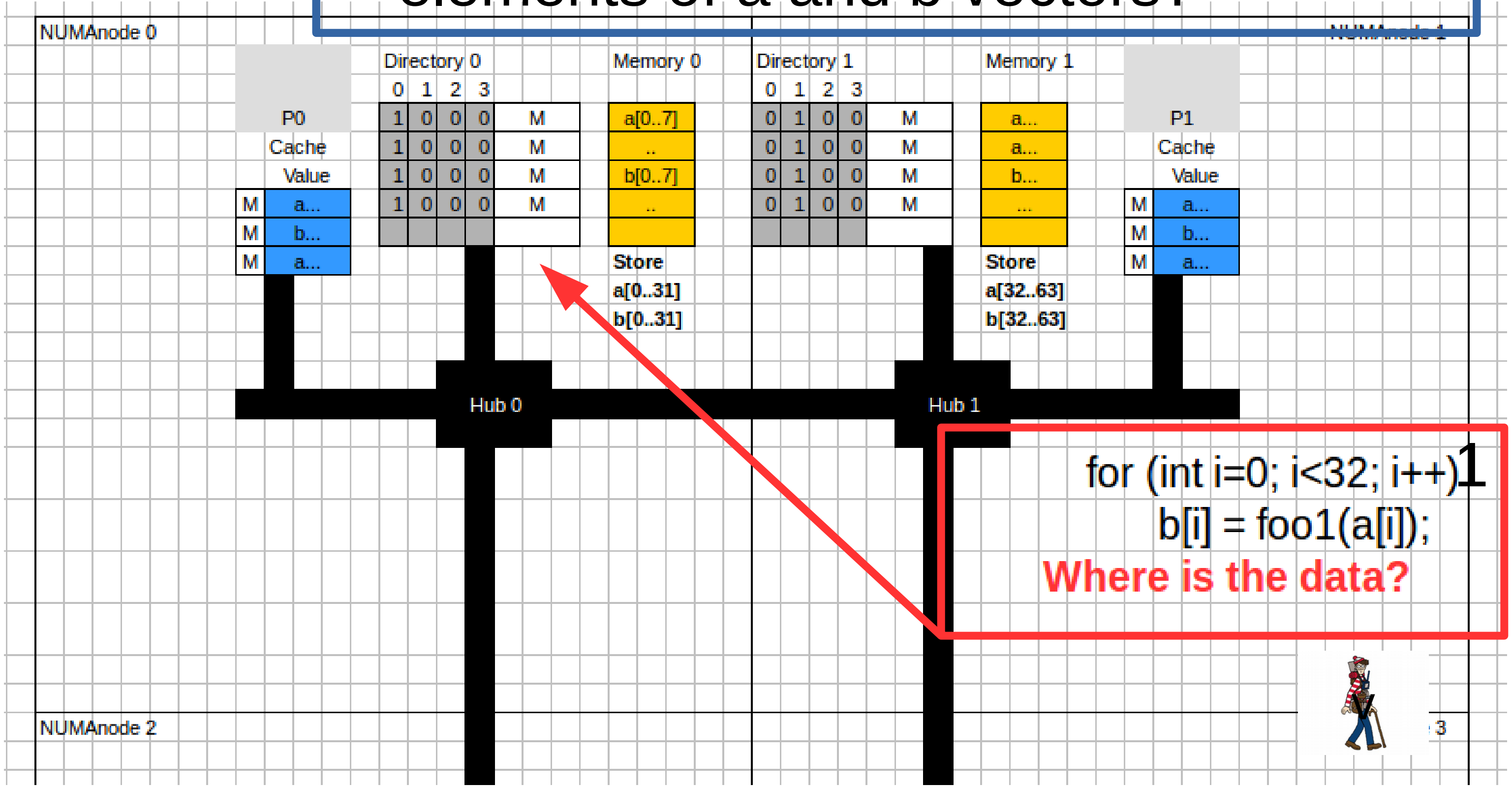
```
for (int i=0; i<32; i++)
    b[i] = foo1(a[i]);
```

Where is the data?

Program Analysis: First Taskloop

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      b[i] = foo1(a[i]);
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      a[i] = foo2(b[i]);
}
```

- First taskloop creates 4 explicit tasks
- Each task has 32 iterations assigned
- Taskloop has an implicit taskwait
- Who perform the access to the first 32 elements of a and b vectors?



for (int i=0; i<32; i++)1
b[i] = foo1(a[i]);
Where is the data?

Program Analysis: First Taskloop

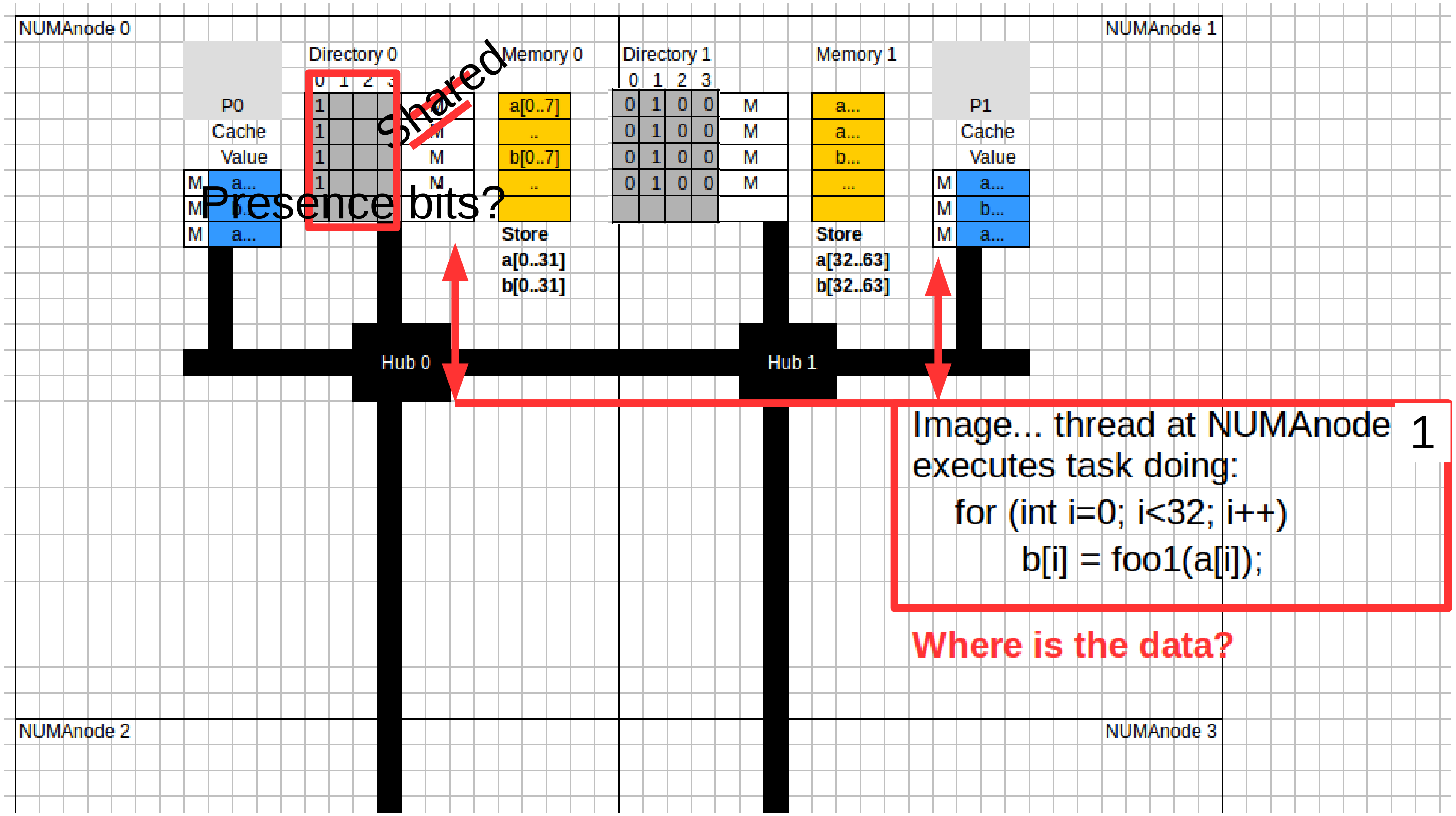
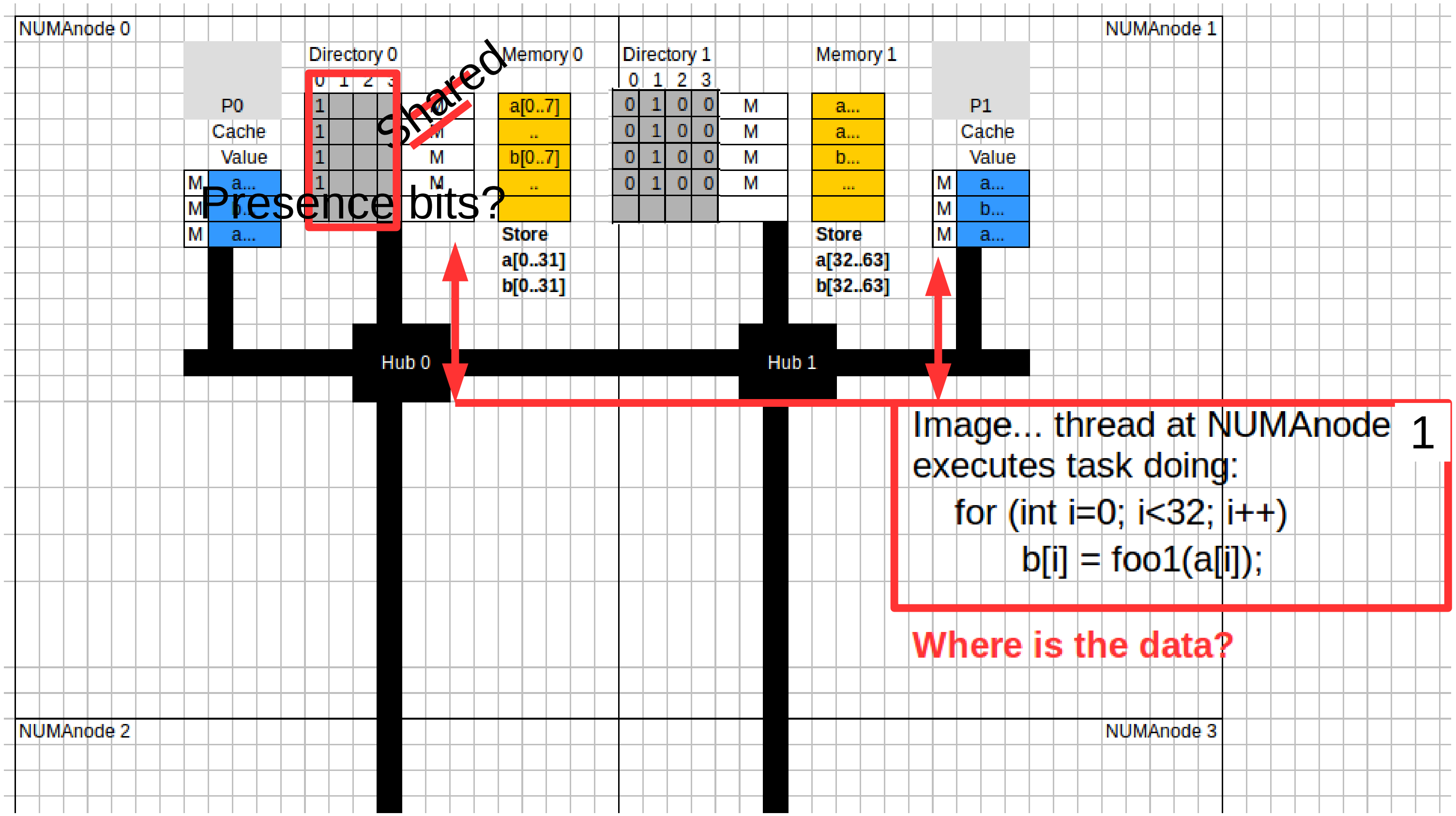


Image... thread at NUMAnode 1 executes task doing:

```
    for (int i=0; i<32; i++)
        b[i] = foo1(a[i]);
```

Where is the data?

Program Analysis: First Taskloop



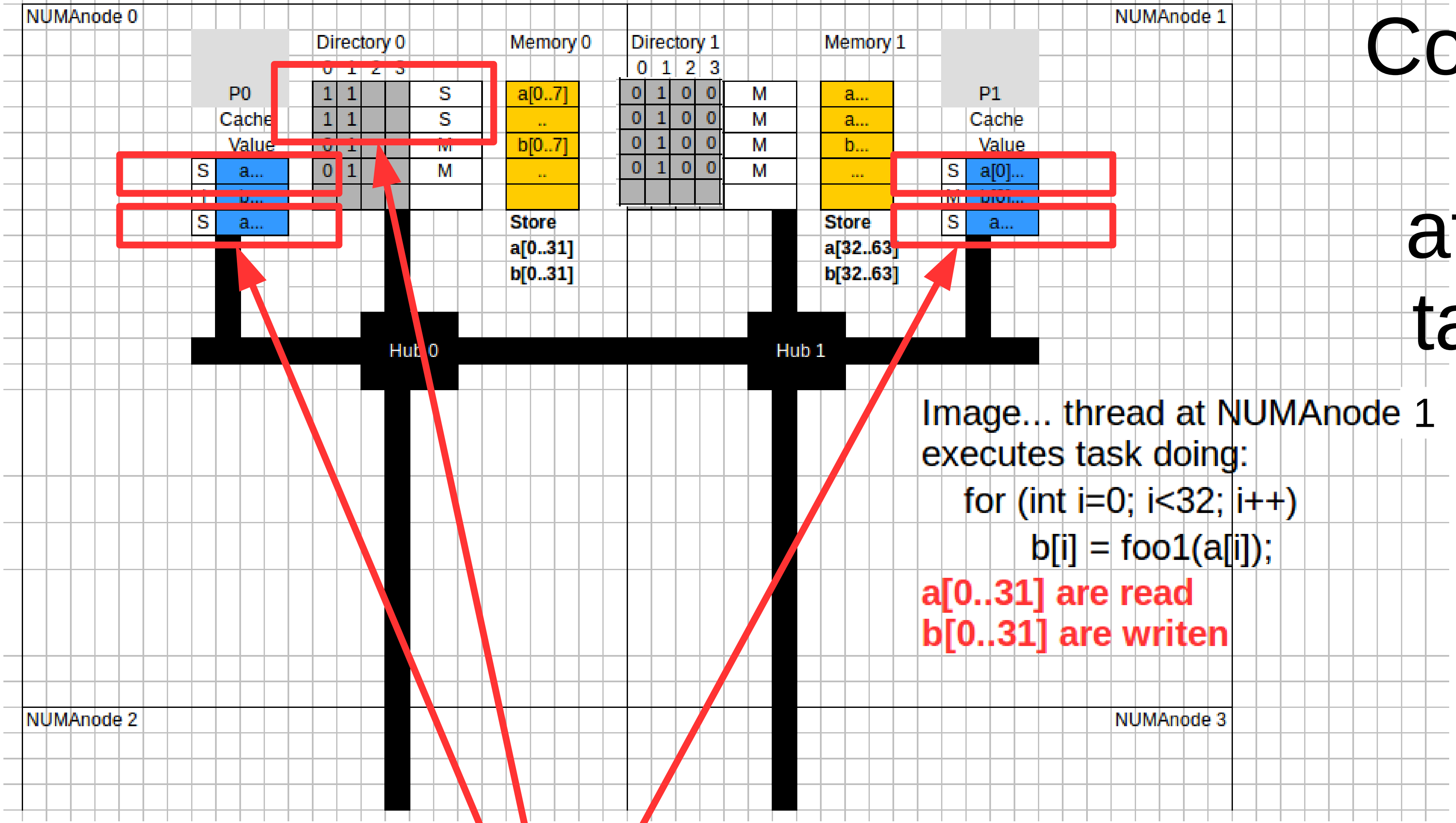
Image… thread at NUMAnode 1 executes task doing:

```
    for (int i=0; i<32; i++)
          b[i] = foo1(a[i]);
```

Where is the data?

Let's see the detail

Coherence data after first taskloop



Directory 0

0	1	2	3	
1	1			S
1	1			S
0	1			M
0	1			M

Directory 1

0	1	2	3	
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M

Image... thread at NUMAnode 1 executes task doing:

```
    for (int i=0; i<32; i++)
          b[i] = foo1(a[i]);
```

a[0..31] are read
b[0..31] are writen

NUMAnode home is NUMAnode 0

First taskloop:

- a[0..31] was read at NUMAnode 1
  - Directory indicates copies at caches of NumaNode0 and 1
  - Cache lines at P1 has shared copies

Coherence data after first taskloop

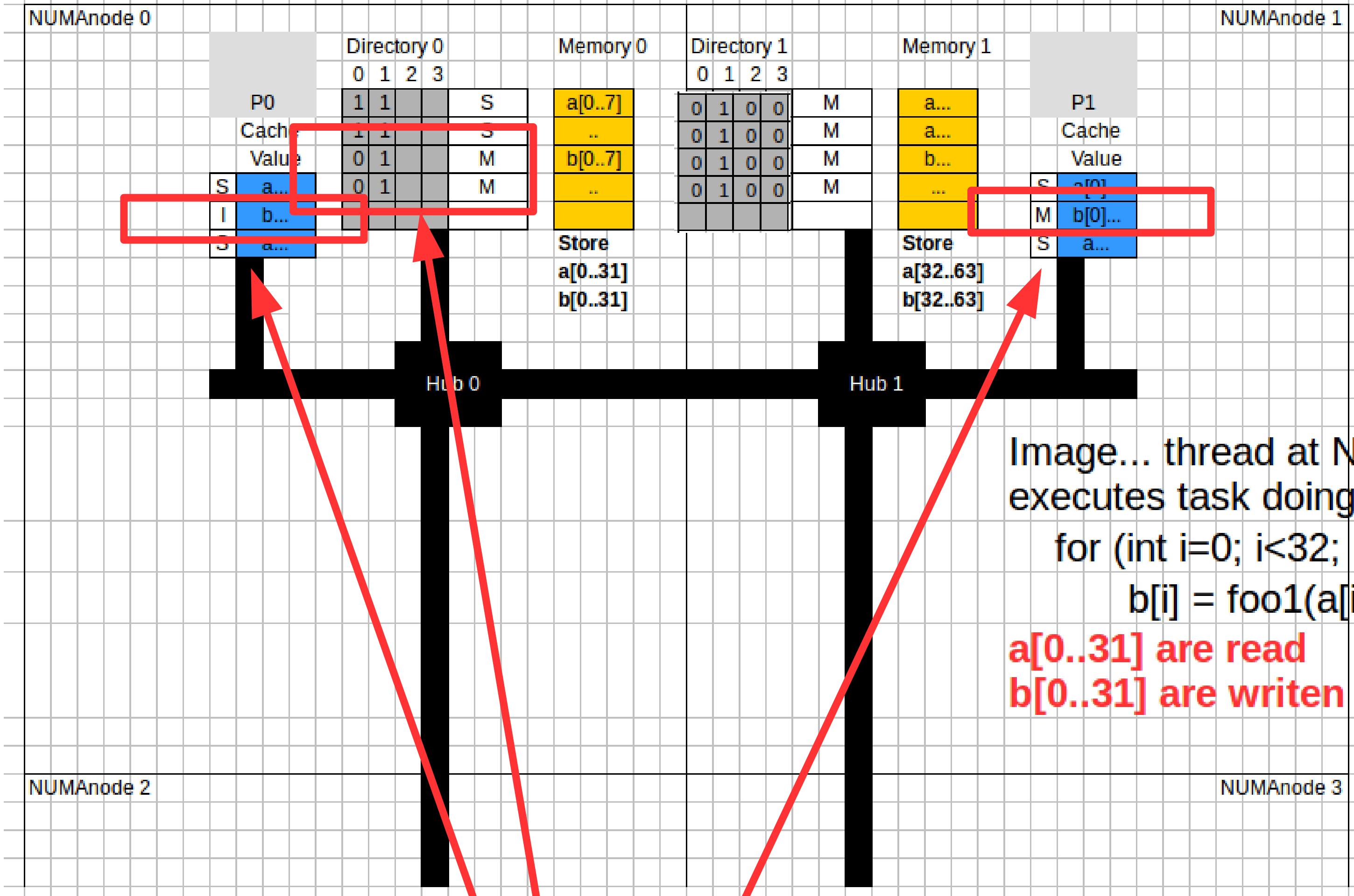


Image... thread at NUMAnode 1 executes task doing:

```
    for (int i=0; i<32; i++)
        b[i] = foo1(a[i]);
```

a[0..31] are read
b[0..31] are writen

NUMAnode home is NUMAnode 0

- a[0..31] was read at NUMAnode 1
  - Directory indicates copies at caches of NumaNode0 and 1
  - Cache lines at P1 has shared copies
- b[0..31] was modified at NUMAnode 1
  - Directory indicates modified copies at NumaNode 1
  - Cache lines at P1 has modified copies

Program Analysis: Second Taskloop

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Second Taskloop creates 4 tasks
- Each task has 32 iterations assigned
- Are the same iterations (portion of vectors a and b) assigned to the same thread?

Program Analysis: Second Taskloop

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Second Taskloop creates 4 tasks
- Each task has 32 iterations assigned
- Are the same iterations assigned to the same thread? NOOO.....

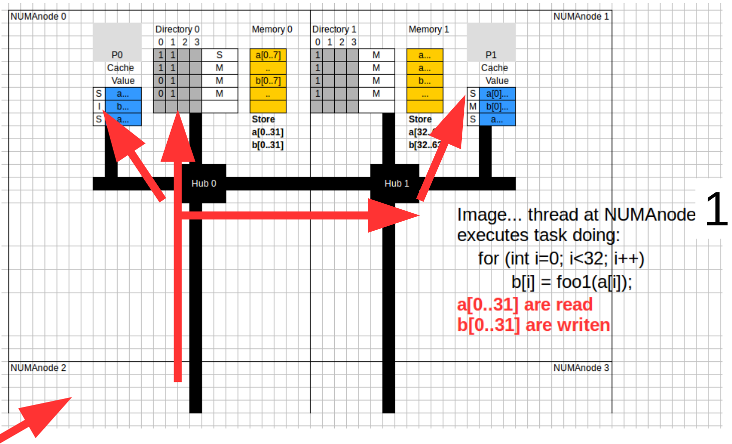


Image... thread at NUMAnode 1 executes task doing:

```
for (int i=0; i<32; i++)
    b[i] = foo1(a[i]);
```

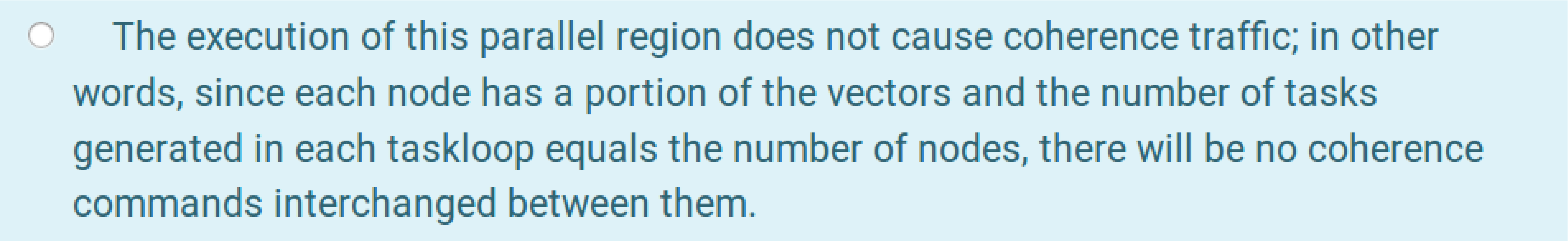
a[0..31] are read
b[0..31] are writen

Initially, after first taskloop:

- NUMAnode home is NUMAnode 0
- a[0..31] was read at NUMAnode 1
  - Memory lines have copies at caches of P0 and P1
- b[0..31] was modified at NUMAnode 1
  - Memory lines only have modified copies at P1

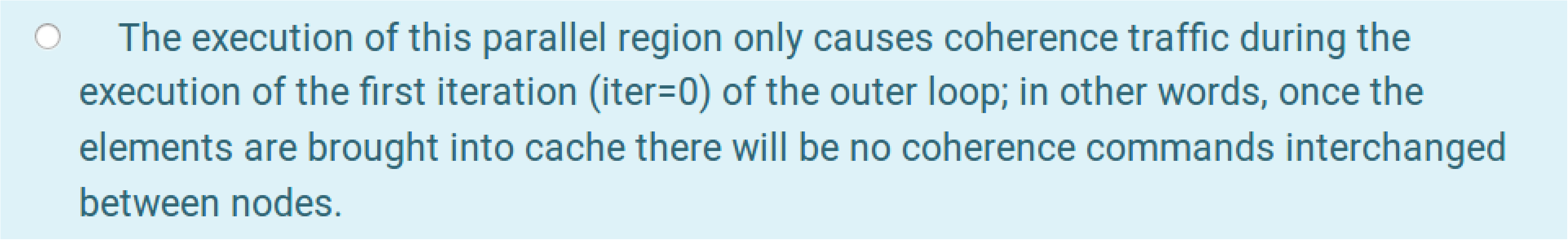
Assume now that a thread running at NUMAnode 2 or 3 does task accessing a[0..31] and b[0..31] ....

- Invalidations have to be done
- Presence bits has to be updated



○ The execution of this parallel region does not cause coherence traffic; in other words, since each node has a portion of the vectors and the number of tasks generated in each taskloop equals the number of nodes, there will be no coherence commands interchanged between them.

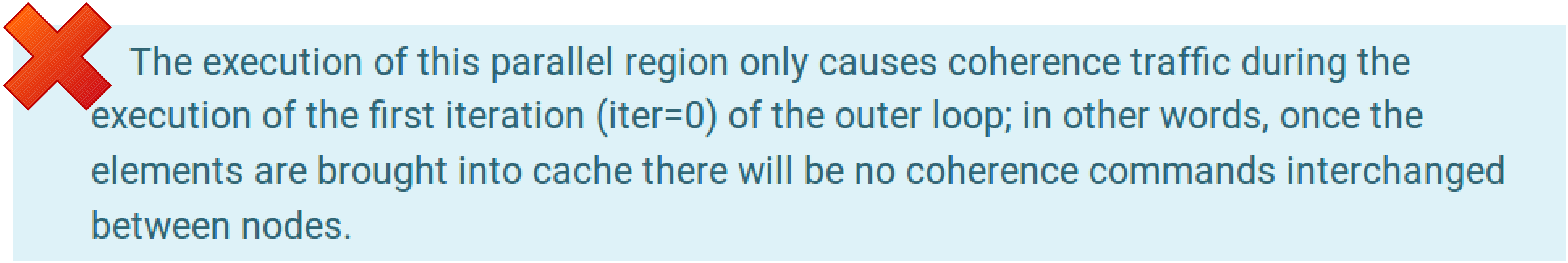
✗ The execution of this parallel region does not cause coherence traffic; in other words, since each node has a portion of the vectors and the number of tasks generated in each taskloop equals the number of nodes, there will be no coherence commands interchanged between them.



○ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      b[i] = foo1(a[i]);
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      a[i] = foo2(b[i]);
}
```

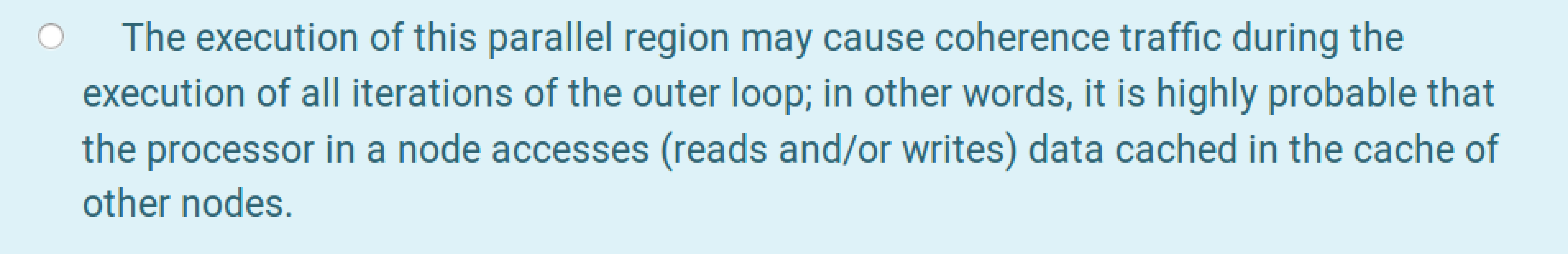
- Taskloops will create new tasks
- Tasks are dynamically assigned



The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

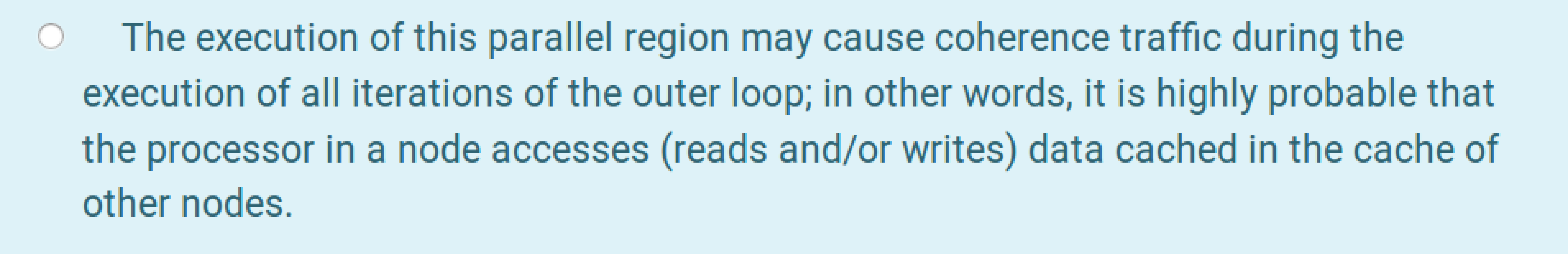
```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      b[i] = foo1(a[i]);
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      a[i] = foo2(b[i]);
}
```

- Taskloops will create new tasks
- Tasks are dynamically assigned
- From one iteration to another iteration there may be different assignment
  - Same problem as before



○ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

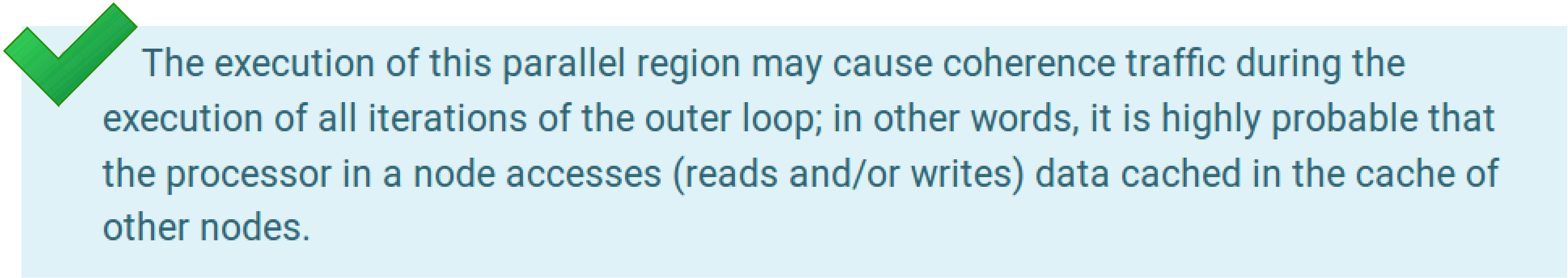
```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      b[i] = foo1(a[i]);
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      a[i] = foo2(b[i]);
}
```



○ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      b[i] = foo1(a[i]);
   #pragma omp taskloop num_tasks(4)
   for (int i=0; i<128; i++)
      a[i] = foo2(b[i]);
}
```

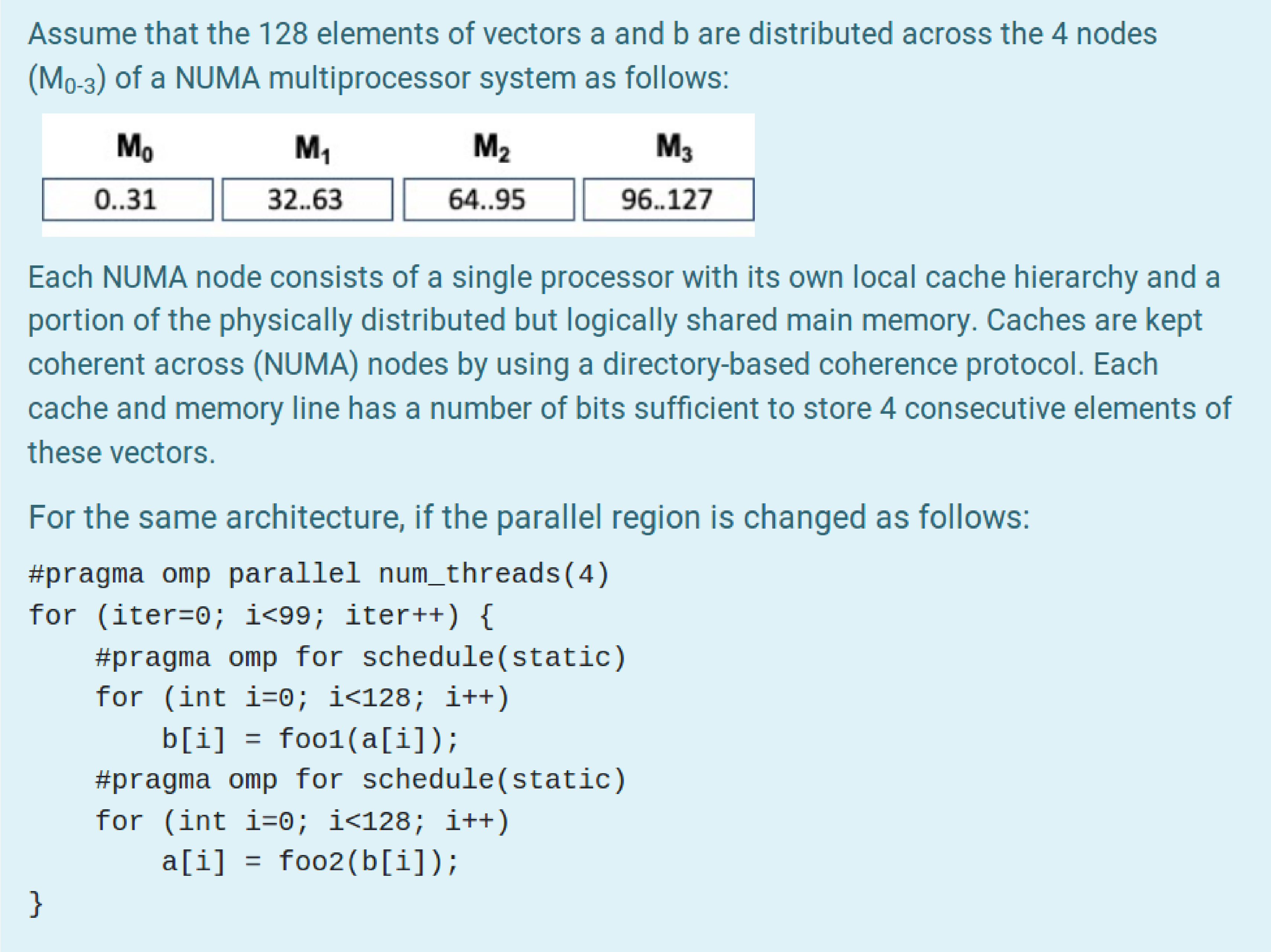
- There may be plenty of coherence traffic due to accesses to data cached in the cache of other nodes


The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- There may be plenty of coherence traffic due to accesses to data cached in the cache of other nodes

Statement



Assume that the 128 elements of vectors a and b are distributed across the 4 nodes (M$_{0-3}$) of a NUMA multiprocessor system as follows:

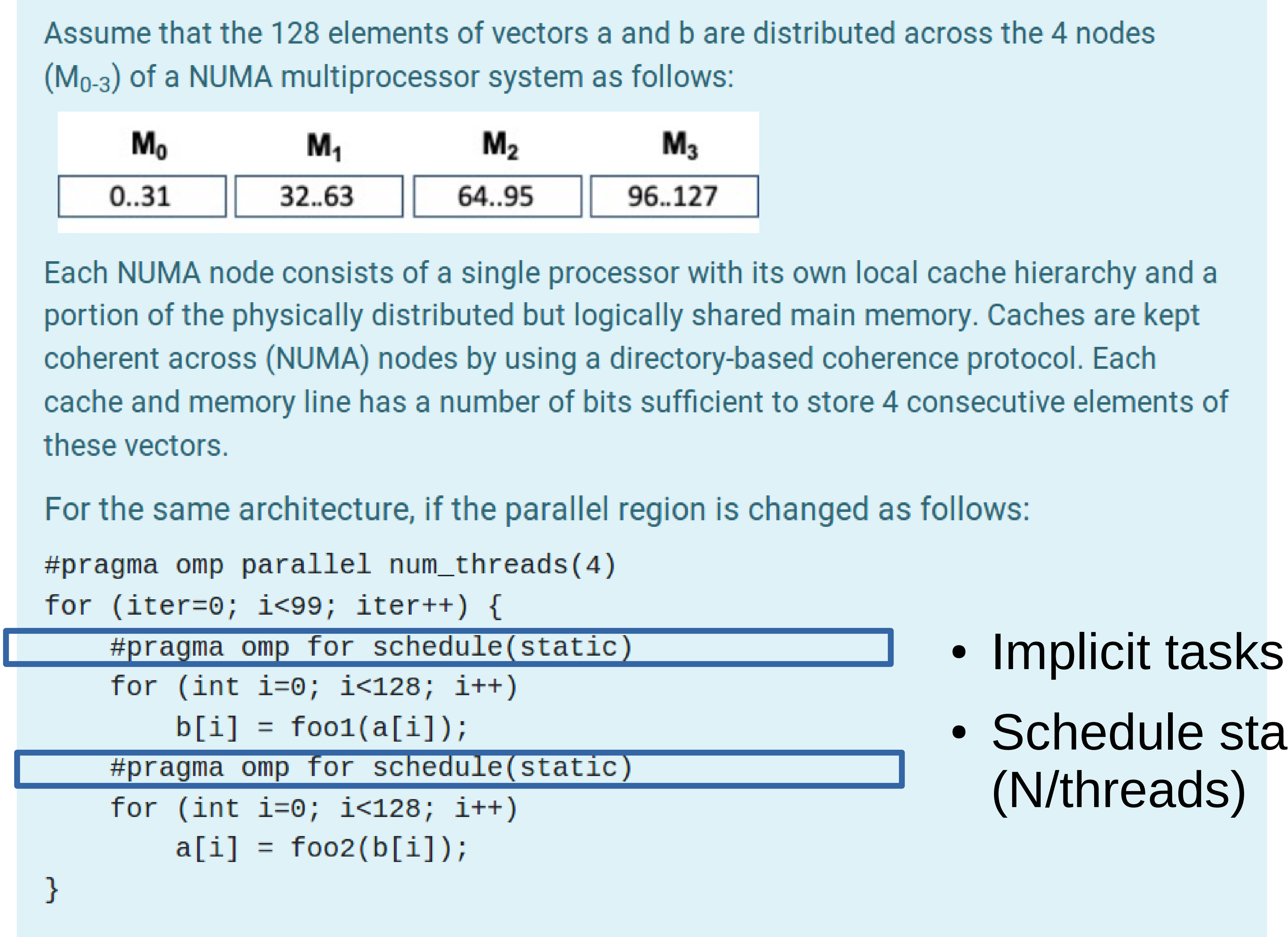
M$_0$	M$_1$	M$_2$	M$_3$
0..31	32..63	64..95	96..127

Each NUMA node consists of a single processor with its own local cache hierarchy and a portion of the physically distributed but logically shared main memory. Caches are kept coherent across (NUMA) nodes by using a directory-based coherence protocol. Each cache and memory line has a number of bits sufficient to store 4 consecutive elements of these vectors.

For the same architecture, if the parallel region is changed as follows:

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

Statement



Assume that the 128 elements of vectors a and b are distributed across the 4 nodes ($M_{0\text{-}3}$) of a NUMA multiprocessor system as follows:

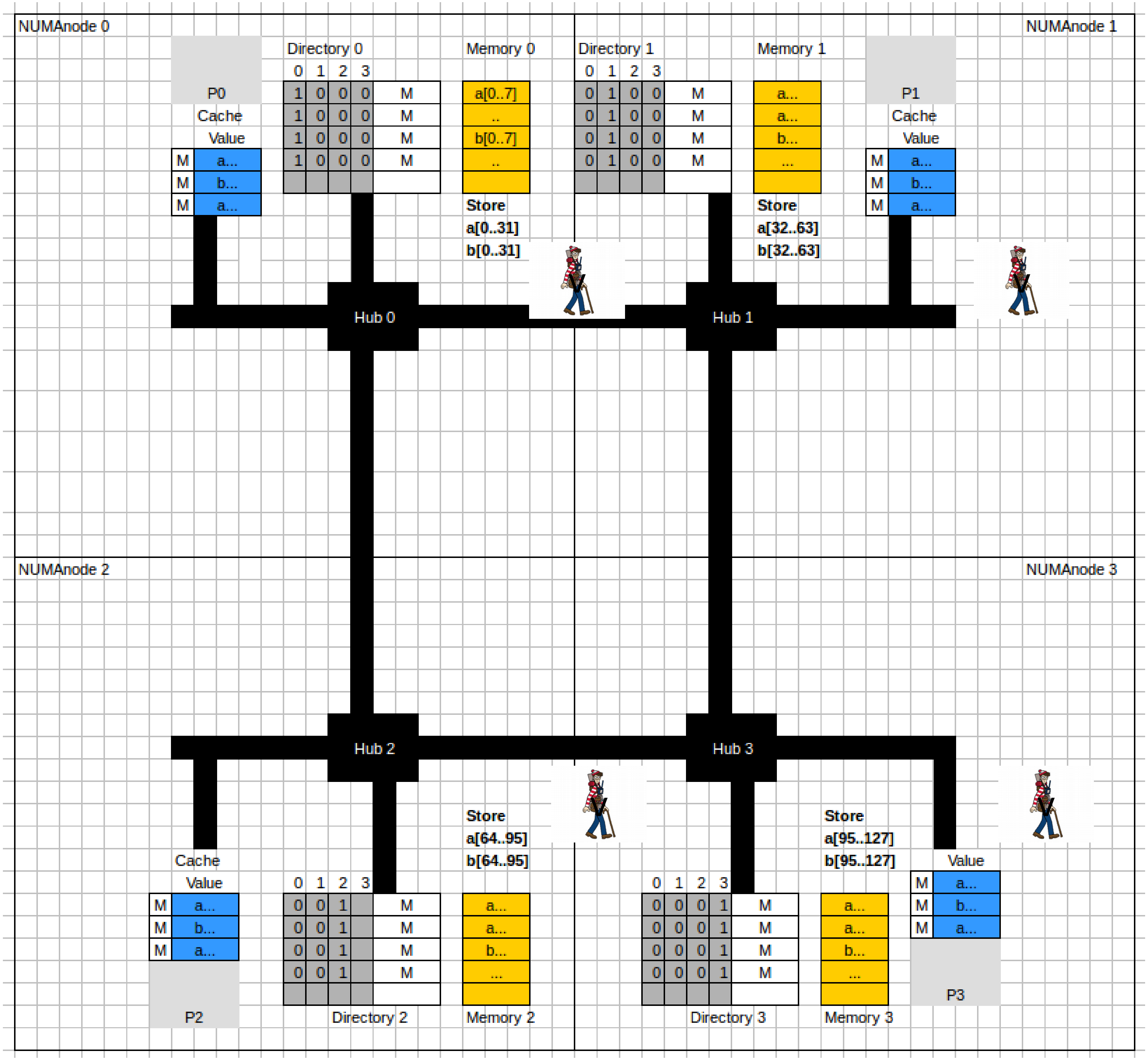
M_0	M_1	M_2	M_3
0..31	32..63	64..95	96..127

Each NUMA node consists of a single processor with its own local cache hierarchy and a portion of the physically distributed but logically shared main memory. Caches are kept coherent across (NUMA) nodes by using a directory-based coherence protocol. Each cache and memory line has a number of bits sufficient to store 4 consecutive elements of these vectors.

For the same architecture, if the parallel region is changed as follows:

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Implicit tasks
- Schedule static (N/threads)



NUMAnode 0

P0

Cache

	Value
M	a...
M	b...
M	a...

Directory 0

0	1	2	3	
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M

Memory 0

a[0..7]
..
b[0..7]
..

Store
a[0..31]
b[0..31]

Hub 0

NUMAnode 1

Directory 1

0	1	2	3	
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M

Memory 1

a...
a...
b...
...

Store
a[32..63]
b[32..63]

P1

Cache

	Value
M	a...
M	b...
M	a...

Hub 1

NUMAnode 2

Hub 2

Cache

	Value
M	a...
M	b...
M	a...

P2

0	1	2	3	
0	0	1		M
0	0	1		M
0	0	1		M
0	0	1		M

Directory 2

Store
a[64..95]
b[64..95]

a...
a...
b...
...

Memory 2

NUMAnode 3

Hub 3

0	1	2	3	
0	0	0	1	M
0	0	0	1	M
0	0	0	1	M
0	0	0	1	M

Directory 3

Store
a[95..127]
b[95..127]

a...
a...
b...
...

Memory 3

	Value
M	a...
M	b...
M	a...

P3

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Schedule static assignming (N/4) to each thread
  - Thread 0: b[0..31],a[0..31]
  - Thread 1: b[32..63],a[32..63]
  - Thread 2: b[64..95],a[64..95]
  - Thread 3: b[96..127],a[96..127]

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

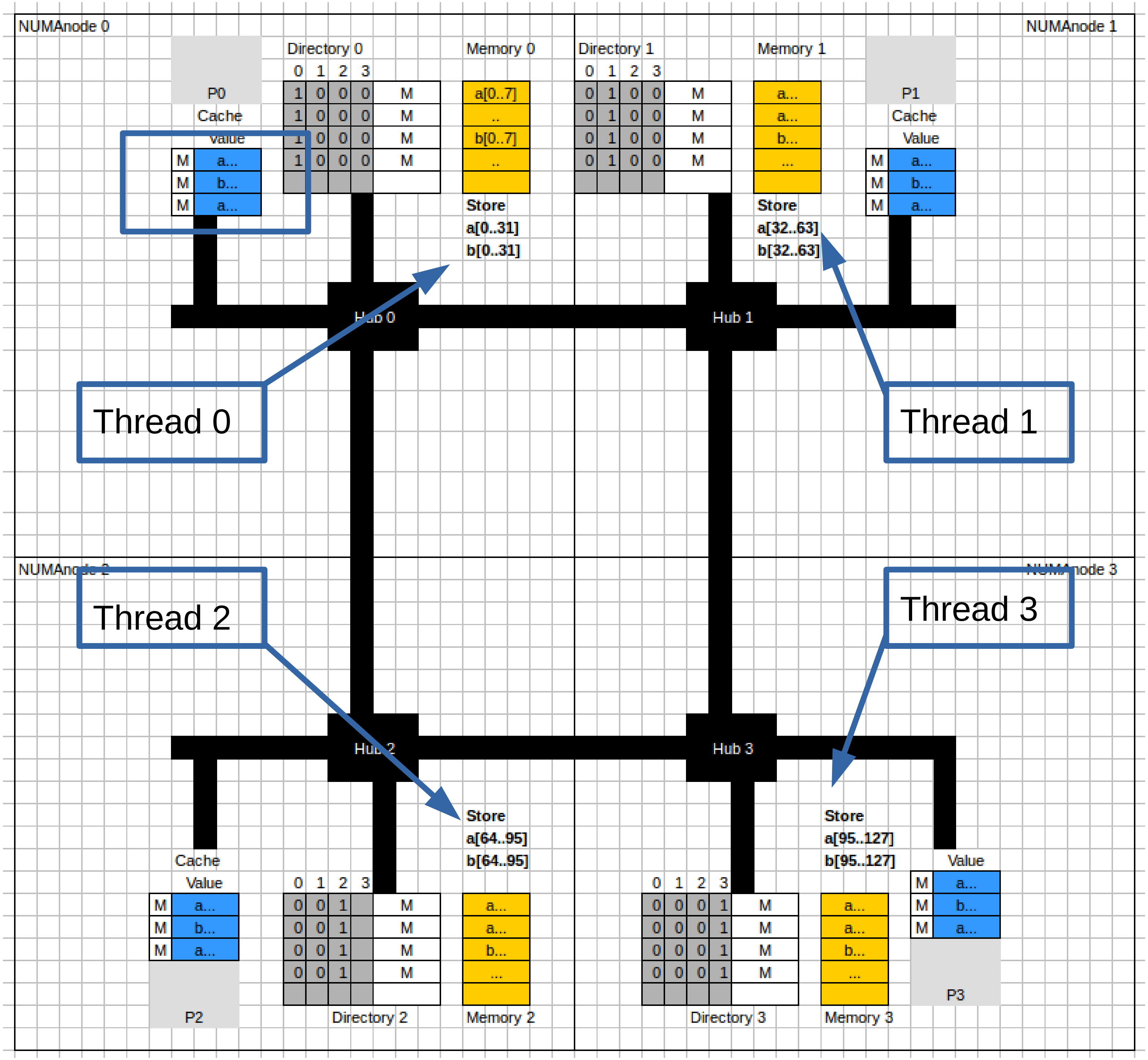
- Schedule static assignming (N/4) to each thread
- Threre is an implicit barrier at the end of each omp for
- Each omp for has the same schedule

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Schedule static assignming (N/4) to each thread
- Threre is an implicit barrier at the end of each omp for
- **Assume thread i runs in NumaNode/Processor i and..**
  - **Thread 0: b[0..31],a[0..31]**
  - **Thread 1: b[32..63],a[32..63]**
  - **Thread 2: b[64..95],a[64..95]**
  - **Thread 3: b[96..127],a[96..127]**

Where is the data that is accessed by each thread?



NUMAnode 0

P0

Cache

Value

M	a...
M	b...
M	a...

Directory 0

0	1	2	3	
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M
1	0	0	0	M

Memory 0

a[0..7]
..
b[0..7]
..

Store
a[0..31]
b[0..31]

Hub 0

NUMAnode 1

Directory 1

0	1	2	3	
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M
0	1	0	0	M

Memory 1

a...
a...
b...
...

P1

Cache

Value

M	a...
M	b...
M	a...

Store
a[32..63]
b[32..63]

Hub 1

Thread 0

Thread 1

NUMAnode 2

Thread 2

Hub 2

Store
a[64..95]
b[64..95]

Cache

Value

M	a...
M	b...
M	a...

P2

0	1	2	3	
0	0	1		M
0	0	1		M
0	0	1		M
0	0	1		M

Directory 2

a...
a...
b...
...

Memory 2

NUMAnode 3

Thread 3

Hub 3

Store
a[95..127]
b[95..127]

0	1	2	3	
0	0	0	1	M
0	0	0	1	M
0	0	0	1	M
0	0	0	1	M

Directory 3

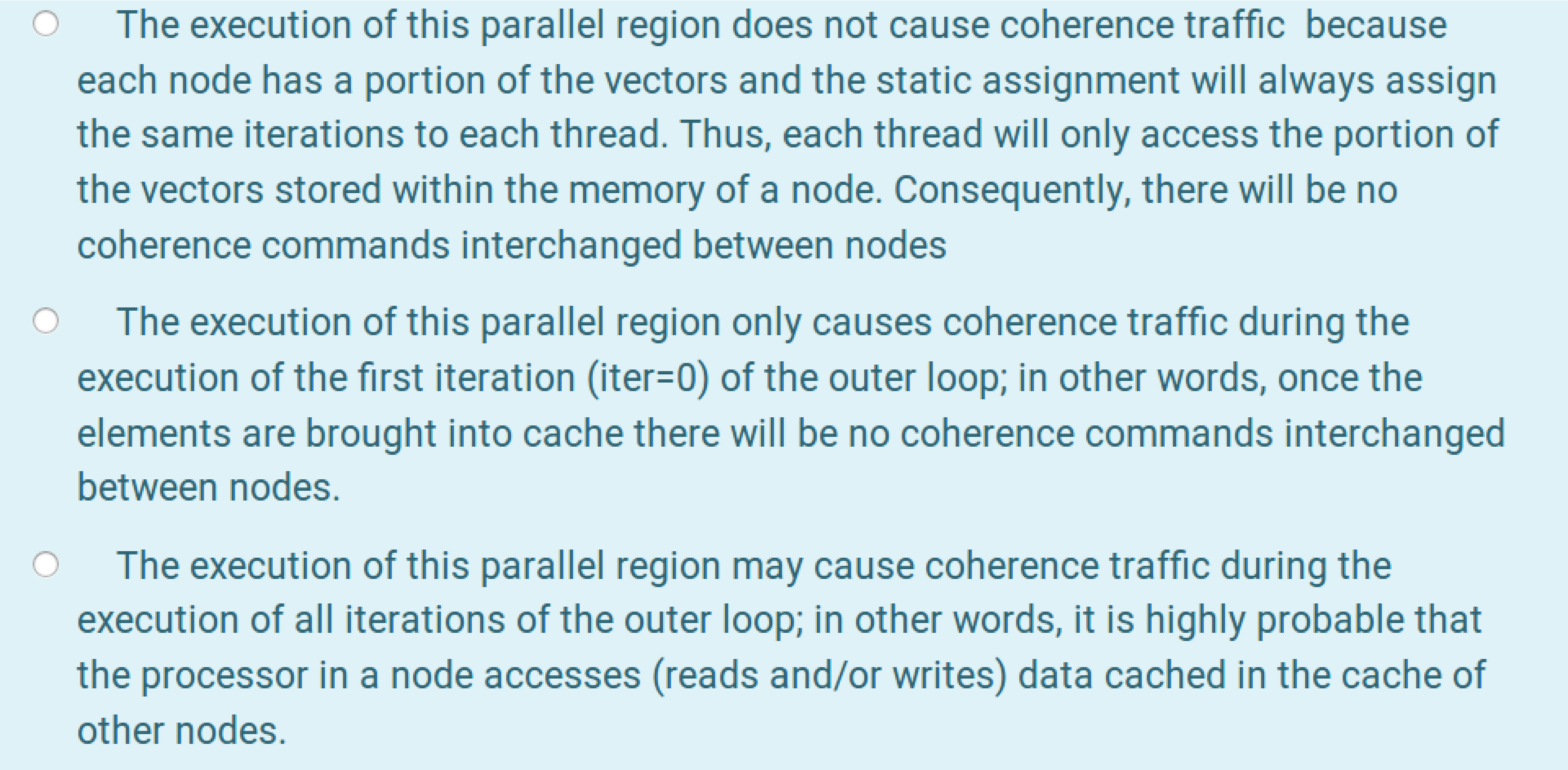
a...
a...
b...
...

Memory 3

Value

M	a...
M	b...
M	a...

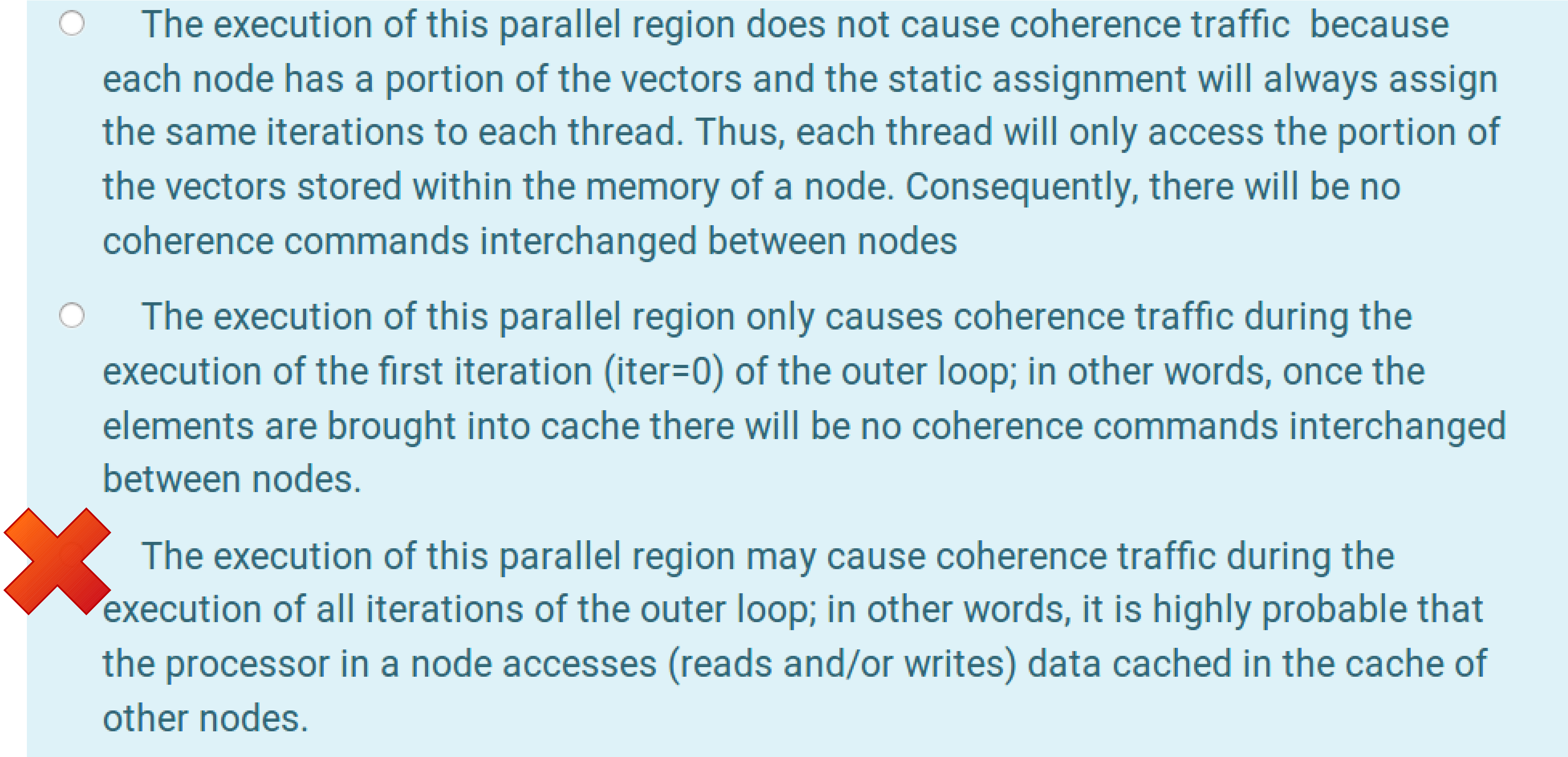
P3



○ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

○ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

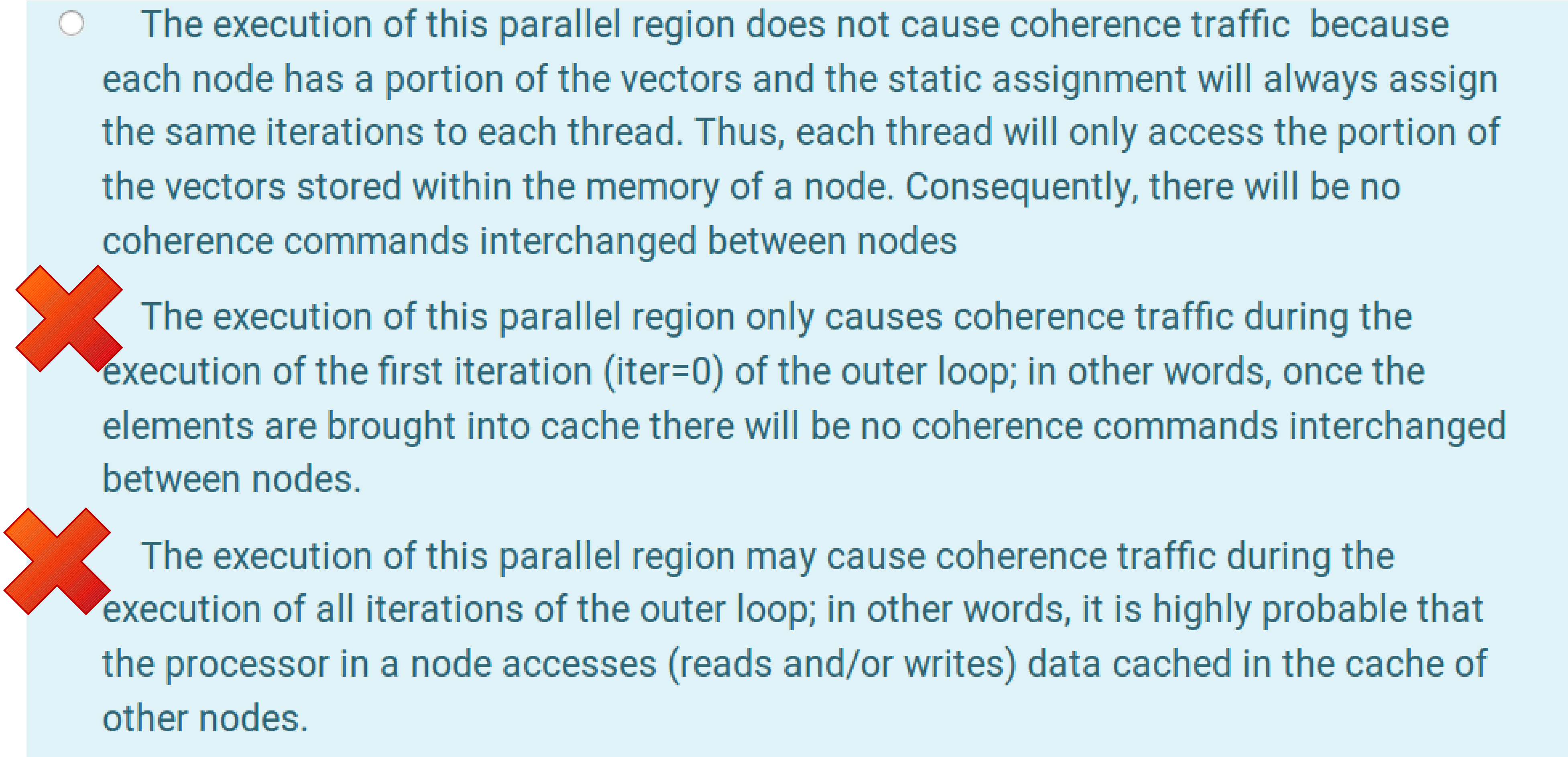
○ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.



○ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

○ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

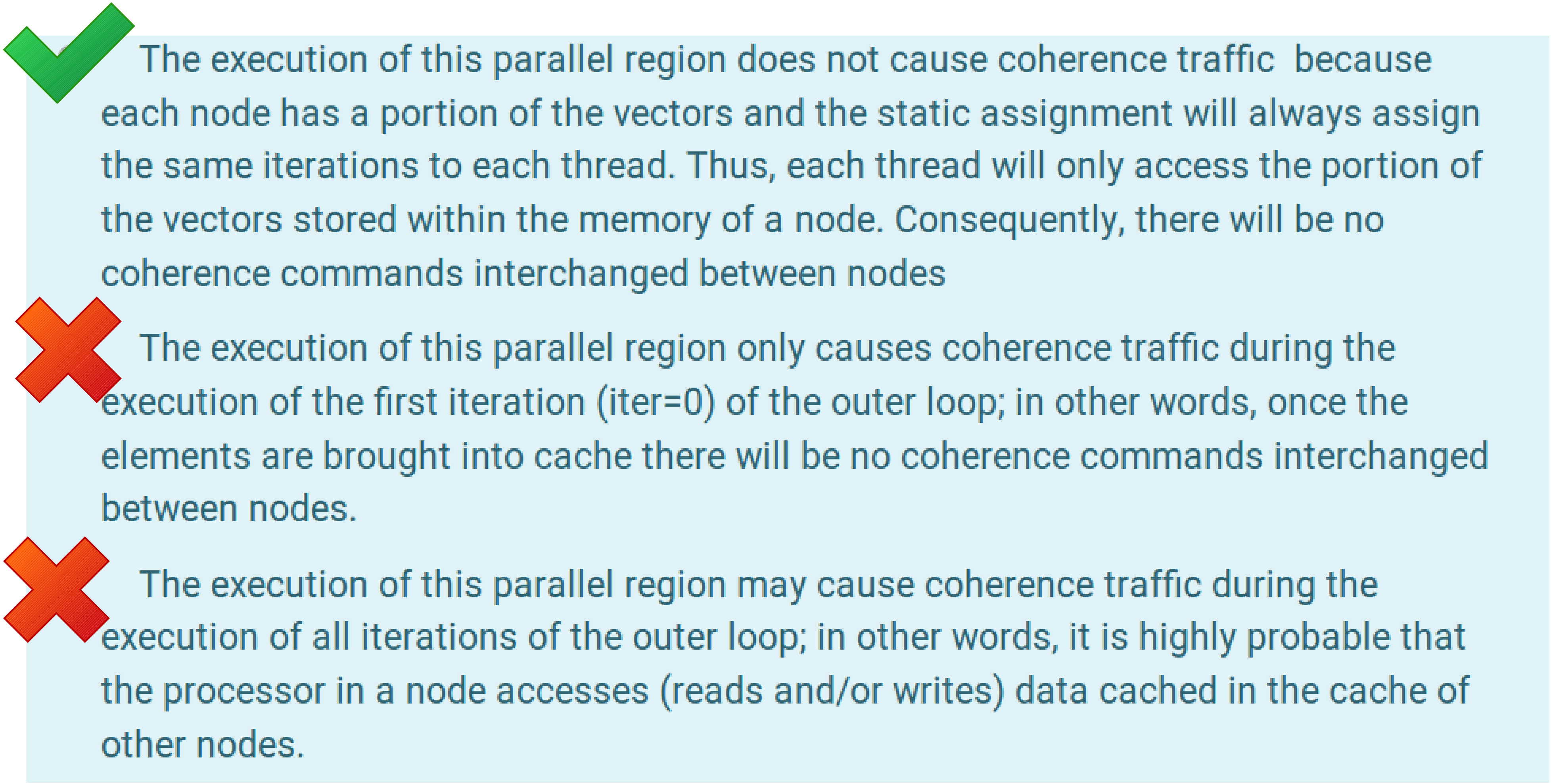
✗ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.



○ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

✗ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

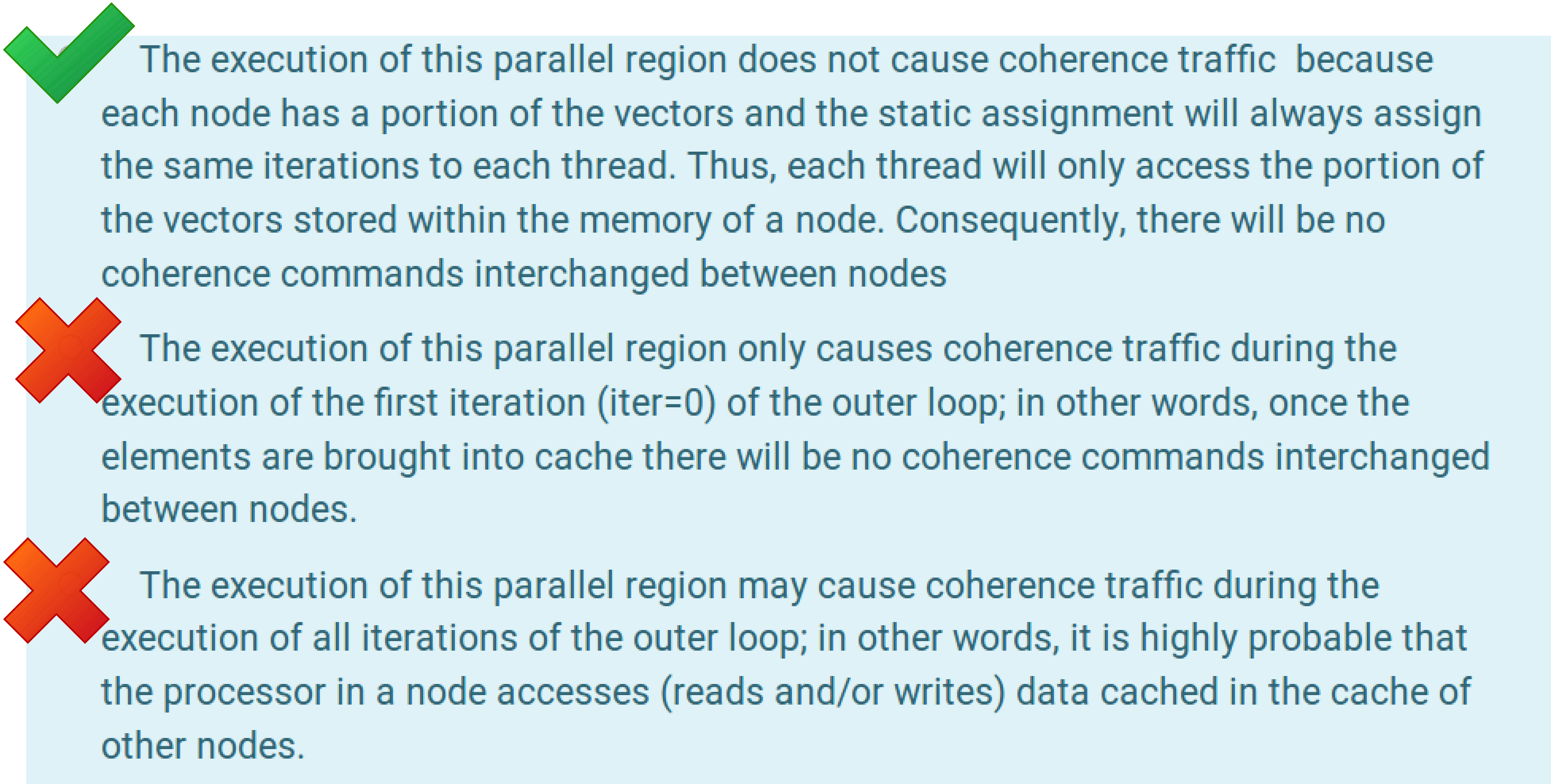
✗ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.



✓ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

✗ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

✗ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

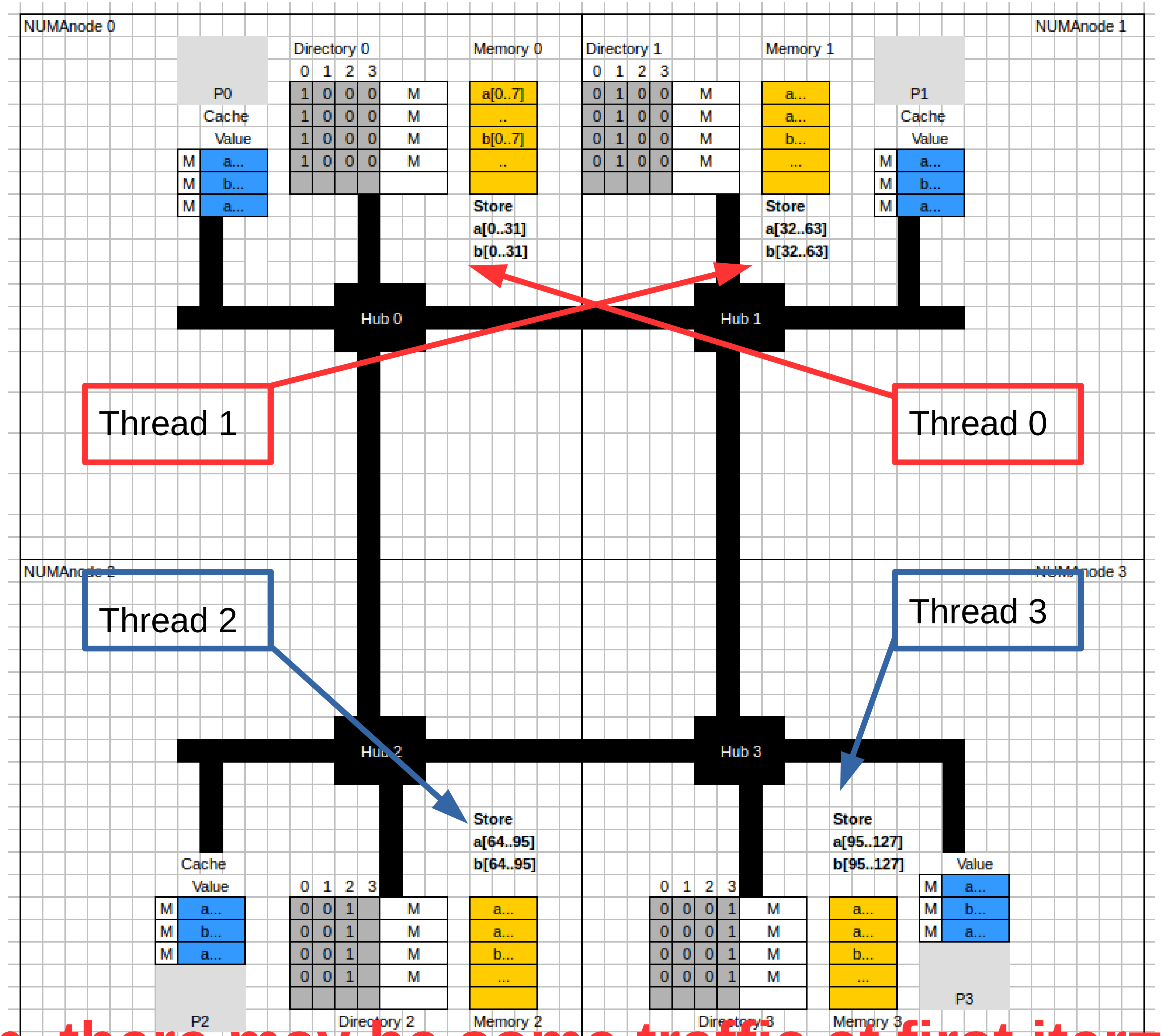


✓ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

✗ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

✗ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

If we don’t assume that thread i goes to NumaNode/Processor i... would be the answers correct?

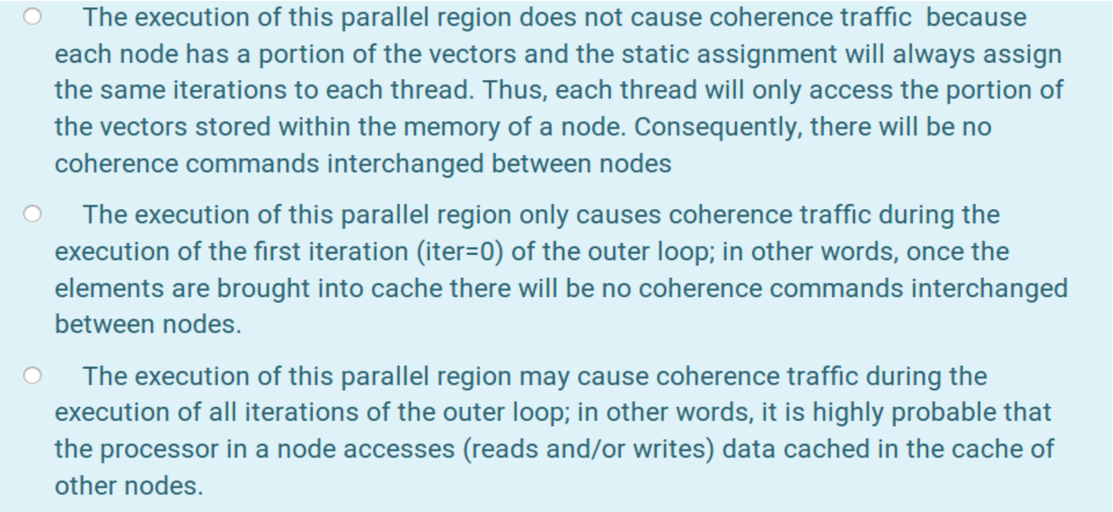


Then, there may be some traffic at first iter=0

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- **First iteration iter=0**
  - thread 0 and thread 1 would need to access NumaNode 1 and 0 respectively and invalidate copies of the caches of the another processor, and update presence bits.
  - Thread 2 and thread 3 would locally access their local data.
- Iter 1 and following
  - Caches of each processor will have already local copies of the data to be accessed in iter 1 and following
  - No more traffic is necessary



○ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

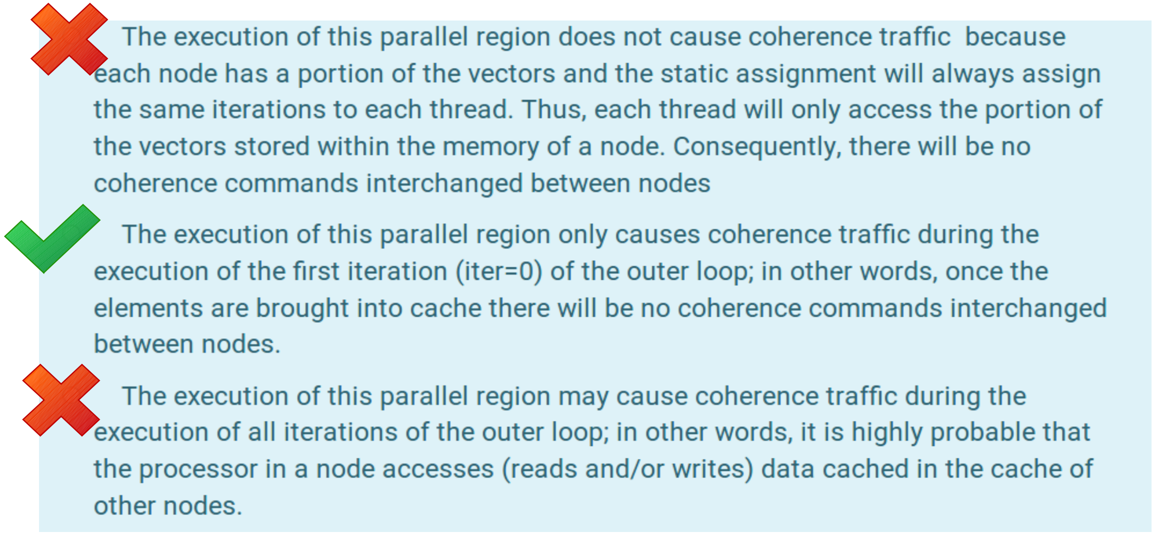
○ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

○ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- **First iteration iter=0**
  - thread 0 and thread 1 would need to access NumaNode 1 and 0 respectively and invalidate copies of the caches of the another processor, and update presence bits.
  - Thread 2 and thread 3 would locally access their local data.
- Iter 1 and following
  - Caches of each processor will have already local copies of the data to be accessed in iter 1 and following
  - No more traffic is necessary

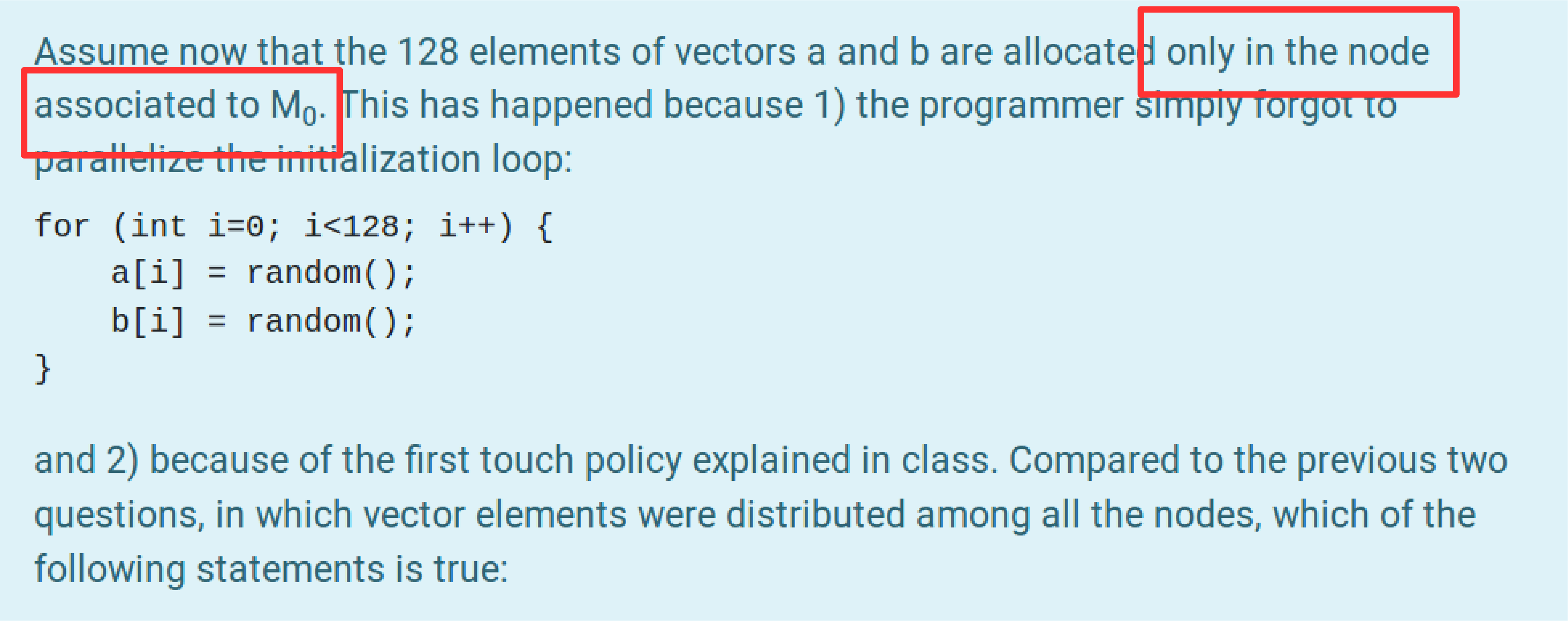


✗ The execution of this parallel region does not cause coherence traffic because each node has a portion of the vectors and the static assignment will always assign the same iterations to each thread. Thus, each thread will only access the portion of the vectors stored within the memory of a node. Consequently, there will be no coherence commands interchanged between nodes

✓ The execution of this parallel region only causes coherence traffic during the execution of the first iteration (iter=0) of the outer loop; in other words, once the elements are brought into cache there will be no coherence commands interchanged between nodes.

✗ The execution of this parallel region may cause coherence traffic during the execution of all iterations of the outer loop; in other words, it is highly probable that the processor in a node accesses (reads and/or writes) data cached in the cache of other nodes.

Statement



Assume now that the 128 elements of vectors a and b are allocated only in the node associated to M_0. This has happened because 1) the programmer simply forgot to parallelize the initialization loop:

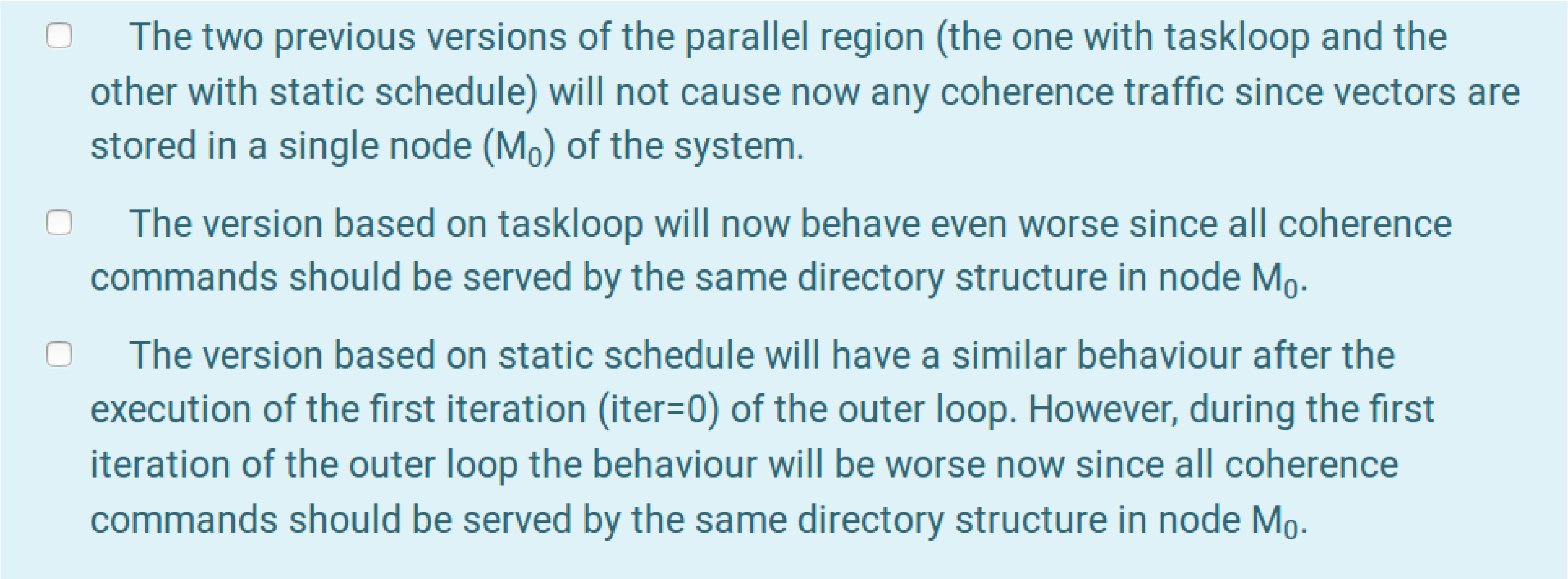
```
for (int i=0; i<128; i++) {
    a[i] = random();
    b[i] = random();
}
```

and 2) because of the first touch policy explained in class. Compared to the previous two questions, in which vector elements were distributed among all the nodes, which of the following statements is true:

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

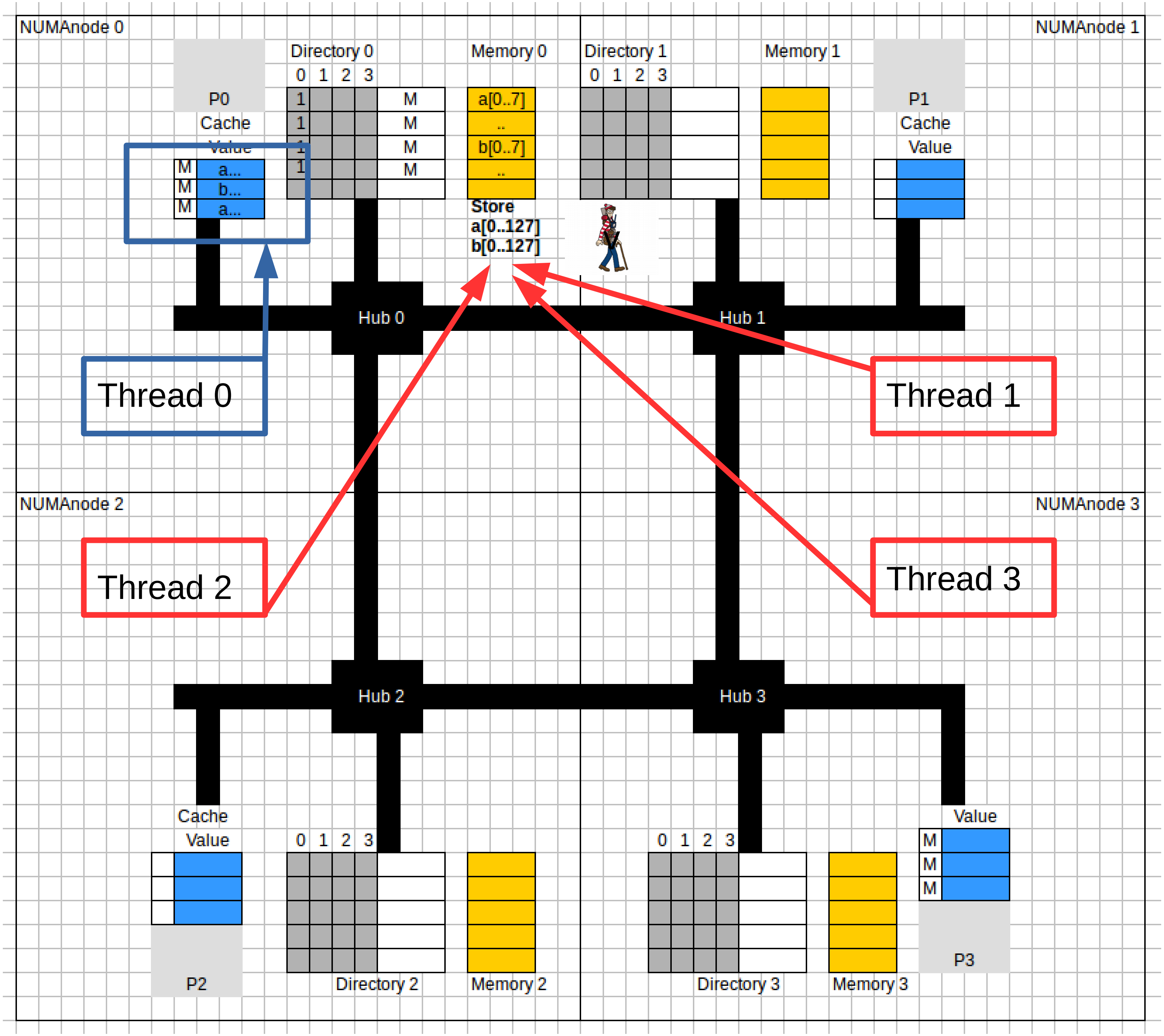
```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```



☐ The two previous versions of the parallel region (the one with taskloop and the other with static schedule) will not cause now any coherence traffic since vectors are stored in a single node (M_0) of the system.

☐ The version based on taskloop will now behave even worse since all coherence commands should be served by the same directory structure in node M_0.

☐ The version based on static schedule will have a similar behaviour after the execution of the first iteration (iter=0) of the outer loop. However, during the first iteration of the outer loop the behaviour will be worse now since all coherence commands should be served by the same directory structure in node M_0.



NUMAnode 0

NUMAnode 1

NUMAnode 2

NUMAnode 3

P0

Cache

Value

M	a...
M	b...
M	a...

Directory 0

0	1	2	3	
1				M
1				M
1				M
1				M

Memory 0

a[0..7]
..
b[0..7]
..

Store
a[0..127]
b[0..127]

Directory 1

0 1 2 3

Memory 1

P1

Cache

Value

Hub 0

Hub 1

Hub 2

Hub 3

Thread 0

Thread 1

Thread 2

Thread 3

Cache

Value

P2

0 1 2 3

Directory 2

Memory 2

0 1 2 3

Directory 3

Memory 3

Value

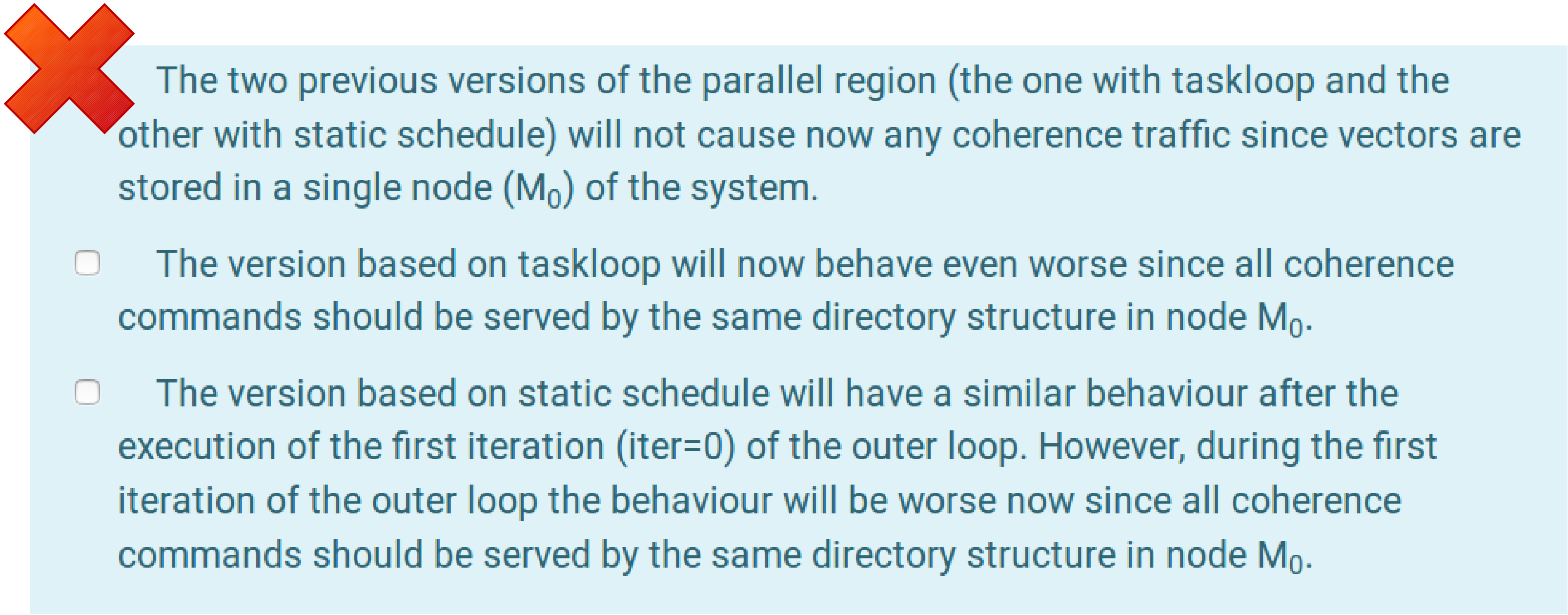
M
M
M

P3

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```



☒ The two previous versions of the parallel region (the one with taskloop and the other with static schedule) will not cause now any coherence traffic since vectors are stored in a single node (M_0) of the system.

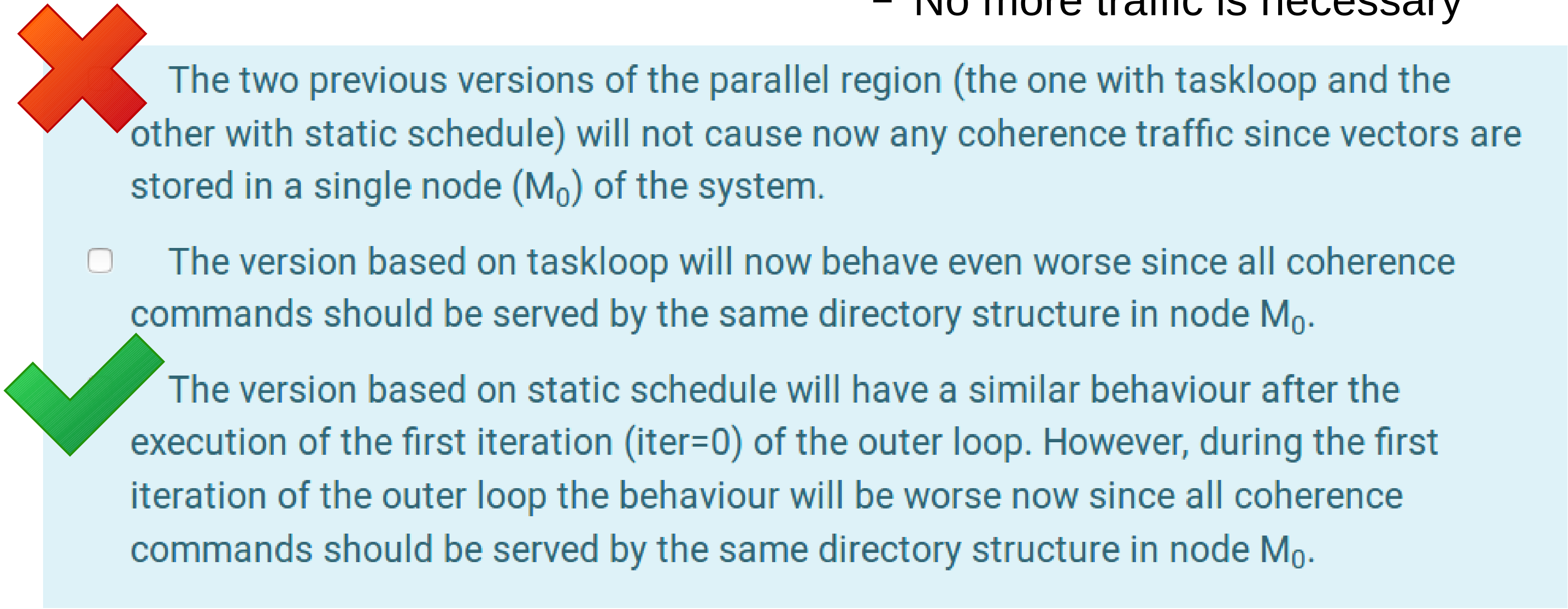
□ The version based on taskloop will now behave even worse since all coherence commands should be served by the same directory structure in node M_0.

□ The version based on static schedule will have a similar behaviour after the execution of the first iteration (iter=0) of the outer loop. However, during the first iteration of the outer loop the behaviour will be worse now since all coherence commands should be served by the same directory structure in node M_0.

Program Analysis

```
#pragma omp parallel num_threads(4)
for (iter=0; i<99; iter++) {
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- **First iteration iter=0**
  - Thread 1,2,3 will need to access NumaNode 0 to ask for data, update presence bits, and notify any WrReq or UpgrReq
  - Thread 0 will locally access their local data.
- Iter 1 and following
  - Caches of each processor will have already local copies of the data to be accessed in iter 1 and following
  - No more traffic is necessary



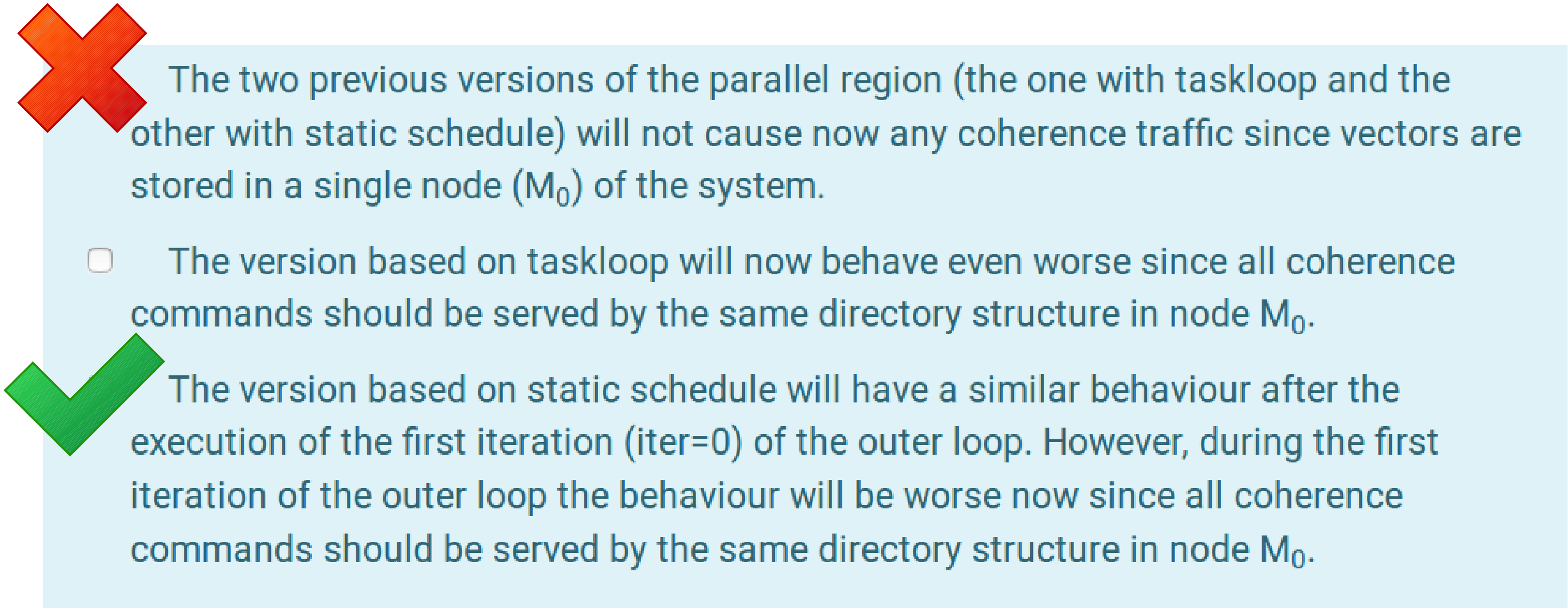
✗ The two previous versions of the parallel region (the one with taskloop and the other with static schedule) will not cause now any coherence traffic since vectors are stored in a single node (M_0) of the system.

☐ The version based on taskloop will now behave even worse since all coherence commands should be served by the same directory structure in node M_0.

✓ The version based on static schedule will have a similar behaviour after the execution of the first iteration (iter=0) of the outer loop. However, during the first iteration of the outer loop the behaviour will be worse now since all coherence commands should be served by the same directory structure in node M_0.

Program Analysis

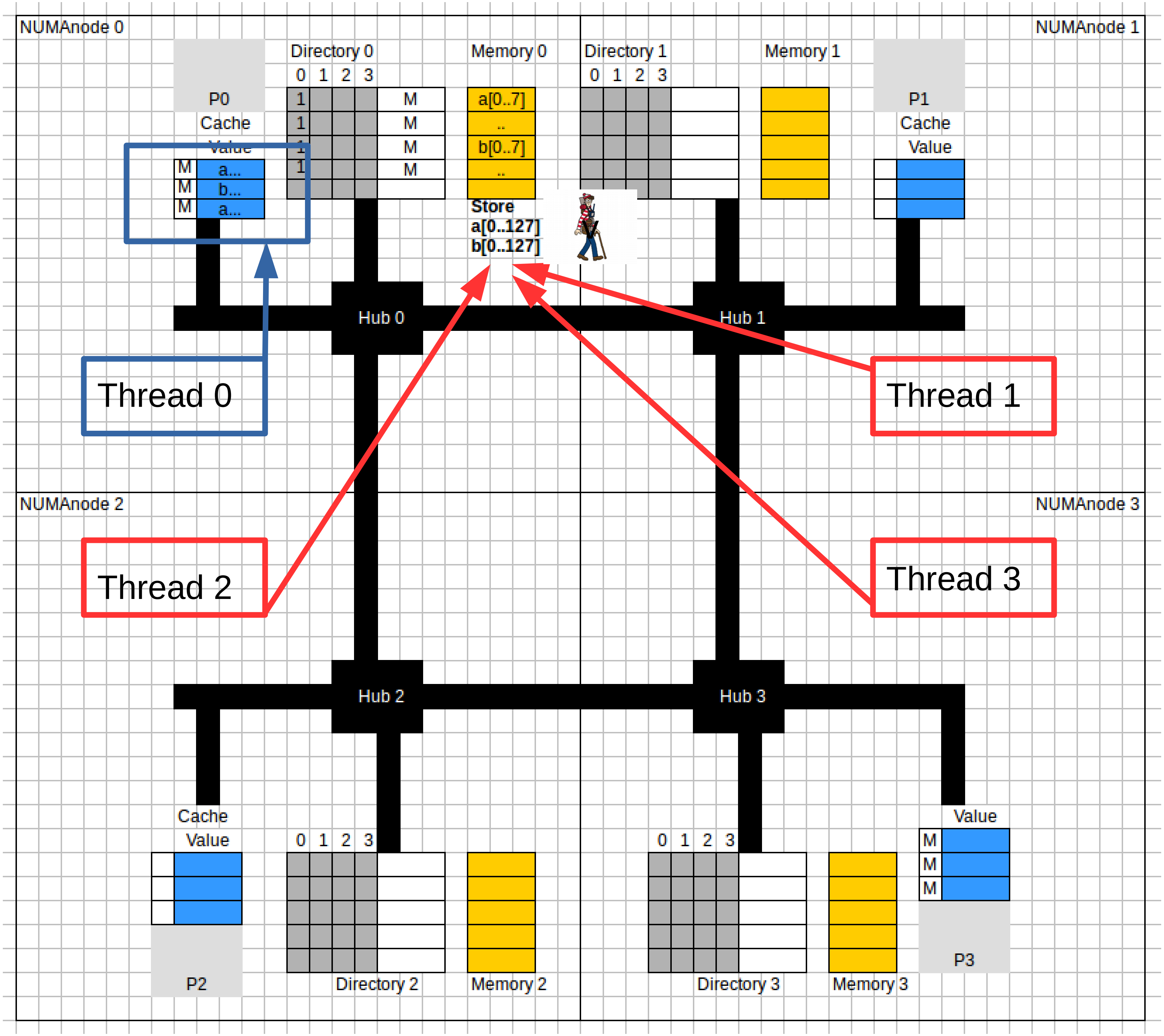
```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```



✗ The two previous versions of the parallel region (the one with taskloop and the other with static schedule) will not cause now any coherence traffic since vectors are stored in a single node (M_0) of the system.

□ The version based on taskloop will now behave even worse since all coherence commands should be served by the same directory structure in node M_0.

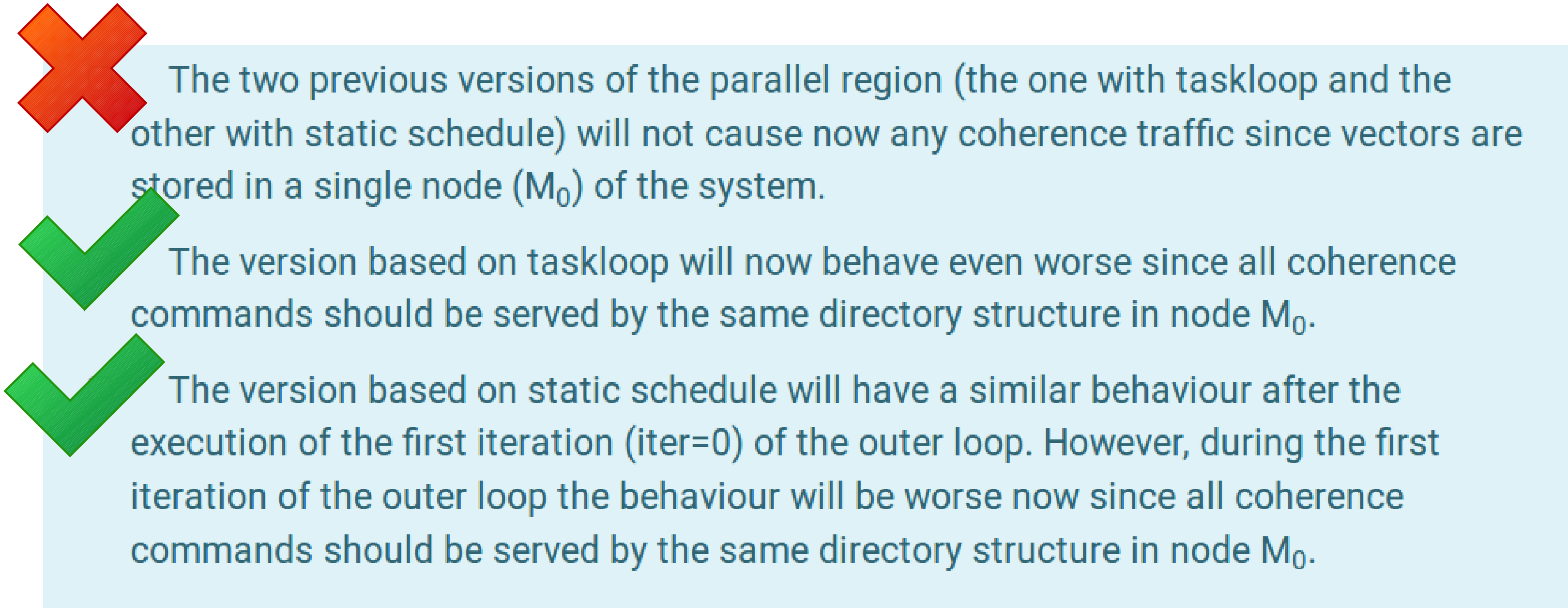
✓ The version based on static schedule will have a similar behaviour after the execution of the first iteration (iter=0) of the outer loop. However, during the first iteration of the outer loop the behaviour will be worse now since all coherence commands should be served by the same directory structure in node M_0.


NUMAnode 0
NUMAnode 1
NUMAnode 2
NUMAnode 3
P0
Cache
Value
M a...
M b...
M a...
Directory 0
0 1 2 3
1 M
1 M
1 M
1 M
Memory 0
a[0..7]
..
b[0..7]
..
Store
a[0..127]
b[0..127]
Directory 1
0 1 2 3
Memory 1
P1
Cache
Value
Hub 0
Hub 1
Thread 0
Thread 1
Thread 2
Thread 3
Hub 2
Hub 3
Cache
Value
P2
0 1 2 3
Directory 2
Memory 2
0 1 2 3
Directory 3
Memory 3
Value
M
M
M
P3

Program Analysis

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (iter=0; i<99; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<128; i++)
        a[i] = foo2(b[i]);
}
```

- Now all threads running tasks will access to the same NumaNode: NumaNode 0



✗ The two previous versions of the parallel region (the one with taskloop and the other with static schedule) will not cause now any coherence traffic since vectors are stored in a single node (M_0) of the system.

✓ The version based on taskloop will now behave even worse since all coherence commands should be served by the same directory structure in node M_0.

✓ The version based on static schedule will have a similar behaviour after the execution of the first iteration (iter=0) of the outer loop. However, during the first iteration of the outer loop the behaviour will be worse now since all coherence commands should be served by the same directory structure in node M_0.