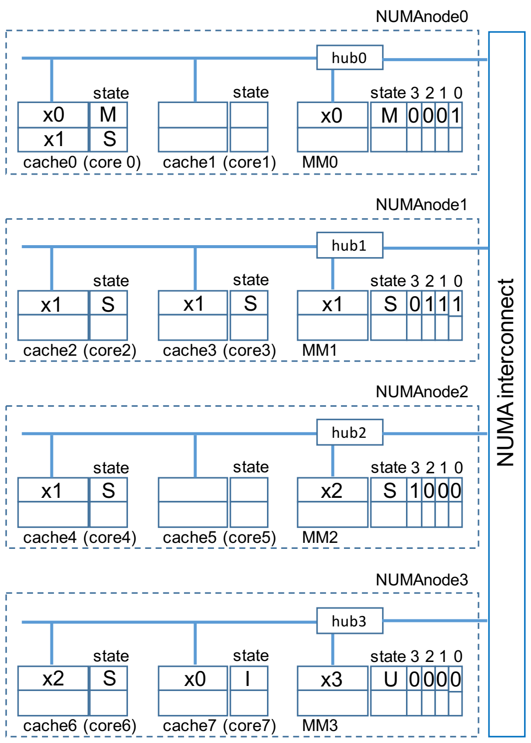
PAR – 2^{nd} In-Term Exam – Course 2016/17-Q2
May 31st, 2017

Problem 1 (3 points) Assume a multiprocessor system composed of four NUMA nodes, each with two processors (cores) with their own cache memory and a shared main memory (MM). Data coherence in the system is maintained using Write-Invalidate MSI protocols, with a Snoopy attached to each cache memory to provide coherency within each NUMA node and directory-based coherence among the four NUMA nodes.

Assuming the initial state of the memories (caches and MM) shown in the following representation of the memory system, in which only two lines are represented for each memory (cache or MM), with 4 variables `x0`, `x1`, `x2` and `x3`, 4 bytes wide each. Variables x2 and x3 reside in the same cache line (cache lines are 32 bytes wide) while the other two reside in a different cache line each.



Coherence commands
- **Core:** $PrRd_i$ and $PrWr_i$, being I the core number doing the action
- **Snoopy:** $BusRd_j$, $BusRdX_j$ and $Flush_j$, being j the snoopy/cache number doing the action
- **Hub/directoty:** $RdReq_{i\to j}$, $WrReq_{i\to j}$, $Dreply_{i\to j}$, $Fetch_{i\to j}$, $Invalidate_{i\to j}$ and $WriteBack_{i\to j}$, from NUMAnode i to NUMAnode j

Line state in cache
- M (modified), S (shared), I (invalid)

Line state in main memory
- M (Modified), S (shared), U (Uncached)

In the representation above we also include the list of possible coherence commands at the different levels and the state names for cache and memory lines; please use this notation whenever necessary.

1. Assuming that the multiprocessor system has 8 GB ($8 * 2^{30}$) of main memory, equally distributed in the four NUMA nodes and each processor has a cache memory of 4 MB ($4 * 2^{20}$), **we ask you** to compute the amount of bits taken by each snoopy to maintain the coherence between the caches inside a NUMA node and the amount of bits used in each node directory to maintain the coherence among NUMA nodes.

   Solution: (1 point) Each node has 2 GB of main memory organized in lines 32 bytes long. Therefore each NUMA node has $2 * 2^{30}/32 = 2^{26}$ lines. For each line the directory needs to store 2 bits for the state and 4 bits for the presence bits. In total $6 * 2^{26}$ bits. Each snoopy is associated to a cache memory with $4 * 2^{20}/32 = 2^{17}$ lines. For each line only the state needs to be maintained, which again for MSI is 2 bits; therefore $2 * 2^{17} = 2^{18}$ bits.

2. Indicate which one of the above mentioned variables (only one) is not in a correct state either in a cache or in main memory and the reason for that. Based on you answer, write that variable in the correct state in the appropriate memories (cache and/or MM) in the provided answer sheet.

Solution: (0.5 points) Variables `x2` and `x3` live in the same memory line, so it is not correct that they are mapped in two different NUMA nodes; in other words, each memory line can only exist in one home node. Then we can consider that variable `x3` does not exist in NUMAnode3 and just add it in the same line as the one containing `x2` in NUMAnode2.

3. If core `c7` writes on variable `x0`, indicate in the provided answer sheet which will be the state of the affected memories and directories after this memory access. You DON'T need to enumerate the sequence of coherence actions that happen.

   Solution: (0.5 points) The Snoopy in cache0 will change its line to *Invalid* state. The directory will not change the status for that line but will change the list of sharers removing NUMAnode0 and adding NUMAnode3. The Snoopy of cache 7 will update the state of the line from Invalid to Modified.

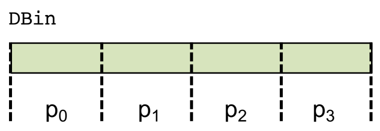
4. Finally, if core `c2` writes on variable `x1`, enumerate the sequence of coherence actions that will occur (in order of occurrence) and indicate in the provided answer sheet which will be the state of the affected memories and directories after this memory access.

   Solution: (1 point) In this case, core `c2` will issue $PrWr_2$; although the access results in a hit in cache2, the associated Snoopy will place $BusRdX_2$ in order to invalidate all possible copies of that line in other cache memories, either inside the same NUMA node or in other NUMAnodes. The state for that line in cache2 will transition from S to M. As a response to $BusRdX_2$, the Snoopy associated to cache3 will invalidate the line containing `x1` and the directory in NUMAnode1 will inform the hub that there are additional copies of that line in NUMAnodes 0 and 2, sending to them the corresponding invalidation commands $Invalidate_{1-0}$ and $Invalidate_{1-2}$. The hubs in these two NUMAnodes will transfer the invalidations to their respective buses by placing a *BusRdX*; as a response, the Snoopies for caches 0 and 4 will invalidate their lines containing `x1`. In addition, the state of that line in the memory of NUMAnode1 will change its state from S to M and the list of sharers updated in order to remove NUMAnodes 0 and 2 from the list.

Problem 2 (3,5 points) Given the following sequential code in C that looks for the position of all instances of a key (contained in variable `key`) in a portion of vector `DBin` (previously initialized randomly) storing the positions where key appears in vector `DBout`:

```
#define DBsize 1048576
int main() {
    double key=0.21;
    double * DBin = (double *) malloc(sizeof(double) * DBsize);
    double * DBout = (double *) malloc(sizeof(double) * DBsize);
    unsigned long counter = 0, min_pos, max_pos;

    for (unsigned long i = 0; i < DBsize; i++) // LOOP1
        DBin[i] = init(i);

    find_limits(DBin, &min_pos, &max_pos);

    for (unsigned long i = min_pos; i < max_pos; i++) // LOOP2
        if (DBin[i] == key) {
            DBout[counter] = i;
            counter++;
        }
}
```

1. Write a parallel version in OpenMP for the loop initializing `DBin` (LOOP1), assuming a *block data decomposition* for the input vector `DBin`, so that each thread is responsible for a block of $DBsize/P$ consecutive elements, being P the number of threads, as shown in the following figure.



Solution: (1 point)

```
#pragma omp parallel
    {
    unsigned long i, num_elems, lower;
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    lower = myid * (DBsize / howmany) +
            (myid < (DBsize % howmany) ? myid : DBsize % howmany);
    num_elems = (DBsize / howmany) + (myid < (DBsize % howmany));
    for (i = lower; i < lower + num_elems; i++)
        DBin[i] init(i);
    }
```

2. Write a parallel version in OpenMP for the second loop (LOOP2), assuming the same data decomposition strategy. You can reuse code from the previous question if needed.

   Solution: (1 point)

```
#pragma omp parallel
    {
    unsigned long i, num_elems, lower;
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    lower = myid * (DBsize / howmany) +
            (myid < (DBsize % howmany) ? myid : DBsize % howmany);
    num_elems = (DBsize / howmany) + (myid < (DBsize % howmany));
    for (i = max(lower, min_pos); i < min(lower + num_elems, max_pos); i++)
        if (DBin[i] == key) {
            DBout[counter] = i;
            counter++;
        }
    }
```

3. Insert the necessary synchronization in LOOP2 to avoid data races, minimizing the overhead that is introduced by that synchronization. Is the order of elements in `DBout` deterministic (i.e. is always the same for different executions of the parallel program)?

   Solution: (0,5 points)

```
        if (DBin[i] == key) {
            #pragma omp critical
            {
                DBout[counter] = i;
                counter++;
            }
        }
```

   The insertion order is not deterministic.

4. The originally proposed data decomposition has an important performance problem in LOOP2. Which problem are we talking about? Propose an alternative data decomposition for `DBin` that solves the performance problem and also maximizes locality in the access to `DBin`. You DON'T have to write the corresponding parallel code, just clearly describe the data decomposition proposed.

Solution: (1 point) There is a load balancing problem because the loop iterates from min_pos to max_pos, not traversing the whole vector DBin. We can solve it using a CYCLIC data decomposition. However, if we want to use all the elements in a cache line for `DBin`, we have to use a *block–cyclic decomposition* with as many elements per block as the cache line size divided by the size of a double. If the size of the cache line is S bytes and the size of a double is D bytes, the block size should be equal to S/D consecutive elements.

Problem 3 (1.5 points) The following piece of code shows the implementation of the `spin_lock` synchronization function, which works in the following way: the thread executing it tries to acquire a lock at the memory address `lock`; if the lock is already acquired, it waits for a while (`x` time units) and then tries again; the process is repeated until the thread succeeds acquiring the lock.

```
void spin_lock (int *lock, int x) {
     int ret;
     do {
          ret=test_and_set(lock, 1);
          if (ret==1)
               pause(x); /* Pause the program x time units */
     } while (ret==1);
}
```

We ask you:

1. Re-implement the `spin_lock` function for a new platform in which the `test_and_set` function is not available; you have to use `load_linked/store_conditional` instead.

   Solution: (0,5 points)

```
void spin_lock (int *lock) {
     int ret;
     int value;
     do {
          value=load_linked(lock);
          ret=store_conditional(lock, 1);
          if (value==1 || ret==0)
               pause(x);
     } while (value==1 || ret==0);
}
```

2. Write an optimized version for each of the two previous implementations of `spin_lock` functions (the one with `test_and_set` and the one with `load_linked/store_conditional`) with the objective of reducing coherency traffic.

   Solution: (1 point)

   The optimization applied is based on the test-test-and-set technique to reduce the number of coherence protocol operations.

```
void spin_lock (int *lock, int x) {
     int ret;
     do {
          /* test */
          while (*lock==1) {
               pause(x);
          }
          /* test-and-set */
          ret=test_and_set(lock, 1);
          if (ret==1)
```

```
            pause(x); /* Pause the program x time units */
        } while (ret==1);
}
```

```
void spin_lock (int *lock) {
     int ret;
     do {
          while (load_linked(lock)==1) {
               pause(x);
          }
          ret=store_conditional(lock, 1);
          if (ret==0)
               pause(x);
          } while (ret==0);
}
```

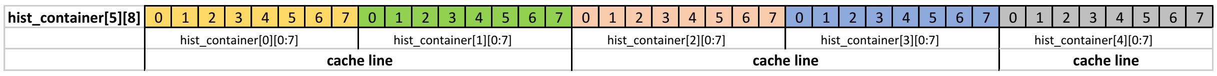
Problem 4 (2 points) Given the following parallel code in OpenMP, prepared to execute on a given number of threads (`nThreads`), that computes the histogram of vector `index` of `n` elements:

```
#define n 100000   // size of index vector
#define m 5        // Number of bins in histogram
#define nThreads 8 // Number of threads

int index[n];                     // input vector
int hist[m];                      // histogram
int hist_containers[m][nThreads]; // per-thread histogram

int main() {
    int iThread;

    InitIndex(index, n); // initialization of vector index
    initHistograms(hist, hist_containers, m, nThreads);

    #pragma omp parallel private(iThread) num_threads(nThreads)
    {
        iThread = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < n; i++)
             hist_containers[index[i]%m][iThread]++;
    }
    for (iThread = 0; iThread < nThreads; iThread++)
        for (int i = 0; i < m; i++)
            hist[i] += hist_containers[i][iThread];
}
```

We ask:

1. Identify the main performance bottleneck (problem) that occurs during the execution of the parallel region.

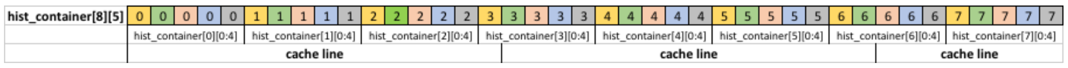
   Solution: (0.5 points) The main performance bottleneck in this parallel region is false sharing. During its execution different threads are continuously WRITING to elements of `hist_containers` that reside in the same cache line, invalidating each other copies. Only reading would not cause the performance bottleneck. In particular, since the line size is 64 bytes and each integer element occupies 4 bytes, a cache line holds 16 consecutive elements of `hist_containers`, which corresponds to 2 rows of the matrix. In total the whole structure `hist_containers` occupies 2 and a half cache lines.



hist_container[5][8]	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
	hist_container[0][0:7]								hist_container[1][0:7]								hist_container[2][0:7]								hist_container[3][0:7]								hist_container[4][0:7]							
	cache line																cache line																cache line							

2. If you change the definition of the per-thread histogram to `hist_containers[nThreads][m]` (and all the accesses to it in the code accordingly), will this solve the performance bottleneck identified?

   Solution: (0.5 points) The main performance bottleneck remains, since the whole data structure still occupies 2 and a half cache lines, although shared in a different way:



hist_container[8][5]	0	0	0	0	0	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5	5	5	5	5	6	6	6	6	6	7	7	7	7	7
	hist_container[0][0:4]					hist_container[1][0:4]					hist_container[2][0:4]					hist_container[3][0:4]					hist_container[4][0:4]					hist_container[5][0:4]					hist_container[6][0:4]					hist_container[7][0:4]				
	cache line																cache line																cache line							

   Now the same line is shared by less processors, but the false sharing problem still remains.

3. If the previous change did not solve the performance problem, do the necessary changes in the definition of `hist_containers` and/or code, without introducing other performance problems.

   Solution: (1 point) In order to avoid the FALSE SHARING problem to the data structure, the best solution is to add padding to the last definition. In particular, since cache lines are 64 bytes and each integer takes 4 bytes, we need to pad each row with 11 dummy elements. In this way each row occupies 64 bytes, a complete cache line.

```
#define m 5         // Number of bins in histogram
#define nThreads 8 // Number of threads

int hist_containers[nThreads][m+11]; // per-thread histogram
```

   This will also require to transpose all the accesses to `hist_containers` in the code.

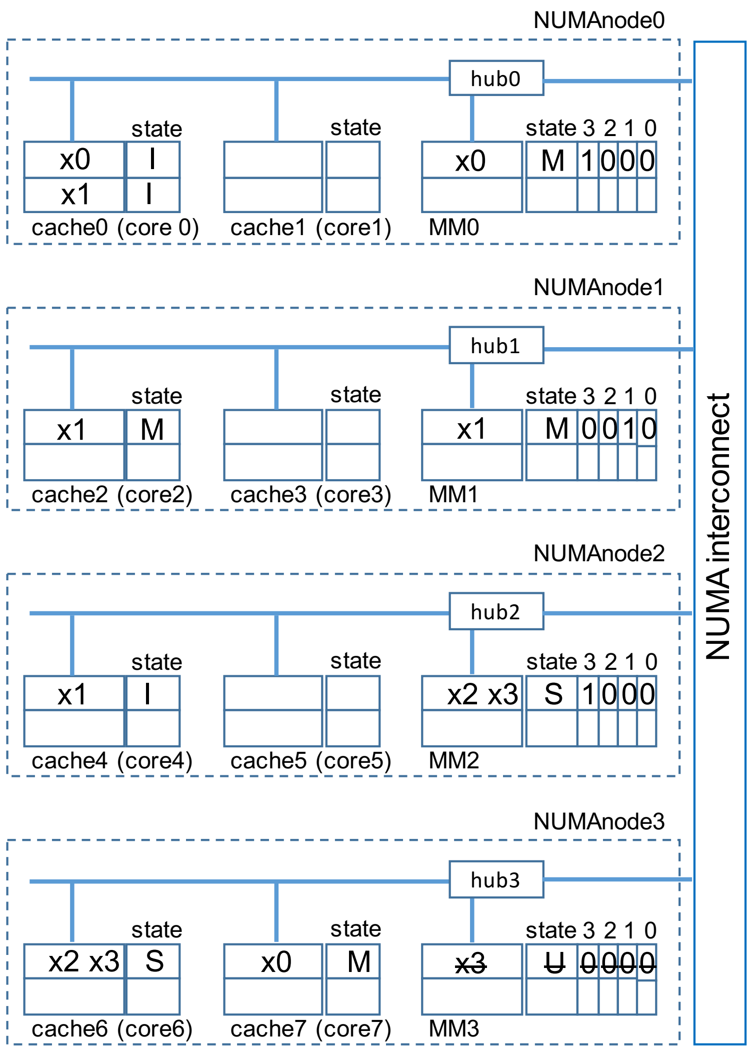
```
    ...
    #pragma omp parallel private(iThread) num_threads(nThreads)
    {
        iThread = omp_get_thread_num();
        #pragma omp for
        for (int i = 0; i < n; i++)
             hist_containers[iThread][index[i]%m]++;
    }
    for (iThread = 0; iThread < nThreads; iThread++)
        for (int i = 0; i < m; i++)
            hist[i] += hist_containers[iThread][i];
    ...
```

Note: You can assume that each integer occupies 4 bytes and cache lines are 64 bytes wide.

Student name: ....

Empty representation of the memory system, to be filled in with the final state for all the memories (caches and MM) after all the memory accesses in **Problem 1**.

Solution:



NUMAnode0
hub0
state
x0 I
x1 I
cache0 (core 0)
state
cache1 (core1)
state 3 2 1 0
x0 M 1 0 0 0
MM0
NUMAnode1
hub1
state
x1 M
cache2 (core2)
state
cache3 (core3)
state 3 2 1 0
x1 M 0 0 1 0
MM1
NUMAnode2
hub2
state
x1 I
cache4 (core4)
state
cache5 (core5)
state 3 2 1 0
x2 x3 S 1 0 0 0
MM2
NUMAnode3
hub3
state
x2 x3 S
cache6 (core6)
state
x0 M
cache7 (core5)
state 3 2 1 0
~~x3~~ ~~U~~ ~~0 0 0 0~~
MM3
NUMA interconnect