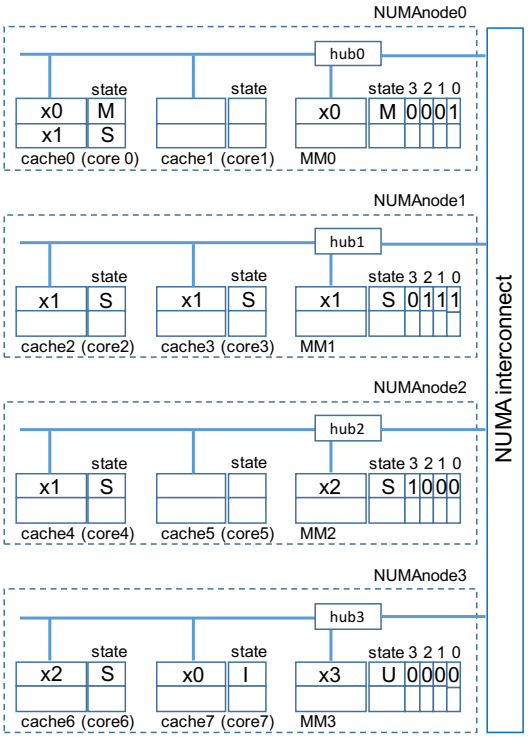
PAR – 2^{nd} In-Term Exam – Course 2016/17-Q2
May 31st, 2017

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.

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.

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.

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.

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`:

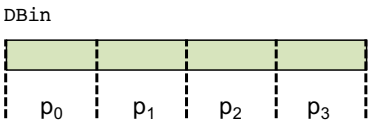
```
#define DBsize 1048576
int main() {
    double key=0.21;
    double * DBin = (double *) malloc(sizeof(double) * DBsize);
    double * DBout = (double *) malloc(sizeof(double) * DBsize);
    unsigned long counter = 0, min_pos, max_pos;

    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.



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.

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)?

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.

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.

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.

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.

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?

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.

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