PAR – Second In-Term Exam – Course 2017/18-Q2

June 6^{th}, 2018

Problem 1 (4 points) Given the following code (sequential version):

```
#define X_SIZE        40960
#define Y_SIZE        40960
#define V_SIZE    128
double *M,*V;
void main(int argc,char *argv[])
{
int i,j,aux,pos;
M=malloc(sizeof(double)*X_SIZE*Y_SIZE);
V=malloc(sizeof(double)*V_SIZE);
for (i=0;i<V_SIZE;i++) V[i]=0;
for (i=0;i<X_SIZE;i++) ReadRowFromFile(M,i);
// Main loop
for (i=0;i<X_SIZE;i++){
   for (j=0;j<Y_SIZE;i++){
                  aux=ComputeElement(M,i,j);
                  pos=ComputePos(i,j);
                  V[pos]=+aux;
   }
}
}
```

Comment: Function ReadRowFromFile is not provided . It reads one row (i) from disk , assuming each row has Y_SIZE elements, and stores the result in M (row=i). ComputePos is a function that computes the position to be inserted on V. It only depends on i,j. The function ComputeElement(M,i,j) doesn't modify M.

1. We ask you to create a OpenMP parallel version applying a block geometric data decomposition strategy per rows on matrix M for both initialization and main loop. You cannot use the `#pragma omp parallel for` neither `#pragma omp for` constructs. The block geometric decomposition should minimize the load unbalance on the distribution of the rows. Also, reason if you need to include any synchronization in the main loop.

2. Create a new version were we will implement an output data decomposition to be sure each processor is accessing to N consecutive positions of vector V. Reason if you need to include some synchronization.

Problem 2 (3 points) Given the following OpenMP code:

```
   struct t_person{
      t_data data; // personal information of bank client
      float balance; // current balance for client
      float interest; // interest for client
   };

   struct t_person best_client;

   void find_best_client(struct t_person *people, int n) {
      #pragma omp parallel for
      for(int i=0; i<n; i++)
         if (best_client.balance< people[i].balance)
              best_client = people[i];
   }

   int main() {
      struct t_person bank_info[N_MAX];
      ...
      best_client.balance=0.0;

      find_best_client(bank_info, N_MAX);
      ...
}
```

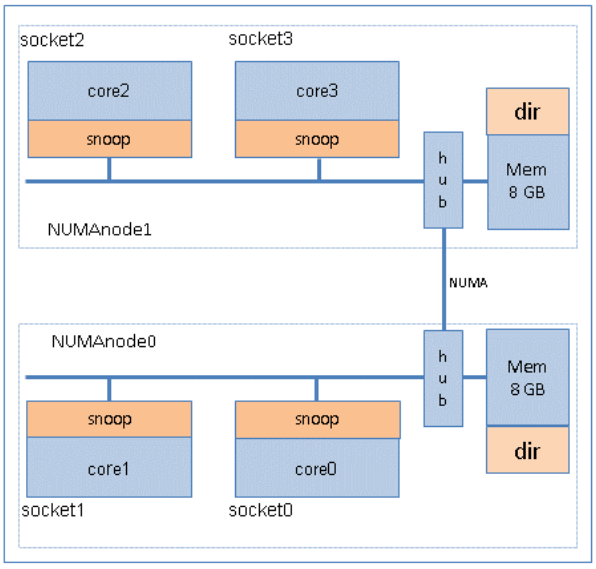
Considering that `reduction` clause cannot be used on variables of type struct, **we ask you:**

1. There is a concurrency problem in the proposed OpenMP code for `find_best_client`. What is this concurrency problem? Reason your answer.

2. Propose a modification of the code to avoid this concurrency problem just inserting a `omp` construct.

3. Propose an improvement of your previous modification to significantly reduce unnecessary synchronization overheads. This solution should also avoid false sharing problems.

Problem 3 (3 points) Assume a multiprocessor system composed of two NUMA nodes, each with two sockets. Each socket has one core with a cache memory of 8MB. **The cache line size is 16 bytes**. 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 two NUMA nodes**.



coreX: Core.
socketY: Package with 1 core,
8 MB cache and snoopy coherence protocol
The cache line size is 16 bytes
NUMAnodeZ: set of 2 sockets connected to the same NUMA "hub"/directory with 8 GB of main memory.
node: Node with 2 NUMAnodes.

Coherence commands
• **Core:** PrRdi and PrWri, being i the core number doing the action
• **Snoopy:** BusRdj, BusRdXj and Flushj, being j the snoopy/cache number doing the action
• **Hub/directoty:** RdReqij, WrReqij, Dreplyij, Fetchij, Invalidateij and WriteBackij, from NUMAnode i to NUMAnode j

Line state in cache
• M (modified), S (shared), I (invalid)
Line state in main memory (MM)
• M (Modified), S (shared), U (Uncached)

Given the following C code:

```
#define N 16
int x[N];
...
#pragma omp parallel num_threads(4)
{
    int myid = omp_get_thread_num();
    int nths = omp_get_num_threads();
    int i_start, i_end;

    i_start = (N / nths) * myid;
    i_end = i_start + N/nths;
    // FOR loop
    for (int i=i_start; i<i_end; i++) x[i]=init();
}
```

and assuming that: 1) the Operating System decides the data allocation using a "first touch" policy; 2) vector `x` is the only variable that will be stored in memory (the rest of variables will be all in registers of the processors); 3) the initial address of vector `x` is aligned with the start of a cache line; 4) the size of an `int` data type is 4 bytes; and 5) $thread_i$ always executes on $core_i$, where $i = [0 - 3]$, **we ask you:**

1. Compute the amount of bits taken by each snoopy to maintain the coherence between caches inside a NUMA node and, the amount of bits in each node directory to maintain the coherence among NUMA nodes.

2. Draw a picture that shows vector `x` and how many memory lines are necessary to store its elements, identifying the range of elements per memory line.

3. Assuming that all cache memories are empty at the beginning of the program, fill in the attached `Table 1` with the information corresponding to each range of elements allocated per memory line once all threads arrive to the end of the parallel region: vector range, the Home node number, the presence bits, main memory line state (`State in MM`) corresponding to accesses to vector `x`, and the state of any copy (`State in cache socket0-3` in the table) of those memory blocks in one or more caches of sockets 0 to 3.

   Solution: Table 1: to be used to deliver your solution to Problem 3.3

Vector x range	# Home NUMA node	Presence bits	State in MM	State in cache			
				socket0	socket1	socket2	socket3
0-3							
4-7							
8-11							
12-15							

4. Assuming the final previous state of the multiprocessor system with the presence bits and state for each cache and memory line, fill in the attached `Table 2` with the sequence of processor commands (Core), bus transactions within NUMA nodes (Snoopy), transactions between NUMA nodes (Directory), the presence bits, state for each cache and memory line, to keep cache coherence, **AFTER the execution of each** of the following sequence of commands:

   (a) $core_2$ reads the contents of `x[2]`

   (b) $core_2$ writes the contents of `x[2]`

   (c) $core_1$ reads the contents of `x[0]`

Solution: Table 2: to be used to deliver your solution to Problem 3.4

Command	Coherence actions			Presence bits	State in MM	State in cache			
	Core	Snoopy	Directory			socket0	socket1	socket2	socket3
$core_2$ reads x[2]									
$core_2$ writes x[2]									
$core_1$ reads x[0]									