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.

   Solution: variables `i`, `j`, `aux` and `pos` must be declared as `private`. It has been considered as correct solution the use of `critical` or `atomic` constructs to protect the access to vector `V`. However, the most efficient solution is using a vector of lock variables with as many elements as `V_SIZE`.

```
#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;
    #pragma omp parallel private(i,j,aux,pos)
    {
```

```
    int id=omp_get_thread_num();
    int nump=omp_get_num_threads();
    int lower,upper;
    // compute the lower and upper limits
    lower=id*(X_SIZE/nump);
    if (id<(X_SIZE%nump)) lower+=id;
    upper=lower+(X_SIZE/nump)+(id<(X_SIZE%nump));
    for (i=lower;i<upper;i++) ReadRowFromFile(M,i);
    // Main loop
    for (i=lower;i<upper;i++){
      for (j=0;j<Y_SIZE;i++){
       aux=ComputeElement(M,i,j);
       pos=ComputePos(i,j);
       #pragma omp atomic
       V[pos]=+aux;
      }
    }
   }
}
```

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.

   Solution: We must compute specific limits for vector `V` and update only those positions assigned to the current thread. With this strategy there is no need to use synchronisation since matrix `M` is not modified and each thread is accessing different positions of vector `V`.

```
#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
    #pragma omp parallel private(i,j,aux,pos)
    {
     // Computing lower and upper limits
     int id=omp_get_thread_num();
     int nump=omp_get_num_threads();
     int pos_lower,pos_upper;

     pos_lower=id*(V_SIZE/nump);
     if (pos_lower<(V_SIZE%nump)) pos_lower+=id;
     pos_upper=pos_lower+(V_SIZE/nump)+(id<(V_SIZE%nump));

     for (i=0;i<X_SIZE;i++){
       for (j=0;j<Y_SIZE;i++){
         pos=ComputePos(i,j);
         if ((pos>=pos_lower) && (pos<pos_upper)){
```

```
                aux=ComputeElement(M,i,j);
                V[pos]=+aux;
            }
         }
      }
    }
 }
```

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.

   Solution:

   There is a data race condition when updating global variable `best_client`: two threads may concurrently compare (read) the value in `best_client.balance` with value in the `people[i].balance` they have read, and decide to update `best_client`. The read and update must be done in mutual exclusion to ensure that the global `best_client` variable is updated appropriately.

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

   Solution:

   We only need to insert a `#pragma omp critical` to force mutual exclusion, including the read (if statement) and update (assignment) of the `best_client` global variable. We cannot use `#pragma omp atomic` since this only works for some basic operations.

   ```
   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++)
      #pragma omp critical
      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);
   ...
}
```

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

   Solution:

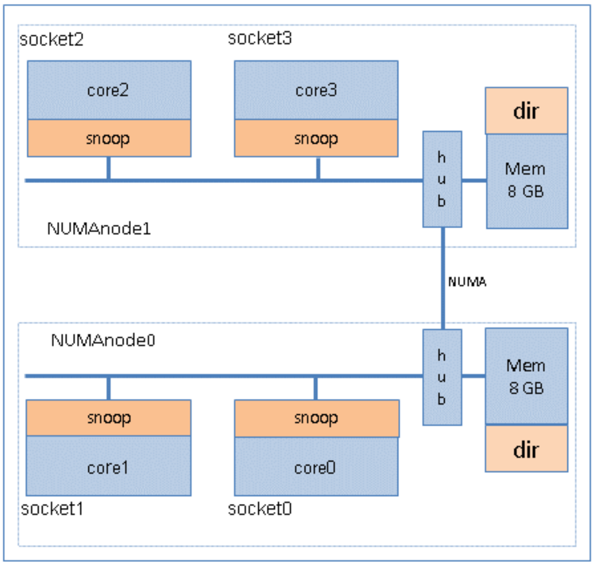
   In order to significantly reduce the amount of unnecessary synchronization we can use the `test&test&set` technique. We will check `best_client.balance` value before forcing a mutual exclusion area. If there is a chance of updating the `best_client` variable, we force a mutual exclusion and check and update the `best_client` global variable if necessary. With this solution there is not a false sharing situation to avoid.

```
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)
        #pragma omp critical
          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);
   ...
}
```

   Other possible solutions are:

- To use a private and local `local_best_client` for each thread, and then, after the for loop and before finishing the `parallel` region, use a `critical` construct to update `best_client` global variable.
- To use a global vector variable of as many entries as number of threads. Then, each thread will update one entry of the vector with the `best_client` they find. However, this last version may have a false sharing problem because two consecutive entries of the vector of `best_client`, updated by two different threads, may share a cache line. In this case, padding the elements of the vector or create a table with the enough padding to provoke that two elements of the vector doesn't share any cache line will avoid the false sharing problem.

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:**

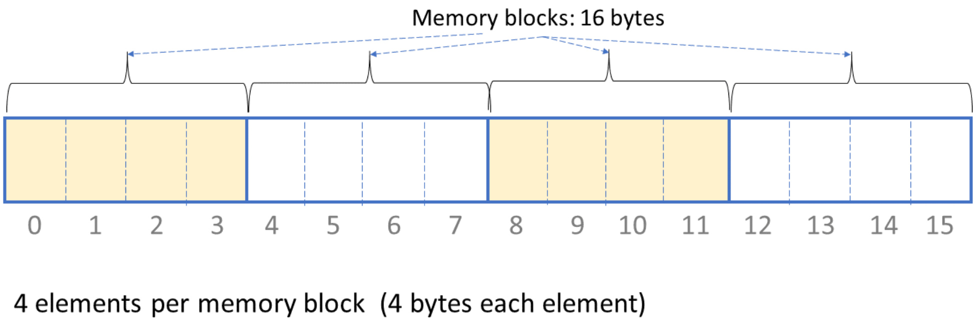
1. Compute the amount of bits taken by each snoopy to maintain the coherence between caches inside a NUMA node and, the amount of bits in each node directory to maintain the coherence among NUMA nodes.

   Solution:

   Each node has 8 GB of main memory organized in lines 16 bytes long. Therefore each NUMA node has $8 * 2^{30}/16 = 2^{29}$ lines. For each line the directory needs to store 2 bits for the state and 2 bits for the presence bits; therefore $4 * 2^{29} = 2^{31}$ bits. Each snoopy is associated to a cache memory with $8 * 2^{20}/16 = 2^{19}$ lines. For each line only the state needs to be maintained, which again for MSI is 2 bits; therefore $2 * 2^{19} = 2^{20}$ bits.

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.

   Solution:



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	0	01	M	*M*	I	I	I
4-7	0	01	M	I	*M*	I	I
8-11	1	10	M	I	I	*M*	I
12-15	1	10	M	I	I	I	*M*

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]	$PrRd_2$	$BusRd_2$ $BusRd/Flush_0$	$RdReq_{10}$ $Dreply_{01}$	11	S	S	I	S	I
$core_2$ writes x[2]	$PrWr_2$	$BusRdX_2$ $BusRdX$	$WrReq_{10}$	10	M	I	I	M	I
$core_1$ reads x[0]	$PrRd_1$	$BusRd_1$ $BusRd/Flush_2$	$Fetch_{01}$ $Dreply_{10}$	11	S	I	S	S	I

1. $core_2$ reads the contents of `x[2]`: $core_2$ triggers a $PrRd$ event; the line that holds `x[2]` is not loaded in the local cache, so the associated snoopy generates a $BusRd$ transaction to notify about the reading operation. The Local Hub is not the Home Hub, so the Local Hub sends a $RdReq$ operation to the Home Hub (in Numa Node 1) to ask for a copy of the value. The Home Hub knows because of the presence bits that there is a modified copy in its Numa node. A $BusRd$ transaction on the bus generates a $Flush$ operation from $core_0$ and the cache line state is modified to Shared. The line state in the directory of the Home Node is also changed to $Shared$ and in the list of sharers Numa Node 1 is added. The Home Hub sends back with a $Dreply$ message to the Local Hub the requested line.

2. $core_2$ writes the contents of `x[2]`: $core_2$ triggers a $PrWr$ event; although the element `x[2]` is already in the local cache (hit), the state is shared so the associated snoopy generates a $BusRdX$ to notify other caches that the line is going to be modified. The local Hub sends a $WrReq$ message to the Home Hub which owns the list of sharers that have a copy of this element. The Home Hub launches a $BusRdX$ transaction (presence bits indicate there is a copy in Numa Node 0), then cache line state where the copy resides is invalidated. Line state in the directory in the Home Node is changed to $Modified$, presence bits are changed to 10 expressing that just Numa Node 1 has a valid copy of the line.

3. $core_1$ reads the contents of `x[0]`: $core_1$ generates a $PrRd$ event; the line where `x[0]` resides is not loaded in the local cache, so this activates the snoopy protocol generating a $BusRd$ transaction. As the variable `x[0]` shares memory line with `x[2]`, the element is already loaded in the cache associated to $core_2$, in the other Numa Node (this information is in the list of sharers). The Local Hub, which in this case is the Home Hub, gets the current value of `x[0]` by asking the line with a $Fetch$ operation to the Owner Hub. The Owner Hub activates the snoopy protocol, sending a $BusRd$ operation, which makes $core_2$ to Flush the cache line and changes its cache line state to $Shared$. The Owner Hub flushes the line and responds to the Home Hub with a $Dreply$ operation. The line state in memory is updated to $Shared$, the list of sharers is updated by adding Home Hub, and the cache line state associated to the cache of $core_1$ is changed to $Shared$.