PAR – 2^{nd} In-Term Exam – Course 2018/19-Q2

May 29^{th}, 2019

Problem 1 (4 points) Given the following incomplete OpenMP implementation of a code:

```
int A[MAX_HASH]; // hash table
int B[N];        // input vector
int sum = 0;
...
#pragma omp parallel
{
   ...
   for(int i=0; i<N; i++)
      A[hash(B[i])]++;
   ...
   for(int i=0; i<MAX_HASH; i++)
      sum += foo(A[i]);
   ...
}
...
```

Assuming that we want to implement a **geometric data decomposition** that should:

- Avoid the need of synchronisation during the execution of the **first loop**.
- Allow the exploitation of temporal data locality **between the first and second loop**.
- Avoid any synchronisation and false sharing situation during the execution of the **second loop**.
- Minimise the load unbalance (maximum unbalance of one element) among threads (processors).

The implementation of the data decomposition cannot make use of the `#pragma omp for` construct neither fuse the two loops (i.e. joint the two loops into a single one). You can assume that both data structures `A` and `B` are cache aligned, i.e. their first element is placed at the beginning of a cache line.

We ask you:

- (1 point) Which data structure(s) should you decompose in the first loop (input and/or output)? Which geometric decomposition(s) should you apply to achieve the required load balance (block, cyclic or block–cyclic)? Why? Reason your answers.

  Solution: The appropriate data decomposition strategy should be *geometric block output* data decomposition (i.e. applied to data structure `A`). With this stretegy each processor is responsible for any update occurring to a block of elements of `A`, with no accesses by other processors. So no synchronization is needed (i.e. each thread will update different positions of `A`.

- (2 points) Implement the OpenMP parallelisation using the geometric decomposition you have decided in the previous question.

  Solution:

  ```
  int A[MAX_HASH];
  int B[N];
  int sum=0;
  ```

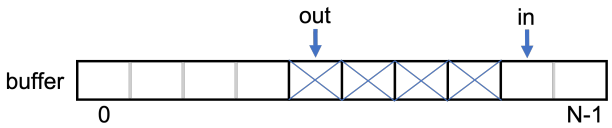
```
#pragma omp parallel reduction(+:sum)
{
  int nt=omp_get_num_threads();
  int id=omp_get_thread_num();
  int mod=MAX_HASH%nt;
  int nelem=MAX_HASH/nt;
  int start_hash=(nelem * id)+(id<mod)?id:mod;
  int end_hash=start_hash+nelem+(id<mod);

  for(int i=0; i<N; i++)
  {
    int value = hash(B[i]);
    if (value>=start_hash && value<end_hash)
       A[hash(B[i])]++;
  }

  for(int i=start_hash; i<end_hash; i++)
    sum+=foo(A[i]);
}
```

To minimize load unbalance we assign one more element to the first `mod` processors. Since the two loops have the same loop bounds (`start_hash` and `end_hash`) they will access the same elements of the hash table `A`; therefore we allow the exploitation of temporal locality. WIth the `reduction` clause we avoid the use synchronization during the execution of the second loop.

- (1 point) In your previous implementation, is there any possibility of false sharing **during the execution of the first loop** due to the use of the selected geometric data decomposition? Why? Reason your answer. If necessary, explain what would you do to avoid it (**we are not asking to re-implement the code to avoid false sharing, just explain).**

  Solution: The initially proposed data decomposition strategy may introduce false sharing when updating the first and last elements of the block assigned to each processor (e.g. the last elements of processor 0 and the first elements of processor 1 may reside in the same cache line. This constraint was not included in the list of constraints, so a block decomposition was totally valid.However, to avoid this false sharing a *geometric block-cyclic output* decomposition would be more appropriate, setting the number of consecutive elements per block to `CACHE_LINE` ÷ `INT_SIZE`, being `CACHE_LINE` the size in bytes of each memory line and `INT_SIZE` the size in bytes of each integer element.

Problem 2 (3 points) Given the following parallel code in OpenMP for a *producer–consumer* problem:




```
int buffer[N];
int full_slots=0;  // number of full positions in buffer
int empty_slots=N; // number of empty positions in buffer
void down (int *slots);
void up (int *slots);
...
#pragma omp parallel
#pragma omp single
{
  #pragma omp task // producer
```

```
{
 int in=0; // first position in buffer to insert to
 while (1) {
    int data = produceData(); // generates data
    down(&empty_slots);       // blocks when empty_slots is 0 (buffer is full).
                              // decreases empty_slots once it is not 0 and return
    buffer[in] = data;
    in = (in+1) % N;
    up(&full_slots);          // increments by 1 the counter of full slots
} }
#pragma omp task // consumer
{
 int out=0; // first position in buffer to remove from
 while (1) {
    down(&full_slots);        // blocks when full_slots is 0 (buffer is empty).
                              // decreases full_slots once it is not 0 and return
    int data = buffer[out];
    out = (out+1) % N;
    up(&empty_slots);         // increments by 1 the counter of empty slots
    consumeData(data);        // process data
} } }
```

We ask you to implement the semaphore operations `down` and `up` that are used to synchronise the access to the shared `buffer`, defined as follows:

```
void down (int *slots) {
    // if the value stored in slots is greater than 0, then decrement slots
    // otherwise wait until slots is greater than 0, then decrement slots
}
void up (int *slots) {
    // Increment slots by 1
}
```

You can assume that the architecture supports any of the low–level synchronisation operations studied during the course and that they are available for use. The implementation **should avoid unnecessary coherence traffic**.Below you have a list of such operations:

```
int test_and_set (int *address, int vallue);
int fetch_and_op (int *address, int value); // op can be add, sub, ...
int load_linked (int *address);
int store_conditional (int *address, int value); // returns 1 in case of success
```

Solution based on the use of `test_and_set`:

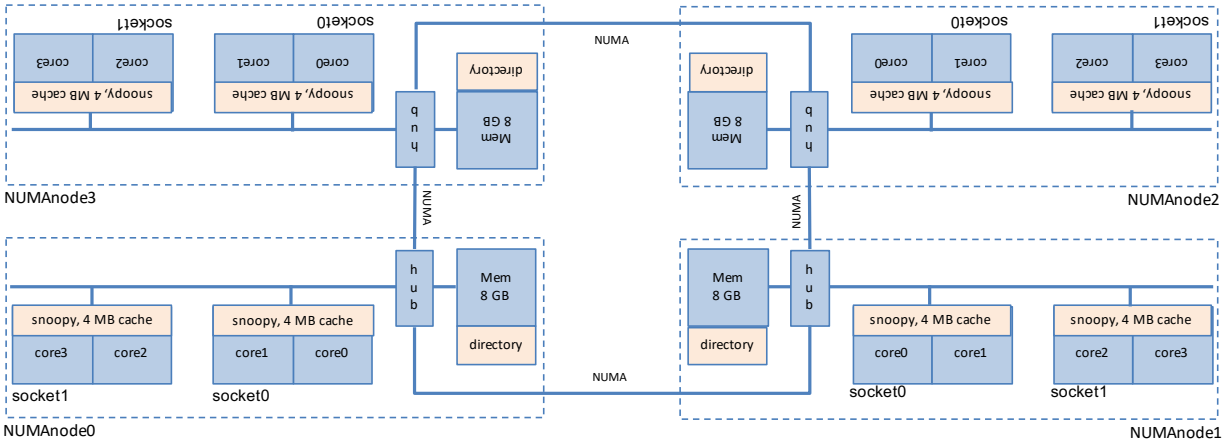
```
 int lock=0;
 void down (int *sem) {
     do {
         while (lock>0);
         while (test_and_set(&lock, 1) == 1);
         int value = *sem;
         if (value > 0) *sem--;
         lock = 0;
     } while (value == 0);
 }
 void up (int *sem) {
     while (test_and_set(&lock) == 1);
     *sem++;
     lock = 0;
```

```
}
```

Solution based on the use of `load linked - store conditional`:

```
void down (int *sem) {
    int ret=0;
    do {
        int value = load_linked (sem);
        if (value > 0)
            ret = store_conditional (sem, value-1);
    } while (ret == 0 || value == 0);
}
void up (int *sem) {
    int ret=0;
    do {
        int value = load_linked (sem);
        ret = store_conditional (sem, value+1);
    } while (ret == 0);
}
```

Problem 3 (3 points) Given the following NUMA system with 4 dual-socket NUMA nodes, each NUMA node with 8 GB of main memory. Each socket has two cores sharing the access to a per-socket 4 MB cache. Coherence inside NUMA nodes is maintained with a snoopy–based mechanism implementing write–invalidate MSI. Coherence among NUMA nodes is maintained with a directory–based mechanism implementing write–invalidate MSU.



coreX: core X within a socket. 16 cores in total.
socketY: socket Y within NUMA node: each socket with 2 cores, sharing 4 MB cache; MSI snoopy coherence protocol.
NUMAnodeW: NUMA node W; each node with two sockets sharing 8 GB of main memory and a hub/directory to keep coherence among nodes, MSU simplified protocol.
Size of memory and cache lines: 64 bytes.

Part I (1 point): Compute the total number of bits that are necessary **in the whole system** to:

1. Maintain the coherence among NUMA nodes.
   Solution: In each NUMA node the memory is able to store $8GB \div 64$ memory lines. This is $2^3 \times 2^{30} \div 2^6 = 2^{27}$ lines. For each line in memory the directory needs to store 2 bits to keep the state of the line (MSU) and 4 presence (sharers) bits (one per NUMA node); so 6 bits per memory line. In total for the whole system this is $4nodes \times 2^{27} lines/node \times 6bits/line = 3 \times 2^{30}$ bits.
2. Maintain the coherence within NUMA nodes.
   Solution: Each cache memory inside a NUMA node is able to store $4MB \div 64$ memory lines. This is $2^2 \times 2^{20} \div 2^6 = 2^{16}$ lines. For each line in the cache the snoopy needs to store 2 bits to keep the state of the line (MSI). In total for the whole system this is $4nodes \times 2caches/node \times 2^{16} lines/cache \times 2bits/line = 2^{20}$ bits.

Part II (2 points): Assume that the home node for the line containing variable `var` is NUMAnode0, and at a given time there exist 3 clean copies of that line in cache memories: in socket0 in NUMAnode0, in socket0 in NUMAnode1 and in socket0 in NUMAnode2. Considering that the following memory accesses are done one after the other: 1) core2 in NUMAnode0 reads `var`; 2) core0 in NUMAnode3 reads `var`; 3) core0 in NUMAnode3 writes `var`; and 4) core0 in NUMAnode0 writes variable `other`. We ask you to select the sentences that are correct for each one of these 4 memory accesses (for each memory access at least one of the sentences is correct). Each correct selection adds 0.25 points; each wrong selection subtracts 0.17 points; the grade for this part of the problem is always in the range 0–2.

1. When core2 in NUMAnode0 reads variable `var`, which of the following sentences are correct?
   (a) Core2 issues PrRd.
   (b) The snoopy in socket1 issues BusRd on its local bus.
   (c) The snoopy in socket0 observes the BusRd command and places the line on the bus (Flush).
   (d) The hub associated to NUMAnode0 updates the directory for the line containing `var` to indicate that a new copy of the line exists inside NUMAnode0.
   (e) No coherence requests are sent to the rest of the NUMA nodes in the system.

   Solution: True, True, False, False, True
2. Then, when core0 in NUMAnode3 reads variable `var`, which of the following sentences are correct?
   (a) The snoopy in socket0 issues BusRd on its local bus.
   (b) The hub associated to NUMAnode3 finds the closest NUMA node that has a copy of the line and sends a RdReq to that NUMA node.
   (c) The hub of the NUMA node receiving the RdReq reads the line from the cache memory that is storing it.
   (d) NUMAnode3 receives a Dreply command with the line containing variable `var` and stores a copy in its main memory.
   (e) At the end the directory in the home NUMA node is updated so that all bits in the sharers list are set to 1 and the state is kept as S

   Solution: True, False, False, False, True
3. Then, when core0 in NUMAnode3 writes variable `var`, which of the following sentences are correct?
   (a) The snoopy in socket0 of NUMAnode3 issues an Invalidate command on its local bus.
   (b) The hub in NUMAnode3 issues an Invalidate command, going to the home NUMA node.
   (c) The home NUMA node checks if there are copies of the line in other NUMA nodes, sending to each one of them an Invalidate command.
   (d) The state for the line in the caches of the remote nodes receiving the Invalidate command (as well as in the home node) changes from S to M to indicate that the line is modified somewhere else.
   (e) At the end the directory in the home NUMA node only has bit 3 in the sharers list set to 1 and the state set to M.

   Solution: False, False, True, False, True
4. Finally, core0 in NUMAnode0 writes variable `other` (observe that we are not giving any information about the home NUMA node for the line containing this variable). When this memory accesses finishes the state in the directory for the line containing variable `var` is kept to M, but the bit set to 1 in the sharers list moves from position 3 to position 0. Which of the following sentences are correct?
   (a) You should tell me which is the home NUMA node for variable `other` in order to decide if the following sentences are correct or not.
   (b) This state in the directory is not possible since `var` and `other` are different variables.
   (c) Since both variables reside in different cache lines, this state is possible if both variables are mapped in the same cache line entry (cache line replacement, i.e. the line containing `var` is replaced by the line containing `other`).
   (d) This is the typical symptom of false sharing.
   (e) Since variables `var` and `other` are different, there is no need to interchange coherence commands between NUMAnode0 and NUMAnode3.

   Solution: False, False, False, True, False