PAR – 2^{nd} In-Term Exam – Course 2019/20-Q1

December 18^{th}, 2019

Problem 1 (3 points) Assume the following serial code computing a two–dimensional NxN matrix `u`:

```
void compute(int N, double *u) {
    int i, j;
    double tmp;
    for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];                   // element u[i][j]
            u[n*i + j] = tmp/4;                     // element u[i][j]
        }
}
```

The code is parallelised on three processors (P_{0-2}) with the assignment of iterations to processors shown on the left part of Figure 1.

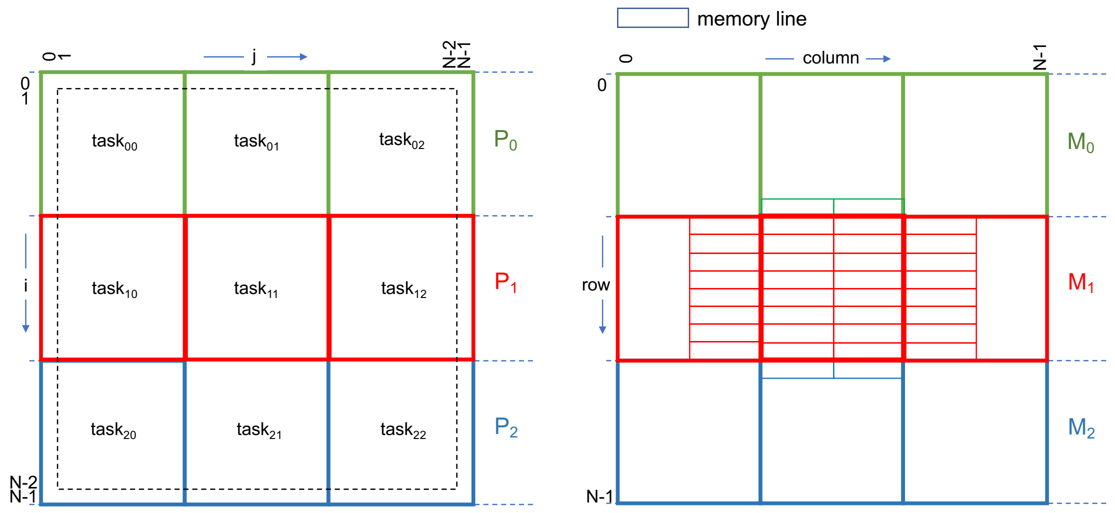


Figure 1: (Left) Assignment of iterations to processors. (Right) Mapping of array elements to memory modules in the NUMA system.

Observe that each processor is assigned the computation of $N/3$ consecutive iterations of the `i` loop (except iterations 0 an N-1), starting with processor P_0 (the number of processors 3 perfectly divides the number of rows and columns N); each processor executes its assigned computation in 3 tasks, each one computing $N/3$ consecutive iterations of the `j` loop. Due to the dependences in this code, you should already know that for example $task_{11}$ can only be executed by P_1 once processor P_0 finishes with the execution of $task_{01}$ and the same processor (P_1) finishes with the execution of $task_{10}$.

The three processors compose a multiprocessor architecture with 3 NUMA nodes sharing the access to main memory, each NUMA node with a single processor P_p, a cache memory (of sufficient size to store all the lines required to execute all tasks) and portion of main memory M_p (p in the range 0–2). Each memory M_p has an associated directory to maintain the coherence at the NUMA level. Each entry in the directory uses 2 bits for the status (M, S and U) and 3 bits in the *sharers list*. The rows of matrix `u` are distributed among NUMA nodes as shown on the right part of Figure 1. In that figure rectangles represent the memory lines that are involved in the computation of $task_{11}$, for a specific case in which N=24 and each cache line is able to host 4 consecutive elements of matrix `u`.

We ask you:

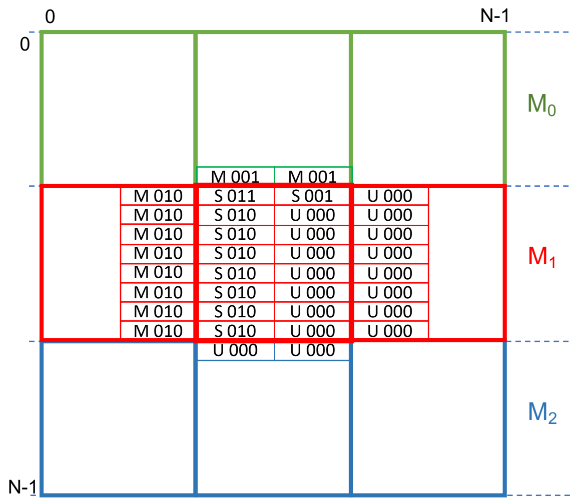
1. Assuming U status for all memory lines at the beginning of the parallel execution, which will be the contents in the directories for the lines shown on the right part of Figure 1 when $task_{11}$ is ready for execution? Use the provided answer sheet for this question, indicating for each memory line the contents of the directory (e.g. S011 meaning that the line is in S state with copies in NUMA nodes 1 and 0).

2. Indicate the sequence of coherence actions (*RdReq, WrReq, UpgrReq, Fetch, Invalidate, Dreply, Ack* or *WriteBack*) that will occur when processor P_1 executes the first iteration in $task_{11}$, i.e. the iteration in the upper left corner of $task_{11}$ on the left part of Figure 1.

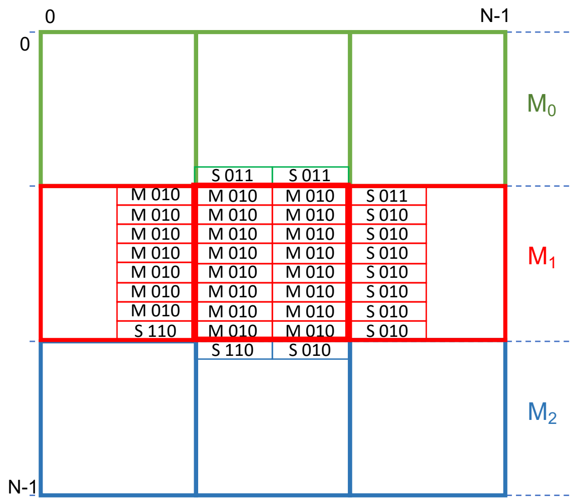
3. Which will be the contents in the directories for the same lines once $task_{11}$ finishes its execution? At that time you should assume that $task_{02}$ and $task_{20}$ have also finished their execution. Use the provided answer sheet for this question.

   Solution: the figure on the left represents the contents in the directories before the execution (question 1.1); the figure on the right after the execution (question 1.3).

Solution for question 1.1 (before execution)



Solution for question 1.3 (after execution)



Regarding question 1.2, processor P_1 first performs several read accesses, one of them to a memory position in state M in memory M_0; to read it, P_0 issues $RdReq_{1\to0}$ to the home node M_0, which responds with the contents of the line and a $Dreply_{0\to1}$. After the computation, P_1 has to write one element for which it is the home node; since the line containing that element is in S state, with copy in P_0's cache, M_1 has to send and $Invalidate_{1\to0}$ command, which is acknowledged with $Ack_{0\to1}$.

Problem 2 (4 points) A *ticket lock* is a lock implemented using two shared counters, `next_ticket` and `now_serving`, both initialised to 0. A thread wanting to acquire the lock uses an atomic operation to fetch the current value of `next_ticket` as its unique sequence number and increments it by 1 to generate the next sequence number. The thread then waits until `now_serving` is equal to its sequence number. Releasing the lock consists on incrementing `now_serving` in order to pass the lock to the next waiting thread.

Given the following data structure and incomplete implementation of the primitives that support the `ticket lock` mechanism:

```
typedef struct {
    int next_ticket;
    int now_serving;
} tTicket_lock;
```

```
void ticket_lock_init (tTicket_lock *lock) {
   lock->now_serving = 0; lock->next_ticket = 0;
}
void ticket_lock_acquire (tTicket_lock *lock) {
   // obtain my unique sequence number from next_ticket
   // generate the next_ticket sequence number
   // wait until my sequence number is equal to now_serving
}
void ticket_lock_release (tTicket_lock *lock) {
   lock->now_serving++;
}
```

We ask:

1. Complete the code for the `ticket_lock_acquire` primitive to be executed on two different platforms that provide the following different atomic operations:

   (a) `fetch_and_inc` atomic operation:

   `int fetch_and_inc(int *addr);`

   Recall that `fetch_and_inc` operation reads the value stored in `addr`, increments it by 1, stores it in `addr` and returns the old value (before doing the increment).

   Solution:

   ```
   void ticket_lock_acquire (tTicket_lock *lock) {
       int my_ticket;
       my_ticket = fetch_and_inc (&lock->next_ticket);
       while (lock->now_serving != my_ticket);
   }
   ```

   (b) `load_linked` and `store_conditional` operations:

   `int load_linked (int *addr);`

   `int store_conditional (int *addr, int value);`

   Recall that `store_conditional` returns 0 in case it fails or 1 otherwise.

   Solution:

   ```
   void ticket_lock_acquire (tTicket_lock *lock) {
      do {
          int my_ticket = load_linked (&lock->next_ticket);
          int last_ticket = my_ticket+1;
      } while (store_conditional (&lock->next_ticket, last_ticket)==0);
      while (lock->now_serving != my_ticket);
   }
   ```

2. Consider the following implementation for the basic lock explained in class using `test-test-and-set`:

   ```
   void lock_init (int *lock)  {
       *lock = 0;
   }
   void lock_acquire (int *lock) {
       do  {
          while (*lock>0);
          int res = test_and_set(lock); // stores 1 at lock address, returns old value
       } while (res > 0);
   }
   void lock_release (int *lock) {
       *lock = 0;
   }
   ```

Given an SMP system with 3 processors (P_0, P_1 and P_2), each with its own cache memory initially empty and a Snoopy-based write-invalidate MSI cache coherency protocol. Fill in the table provided in the answer sheet indicating CPU events (PrRd or PrWr), Bus transactions (BusRd, BusRdX, BusUpgr or Flush) and state of the line cache (M, S or I) in each cache memory after the access to memory, assuming the indicated sequence of instructions. There are three threads, each one executing on a different processor (T_i indicates that thread i executes on processor P_i). The three threads try to acquire the lock at almost the same time, following the order T_0, T_1, T_2 and succeed in acquiring it in the same order.

Solution for Problem 2.2

		P0			P1			P2			
steps	instructions	CPU event	Bus transaction	cache line status	CPU event	Bus transaction	cache line status	CPU event	Bus transaction	cache line status	lock
	Initially			*I*			*I*			*I*	
T0 tries to acquire lock	load lock	PrRd	BusRd	S			I			I	0
T1 tries to acquire lock	load lock			S	PrRd	BusRd	S			I	0
T2 tries to acquire lock	load lock			S			S	PrRd	BusRd	S	0
T0 adquires lock	t&s lock	PrWr	BusUpgr	M			I			I	1
T1 tries to acquire lock	t&s lock		Flush	I	PrWr	BusRdX	M			I	1
T2 tries to acquire lock	t&s lock			I		Flush	I	PrWr	BusRdX	M	1
T1 tries to acquire lock	load lock			I	PrRd	BusRd	S		Flush	S	1
T2 tries to acquire lock	load lock			I			S	PrRd	-	S	1
T0 release lock	store lock	PrWr	BusRdX	M			I			I	0
T1 tries to acquire lock	load lock		Flush	S	PrRd	BusRd	S			I	0
T2 tries to acquire lock	load lock			S			S	PrRd	BusRd	S	0
T1 adquires lock	t&s lock			I	PrWr	BusUpgr	M			I	1
T2 tries to acquire lock	t&s lock			I		Flush	I	PrWr	BusRdX	M	1
T1 release lock	store lock			I	PrWr	BusRdX	M		Flush	I	0
T2 tries to acquire lock	load lock			I		Flush	S	PrRd	BusRd	S	0
T2 adquires lock	t&s lock			I			I	PrWr	BusUpgr	M	1
T2 release lock	store lock			I			I	PrWr	-	M	0

3. Compare the *ticket lock*, implemented with `fetch_and_inc`, with the basic lock, implemented as shown in the previous question using `test-test-and-set`, in terms of coherence traffic and assuming the same scenario described in the previous question. Hint: Take a look at the number of writes to memory in the scenario previously proposed.

   Solution: The basic lock even optimized with a test-test-and-set technique, in the worst case where the first test allows to continue with the lock, the test-and-set could fail, generating more than one invalidate transaction per thread on the lock variable. The ticket lock mechanism only generates one invalidate transaction (when obtaining the ticket number) per thread.

Problem 3 (3 points)

We have the parallel code shown in the following code excerpt:

```
int a[N], b[N], c[N];
int i;
...
#pragma omp parallel
#pragma omp single
{
  #pragma omp taskloop
  for (i = 0, i < N; i++) {                        // Initialization
    a[i] = i;
    b[i] = i*i;
  }
  for (int iter = 0; iter < num_iters; iter++) {   // Computation
    #pragma omp taskloop
    for (i = 0, i < N; i++)
      c[i] = foo(a[i], b[i]);

    #pragma omp taskloop
    for (i = 0, i < N; i++) {
      a[i] = goo(c[i]);
      b[i] = hoo(a[i]);
    }
  }
}
...
```

Since the computation part is repeteated `num_iters` times we want to exploit locality in all levels of the memory hierarchy in order to speed up the parallel execution of code. The first time a memory location is accessed by a thread, the operating system will assign memory in the NUMA node where the thread that is doing such first access is being executed (provided there is free space). This can help us get memory for each thread within the memory of the NUMA node where it is executed. Therefore, accesses to memory will be faster if we manage to *have a thread execute the very same iterations in all three loops controlled by variable* `i`. And locality will be exploited by implementing a *block data decomposition.*

1. (1.5 points) Given these indications above, **we ask** you to write a faster parallel code for both the `initialization` and `computation` stages using the appropriate OpenMP pragmas and invocations to intrinsic functions, assuming the following constraints: 1) you **cannot** make use of the `for` work–sharing construct in OpenMP to distribute work among threads; 2) you **cannot** assume that the number of threads evenly divides the problem size `N`; 3) physical memory has not been assigned yet by the time we reach the initialization loop; 4) parallelization overheads should always be kept as low as possible (due to thread/task creation, synchronization, load imbalance, ...).

Solution:

We need to make sure that a thread executes the very same iterations for all the executions of each loop, both in the initialization and computation stages.

```
#define N ...
 int i;
 ...
 #pragma omp parallel private(i)
 {
   int id        = omp_get_thread_num();
   int howmany   = omp_get_num_threads();
   int basesize  = N / howmany;
   int reminder  = N % howmany;
   int extra     = id < reminder;
   int extraprev= extra ? id : reminder;
   int lb        = id * basesize + extraprev;   /* Loop lower bound */
   int ub        = lb + basesize + extra;       /* Loop upper bound */

   for (i = lb, i < ub; i++) {                  /* Initialization */
     a[i] = i;
     b[i] = i*i;
   }

   for (int iter = 0; iter < num_iters; iter++) {   /* Computation */
     for (i = lb, i < ub; i++) {
       c[i] = foo(a[i], b[i]);
     }
     for (i = lb, i < ub; i++) {  /* This loop could be fused with the one
                                     above to have a single inner loop */
       a[i] = goo(c[i]);
       b[i] = hoo(a[i]);
     }
   }

 }
 ...
```

After the previous code finishes its computation the code continues with the following function call:

```
 ...
 final_processing (a, b, c);
 ...
```

where function `final_processing` is defined as follows

```
 void final_processing (int *a, int *b, int *c) {
   for (int i = 0; i < N; i++) c[i] = zoo(a[i], b[i], c[i]);
 }
```

Contrary to the `computation` stage studied above, the loop that appears in the `final_processing` stage is executed only once. However, we need to solve another problem, namely, that function `zoo` presents a highly variable execution time depending on the actual numerical values received as inputs. Consequently, the block distribution used in the previous stages could easily cause load imbalance in this final processing stage. Additionally, assume cache lines are 64 bytes long and integers occupy 4 bytes.

2. (1.5 points) **We ask you** to provide an efficient parallelization of the loop above. As before, you **are not allowed** to make use of the `for` work–sharing construct in OpenMP to distribute work among threads.

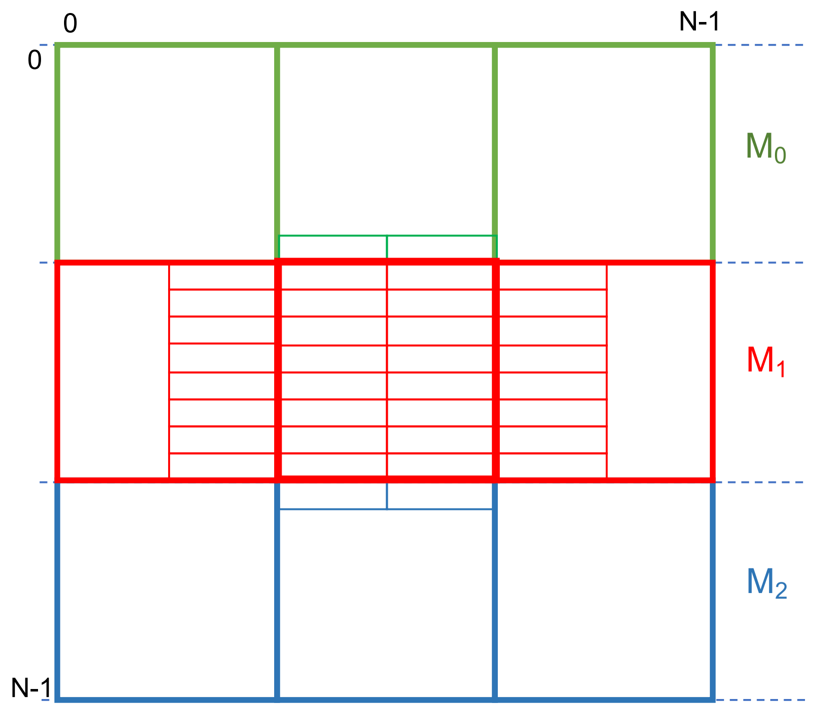
   Solution:

   We are interested in avoiding load imbalance while avoiding *false sharing* in the accesses to vector `c`. For this, we parallelize the loop following a *block-cyclic geometric data decomposition* for the *output* vector `c`. The block size should allow the processor to use all the elements in a cache line, benefiting from spatial locality and avoiding false sharing. To implement the block-cyclic decomposition we need two nested loops, with an outer loop jumping the blocks cyclically and an inner loop traversing all the elements in each block, as follows:

```
#define min(a,b) ( (a) > (b) ? (b) : (a) )

#define CACHE_LINE_SIZE 64

    ...
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        /* Computation of the block size */
        int block_size = CACHE_LINE_SIZE / sizeof(int);

        /* Loop used to jump blocks cyclically */
        for (int ii = id*block_size; ii < N; ii+=(howmany*block_size))
            /* Loop used to traverse each block */
            for (int i = ii; i < min(N, ii+block_size); i++)
                c[i] = zoo(a[i], b[i], c[i]);
    }
    ...
```

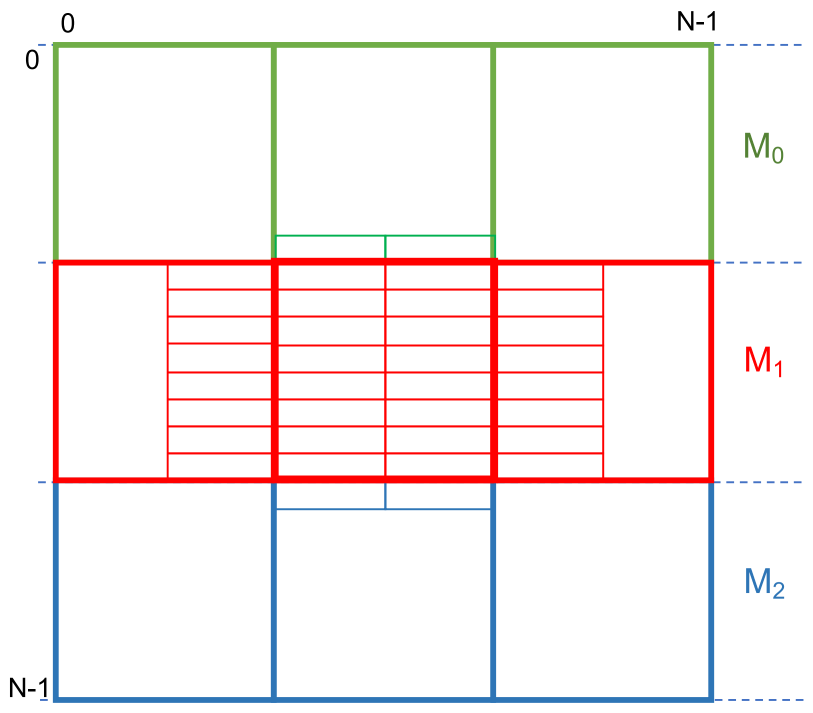
Student name: ……………………………………………………………………………………

Answer sheet for question 1.1 (before execution)



▭ Directory entry for memory line (for example S011 to Indicate line in state S with copies in nodes 1 and 0.

Answer sheet for question 1.3 (after execution)



▭ Directory entry for memory line (for example S011 to Indicate line in state S with copies in nodes 1 and 0.

Student name: ……………………………………………………………………………………

		P0			P1			P2			
steps	instructions	CPU event	Bus transaction	cache line status	CPU event	Bus transaction	cache line status	CPU event	Bus transaction	cache line status	lock
	Initially			I			I			I	0
T0 tries to acquire lock	load lock										
T1 tries to acquire lock	load lock										
T2 tries to acquire lock	load lock										
T0 adquires lock	t&s lock										
T1 tries to acquire lock	t&s lock										
T2 tries to acquire lock	t&s lock										
T1 tries to acquire lock	load lock										
T2 tries to acquire lock	load lock										
T0 release lock	store lock										
T1 tries to acquire lock	load lock										
T2 tries to acquire lock	load lock										
T1 adquires lock	t&s lock										
T2 tries to acquire lock	t&s lock										
T1 release lock	store lock										
T2 tries to acquire lock	load lock										
T2 adquires lock	t&s lock										
T2 release lock	store lock										