

**PAR – 2<sup>nd</sup> In-Term Exam – Course 2018/19-Q1**  
**December 19th, 2018**

**Problem 1** (3 points) Given the following OpenMP parallel function that builds some sort of histogram `h` for the elements of vector `v` received as argument:

```
#define SIZE1 131072
#define SIZE2 128
#define MAX 1024
#define CACHE_LINE_SIZE 64 // size of the cache line in bytes
#define INT_SIZE 4          // size of int data type in bytes
#define LOCKT_SIZE 4        // size of omp_lock_t data type in bytes

typedef int tVector[SIZE1];

typedef struct {
    int bin;
    int nelem;
    omp_lock_t lock;
} tBin;
typedef tBin tHisto[SIZE2];

void build_histogram (tVector v, tHisto h) {
    #pragma omp parallel
    #pragma omp single
    {
        for (int k=0; k<SIZE1; k++)
            #pragma omp task
            {
                int ind = v[k] % SIZE2;
                omp_set_lock(&h[ind].lock);
                if (h[ind].nelem < MAX) {
                    h[ind].bin += v[k];
                    h[ind].nelem++;
                }
                omp_unset_lock(&h[ind].lock);
            }
    }
}
```

Each entry of the histogram (`tHisto`) contains information about the values accumulated in each bin of the histogram (field `bin`) and the number of elements in each bin (field `nelem`). The program only accumulates the first `MAX` elements that go to each bin. The implementation makes use of locks (field `lock`) to protect the appropriate access to each element of type `tBin`; you can assume that locks are appropriately created and destroyed in the main program.

1. Assuming that the granularity for the tasks generated is not an issue, identify the two main issues that clearly degrade the performance during the execution of the parallel region in the histogram computation. Identify which parts of the code and/or data accesses are the responsible for that performance degradation.

**Solution:** The two main issues affecting parallel performance during the execution of the parallel region are:

- Synchronization overheads due to the locking scheme applied to protect the access to the histogram `h`, checking for the value of `h[ind].nelem`.

- False sharing overheads due to the contiguous storage in the same line of multiple elements of the histogram h.
2. Write a new version of the parallel code that follows **the same task decomposition** and solves the two previously identified performance issues.

**Solution:** In the proposed solution, the synchronization overheads are minimized by using the *test-and-set* technique to only acquire the lock when there are chances that the condition is met; false sharing overheads are solved by padding the definition of the `tBin` data type, so that each element of the histogram occupies a complete cache line.

```
#define SIZE1 131072
#define SIZE2 128
#define MAX 1024
#define CACHE_LINE_SIZE 64 // size of the cache line in bytes
#define INT_SIZE 4 // size of int data type in bytes
#define LOCKT_SIZE 4 // size of omp_lock_t data type in bytes

typedef int tVector[SIZE1];

typedef struct {
    int bin;
    int nelem;
    omp_lock_t lock;
    char padding[CACHE_LINE_SIZE-INT_SIZE*2-LOCKT_SIZE];
} tBin;

typedef tBin tHisto[SIZE2];

void build_histogram (tVector v, tHisto h) {
    #pragma omp parallel
    #pragma omp single
    {
        for (int k=0; k<SIZE1; k++)
            #pragma omp task
            {
                int ind = v[k] % SIZE2;
                if (h[ind].nelem < MAX) {
                    omp_set_lock(&h[ind].lock);
                    if (h[ind].nelem < MAX) {
                        h[ind].bin += v[k];
                        h[ind].nelem++;
                    }
                    omp_unset_lock(&h[ind].lock);
                }
            }
    }
}
```

3. The implementation of the `omp_set_lock` and `omp_unset_lock` function in our experimental system is as follows:

```
typedef int omp_lock_t;
// *lock is initialized to 0
void omp_set_lock (omp_lock_t *lock) {
    while (compare_and_swap(lock, 0, 1)==1);
}
void omp_unset_lock (omp_lock_t *lock) {
    *lock=0;
}
```

where the atomic function `compare_and_swap (omp_lock_t *p, int old, int new)` works as follows: it compares the memory location pointed by `p` with `old`, and if they are the same, the content of this memory location is set to `new`, otherwise is unchanged. The function returns the original value in the memory location pointed by `p`. **We ask you** to write an implementation for the atomic function `compare_and_swap` that makes use of the built-in *load-linked* and *store-conditional* operations. The *load-linked* operation is performed by a call to `load_linked(int *p)` and the *store-conditional* operation is performed by a call to `int store_cond(int *p, int value)`.

**Solution:**

```
int compare_and_swap (int *p, int old, int new) {
    int ret, value;
    do {
        ret = load_linked(p);
        if (old == ret)
            value = new;
        else
            value = ret;
    } while (store_cond(p, value) == 0);
    return ret;
}
```

**Problem 2** (4 points) The following code excerpt implements the multiplication of the lower triangular part of a dense matrix `T` times a vector `X`, accumulating the result in vector `Y`:

```
/* Y += T * X */
...
for (i=0; i<N; i++) {
    for (j=0; j<=i; j++) {
        Y[i] += T[i][j] * X[j];
    }
}
...
```

We want to parallelise the code using appropriate OpenMP pragmas and invocations to intrinsic functions. You can NOT make use of the `for` work-sharing construct in OpenMP to distribute work among threads. You can assume that the number of threads evenly divides `N`. **We ask you** to:

1. Implement a parallel version that obeys to an *input block geometric data decomposition* of the input vector `X`.

**Solution:**

This data decomposition requires synchronization in the update to the output vector `Y`. In order to reduce synchronizations to only one at the very end of the computation of the partial result computed by each thread, we use a temporary scalar variable `tmp` to accumulate results within the inner loop. [This technique, which is oftentimes applied automatically by a compiler is known as *scalar replacement* and aims at replacing array references in the inner loop with a register reference.]

Although it was not requested in the statement of this exercise, for the sake of completeness we also show the code which handles the case where the number of threads does not divide the problem size exactly. In the solution below this would be the one generated using conditional compilation if the symbol `EXACT_DIV` was not defined at compilation time.

```
#define min(a,b)      ( (a) > (b) ? (b) : (a) )
...
#pragma omp parallel private(i,j)
{
```

```

    int id          = omp_get_thread_num();
    int howmany     = omp_get_num_threads();
    int basesize    = N / howmany;
#ifdef EXACT_DIV
    int reminder    = N % howmany;
    int extra       = id < reminder;
    int extraprev   = extra ? id : reminder;
#else
    int extra       = 0;
    int extraprev   = 0;
#endif
    int lb          = id * basesize + extraprev; /* Inner loop lower bound */
    int ub          = lb + basesize + extra;    /* Inner loop upper bound */

    for (i = 0; i < N; i++) {
        double tmp = 0.0; /* Scalar replacement */
        for (j = lb; j <= min( i , ub-1 ); j++) {
            tmp += T[i][j] * X[j];
        }
        #pragma omp atomic
        Y[i] += tmp;
    }
}
...

```

2. Implement a new parallel version that obeys to an *output block geometric data decomposition* of the output vector Y.

**Solution:**

This solution does not require synchronizations, but suffers from severe load unbalance.

```

...
#pragma omp parallel private(i,j)
{
    int id          = omp_get_thread_num();
    int howmany     = omp_get_num_threads();
    int basesize    = N / howmany;
#ifdef EXACT_DIV
    int reminder    = N % howmany;
    int extra       = id < reminder;
    int extraprev   = extra ? id : reminder;
#else
    int extra       = 0;
    int extraprev   = 0;
#endif
    int lb          = id * basesize + extraprev; /* Outer loop lower bound */
    int ub          = lb + basesize + extra;    /* Outer loop upper bound */

    for (i = lb; i < ub; i++) {
        double tmp = 0.0; /* Scalar replacement */
        for (j = 0; j <= i; j++) {
            tmp += T[i][j] * X[j];
        }
        Y[i] += tmp;
    }
}
...

```

3. Improve the previous last parallel version by implementing an enhanced parallel version which improves load balancing. This new version should also exploit spatial data locality and avoid false sharing.

**Solution:**

We implement a *block-cyclic geometric data decomposition of the output vector*.

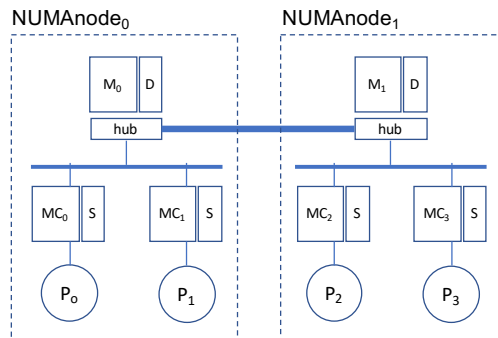
```
#define min(a,b)      ( (a) > (b) ? (b) : (a) )
#define NUM_ELEMENTS_PER_CACHE_LINE 8
...

#pragma omp parallel private(i,j)
{
    int id          = omp_get_thread_num();
    int howmany     = omp_get_num_threads();
    int block_size  = NUM_ELEMENTS_PER_CACHE_LINE;
    int lb          = id*block_size;          /* Outer loop lower bound */
    int step        = howmany*block_size;     /* Outer loop step */

    /* Loop jumping blocks cyclically */
    for (int ii = lb; ii < N; ii+=step)
        /* Loop traversing each block */
        for (i = ii; i < min( N, ii+block_size ); i++) {
            double tmp = 0.0;                /* Scalar replacement */
            for (j = 0; j <= i; j++) {
                tmp += T[i][j] * X[j];
            }
            Y[i] += tmp;
        }
}
...
```

**Note:** If needed, you can assume that the number consecutive of elements of the data structures that fit into a cache line is NUM\_ELEMENTS\_PER\_CACHE\_LINE.

**Problem 3** (3 points) Consider the following parallel architecture composed of two NUMA nodes, each with its own main memory, directory to keep coherence between NUMA nodes (write-invalidate MSU) and two processors with their own cache memory and snoop to keep coherence within each node (write-invalidate MSI). For the purposes of this problem, you can assume: 1) infinite sizes for both main and cache memories; and 2) line size stores 16 consecutive integer values.



**Legend:**

NUMAnode<sub>i</sub>: NUMA node  $i$  with two processors  
 $M_i$ : main memory for NUMAnode <sub>$i$</sub>   
 $D$ : directory associated to  $M$   
hub: interconnect between NUMA nodes  
 $P_j$ : processor  $j$  inside NUMA node  
 $MC_j$ : cache memory local to processor  $P_j$   
 $S$ : snoop associated to  $MC$

**Coherence commands:**

- Snoopy: BusRd <sub>$j$</sub> , BusRdX <sub>$j$</sub>  and Flush <sub>$j$</sub> , being  $j$  the snoop/cache number doing the action
- Hub/directoty: RdReq <sub>$i \rightarrow j$</sub> , WrReq <sub>$i \rightarrow j$</sub> , Dreply <sub>$i \rightarrow j$</sub> , Fetch <sub>$i \rightarrow j$</sub> , Invalidate <sub>$i \rightarrow j$</sub>  and WriteBack <sub>$i \rightarrow j$</sub> , from NUMAnode <sub>$i$</sub>  to NUMAnode <sub>$j$</sub>

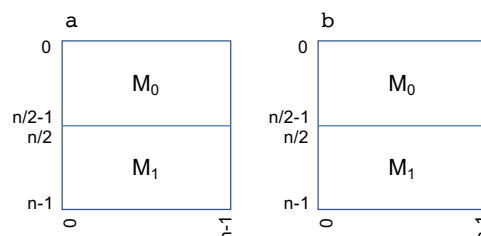
Also consider the following skeleton for a parallel program accessing to matrices  $a$  and  $b$ :

```
#define CACHE_LINE 16 // number of integers in a cache line
#define n 64
int a[n][n], b[n][n];
...
// initialization loop
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++) {
        a[i][j] = ...;
        b[i][j] = ...;
    }

// compute loop 1: can only be parallelized with 2 processors, only dimension i
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        a[i][j] = foo(a[i][j], b[i][j]);

// compute loop 2: can be parallelized with 4 processors, both dimensions i and j
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        b[i][j] = goo(b[i][j], a[j][i]);
```

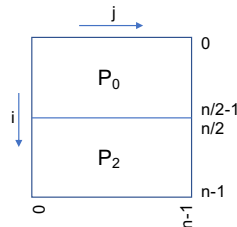
Functions  $foo$  and  $goo$  do not have any lateral effect on memory (i.e. don't modify any value in memory), variables  $i$  and  $j$  are stored in registers and elements of matrices are stored by rows. Assuming that after the initialization loop matrices  $a$  and  $b$  are mapped to NUMA nodes as follows with NO copies in cache (i.e. all of them in U state in the corresponding directory):



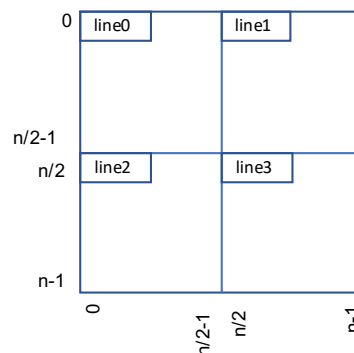
1. How many entries in the directory associated to each main memory  $M_0$  and  $M_1$  are occupied by matrices  $a$  and  $b$ ? How many bits are necessary in each directory entry to keep coherence information? Clearly indicate which information is stored in these bits.

**Solution:** Each row occupies 4 cache lines (64 columns per row divided by 16 elements per line); therefore, each matrix occupies 64 rows times 4 lines per row, so in total 256 lines. In the directory of each memory 128 entries are taken by each matrix, so in total **256 entries in  $M_0$  and 256 entries in  $M_1$** . Each entry has 2 state bits (enough for three states M, S and U) and 2 bits in the sharers list (one for each NUMA node); so in total **4 bits** per directory entry.

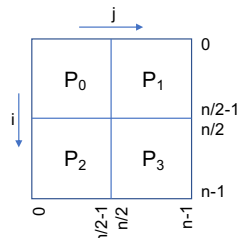
2. Assume that only the  $i$  loop can be parallelized in *compute loop 1*, using processors  $P_0$  and  $P_2$ , as shown in the iteration space below:



During the execution of *compute loop 1*, which coherence commands are placed on the bus by the snoopies and which coherence commands are exchanged between NUMA nodes to maintain the coherence for the 4 memory lines of matrices a and b shown below? After the execution of *compute loop 1*, which is going to be the status for these 4 lines of matrices a and b both in the cache memories and in the directories? Fill in the table in the solutions page to answer this question.



3. Assume that both loops can be parallelized in *compute loop 2*, using all processors in the system as shown in the iteration space below:



During the execution of *compute loop 2*, which coherence commands are placed on the bus by the snoopies and which coherence commands are exchanged between NUMA nodes to maintain the coherence for the same 4 memory lines of matrices a and b? Please note the transposed access to matrix a in this loop. After the execution of *compute loop 2*, which is going to be the status for these 4 lines of matrices a and b both in the cache memories and in the directories? Fill in the table in the solutions page to answer this question.

### Solution for questions 1.2 and 1.3:

		NUMA node 0					NUMA node 1				
		Bus command	MC0 status	MC1 status	NUMA commands	Directory entry	Bus command	MC2 status	MC3 status	NUMA commands	Directory entry
compute 1 loop	line 0 for a	BusRd0, BusRdX0	S -> M			01 M					
	line 1 for a	BusRd0, BusRdX0	S -> M			01 M					
	line 2 for a						BusRd2, BusRdX2	S -> M			10 M
	line 3 for a						BusRd2, BusRdX2	S -> M			10 M
	line 0 for b	BusRd0	S			01 S					
	line 1 for b	BusRd0	S			01 S					
	line 2 for b						BusRd2	S			10 S
	line 3 for b						BusRd2	S			10 S
compute 2 loop	line 0 for a										
	line 1 for a	(*) Flush0	S		Dreplay0->1	11 S	BusRd2	S		RdReq1->0	
	line 2 for a	BusRd1		S	RdReq0->1		(*) Flush2	S		Dreplay1->0	11 S
	line 3 for a						BusRd3, Flush2	S	S		10 S
	line 0 for b	BusRdX0	M			01 M					
	line 1 for b	BusRd1, BusRdX1	I	M		01 M					
	line 2 for b						BusRdX2	M			10 M
	line 3 for b						BusRd3, BusRdX3	I	M		10 M

(\*) hub injects a BusRd after receiving RdReq from the other node

Note: We are not including in the table the actions in the directory that are generated when the local and the home node are the same and there are no other remote nodes with a copy of the line being accessed.



Student name: .....

Table to be used to deliver your solution to **Problem 3**

[illegible]