PAR – 2^{nd} In-Term Exam – Course 2017/18-Q1
January 10th, 2018

Problem 1 (4 points) Consider the following sequential C code for reversing the order of a vector of data items and doing some computation on them:

```
int * array; // pointer to start of array
int N; // length of array, assumed to be multiple of the number of processors

void swap (int i, int j) {
    int tmp = array[i]; array[i] = array[j]; array[j] = tmp;
}

void reverse ( ) {
    for (int i = 0; i < N/2; i++) swap (i, N - 1 - i);
}

void compute ( ) {
    for (int i = 0; i < N; i++) array[i] = foo(array[i], i);
}

void main( ) {
    reverse( );
    compute( );
}
```

We ask you:

1. Implement, using OpenMP, a parallel version for function `compute` that follows a *block-cyclic geometric data decomposition* for the *output* vector `array`. Decide the minimum value for the block size that better exploits data locality (assuming that cache lines are 32 bytes long and integers occupy 4 bytes). **Important:** You are **NOT ALLOWED to use the `for`** clause in your parallel version.

   Solution:

   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 a nested loop, with an outer loop jumping the blocks cyclically and an inner loop traversing all the elements in each block, as follows:

```
#define CACHE_LINE_SIZE 32

void compute ( ) {
    #pragma omp parallel
    {
        int P = omp_get_num_threads();
        int id = omp_get_thread_num();
        // computation of the block size for the block-cyclic decomposition
        int block_size = CACHE_LINE_SIZE / sizeof(int)

        // loop jumping blocks cyclically
        for (int ii = id*block_size; ii < N; ii+=(P*block_size))
            // loop traversing each block
            for (int i = ii; ii < min(N, ii+block_size); i++)
                array[i] = foo(array[i], i);
    }
}
```

2. Draw the *geometric data decomposition* for the *output* vector `array` that would be implemented with the (incomplete) parallel version for function `reverse` that is given below:

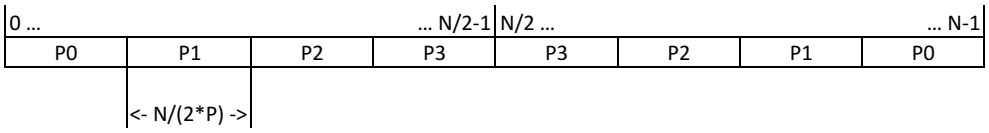
```
void reverse ( ) {
    int P = ...;  // number of threads executing this function
    int id = ...; // identifier of the thread executing this instance (0 .. P-1)

    int segmentLength = N / ( 2 * P );
    int segmentStart = id * segmentLength;
    for (int i = segmentStart; i < segmentStart + segmentLength; i++)
        swap (i , N - 1 - i);
}
```

Complete the parallel code with the appropriate OpenMP pragmas and invocations to intrinsic functions.

Solution:

According to the owner computes rule (i.e. the output is computed/updated by the thread to which the output data is assigned) the data decomposition is:



The complete parallel code is the following:

```
void reverse ( ) {
    #pragma omp parallel
    {
        int P = omp_get_mum_threads();
        int id = omp_get_thread_num();

        int segmentLength = N / ( 2 * P );
        int segmentStart = id * segmentLength;
        for (int i = segmentStart; i < segmentStart + segmentLength; i++)
            swap (i , N - 1 - i);
    }
}
```

3. Finally, implement a new parallel version for function `compute` that follows the same *output geometric data decomposition* that has been specified above in function `reverse`.

Solution:

In order to implement the above data decomposition, the easiest way is to split the iteration space in two halves: the first assigning half of the iterations to threads 0..P-1 and the second assigning the other half of the iterations to the threads in reverse order (P-1..0).

```
void compute ( ) {
    #pragma omp parallel
    {
        int P = omp_get_mum_threads();
        int id = omp_get_thread_num();
        int segmentLength = N / ( 2 * P );

        // Loop traversing the first half of the vector
        int segmentStart = id * segmentLength;
```

```
        for (int i = segmentStart; i < segmentStart + segmentLength; i++)
                array[i] = foo(array[i], i);

        // Loop traversing the second half of the vector
        int segmentStart = N - ((id+1) * segmentLength);
                          // or also valid N/2 + (P-1-id) * segmentLength
        for (int i = segmentStart; i < segmentStart + segmentLength; i++)
                array[i] = foo(array[i], i);
    }
}
```

A simpler solution to write fuses both loops into a single one, with two calls to function `foo`, following the same pattern of invocations that was done in `traverse`.

```
void compute ( ) {
    #pragma omp parallel
    {
        int P = omp_get_mum_threads();
        int id = omp_get_thread_num();
        int segmentLength = N / ( 2 * P );

        // Loop traversing the first half of the vector
        int segmentStart = id * segmentLength;
        for (int i = segmentStart; i < segmentStart + segmentLength; i++) {
                array[i] = foo(array[i], i);
                array[N - 1- i] = foo(array[N - 1- i], N - 1- i);
        }
    }
}
```

Problem 2 (2 points) Given the following sequential code in C that counts the number of times each key in a set of keys (contained in vector `keys`) appears in a vector `DBin`:

```
#define DBsize 1048576
#define nkeys 128 // much larger than the number of processors

int main() {
    double DBin[DBsize];
    double keys[nkeys];
    unsigned int counter[nkeys];

    getkeys(keys, nkeys);      // get keys
    initialize(DBin, DBsize);  // initialize elements in DBin

    #pragma omp parallel
    for (unsigned int i = 0; i < DBsize; i++)
        #pragma omp for schedule(static, 1)
        for (unsigned int k = 0; k < nkeys; k++)
            if (DBin[i] == keys[k]) counter[k]++;
}
```

Does the proposed parallelisation strategy suffer from false sharing? Justify your answer and in affirmative case propose two alternative solutions that eliminate the false sharing issue:

1. Solution 1: only changing the `schedule` clause.
2. Solution 2: changing the declaration(s) of data structure(s) and its/their access in the program.

Notes: 1) Assume that vectors `keys` and `counter` start at the beginning of a cache line; 2) cache lines are 32 bytes long; and 3) `double` and `int` data elements occupy 8 and 4 bytes, respectively.

Solution: Yes, it suffers false sharing since a chunk size equal to 1 implies that several threads can access consecutive positions of vector `counter`. Since several consecutive positions fall in the same cache line, and several threads may write into it, this can cause false sharing.

Being 32 bytes the cache line size, and 4 bytes the size of an `unsigned integer`, this implies that up to 8 elements of vector `counter` can fall in the same cache line.

```
#define NUM_COUNTER_ELEMS_PER_CACHE_LINE 8
```

- Solution 1: Change the *chunk* size.

  ```
  ... schedule(static,NUM_COUNTER_ELEMS_PER_CACHE_LINE)
  ```

- Solution 2: Introduce *padding* in `counter`.

  ```
  ...
  unsigned int counter[nkeys][NUM_COUNTER_ELEMS_PER_CACHE_LINE];
  ...
  if (DBin[i] == keys[k]) counter[k][0]++;
  ...
  ```

Problem 3 (4 points) Given a NUMA system with 2 nodes, each node with 2 cores (each with its own local cache memory), which implements a write–invalidate coherence scheme based on directories among NUMA Nodes and MSI snoopy within each NUMA Node. Assume that the following code is executed on 3 cores of that system:

```
// In cores 0 and 1 (NUMA Node 0)          // In core 2 (NUMA Node 1)
count = 0;
flag = 1;                                  ...
#pragma omp parallel for                   tmp = ll(flag);
for (i = 0; i < 4; i++)              lock: while (!(sc(1, flag) && tmp==0))
    #pragma omp atomic                         tmp = ll(flag);
    count++;                               ...
flag = 0;
```

Assumptions: 1) variables `i` and `tmp` are stored in two registers in the register file of each core; variables `count` and `flag` are stored in the same memory line, mapped in NUMA Node 1; 2) local cache memories are initially empty; 3) the atomic pragma does not imply additional memory accesses; and 4) the execution of `ll` (load linked) implies a $BusRd$ and $RdReq$, `sc` (store conditional) implies a single $BusRdX$ and $WrReq$, and the increment ++ implies a $BusRdX$ and $WrReq$, if any of them is necessary.

We ask you to complete the following table with the actions that occur (*NUMA transactions* and *Bus transactions*) and changes in the state of caches and directories to maintain memory coherency, assuming the temporal ordering of memory instructions shown in the same table.

Solution: Since both variables count and flag reside in the same line, there is a false sharing problem:

- the state of the line switches between M and I alternatively in the two caches; the line is only in S state when a read in one thread is done immediately after the write on the other.
- the state of the line in the directory is kept M all the time switching the list of sharers between 01 and 10, depending on which node has the line in M state in its cache; the line is only in S state in the directory when two caches have the copy also in S state.
- the coherence commands between NUMA nodes is shown in the table below, with the pattern WrRq-Dreply followed by Dreply-Fetch/invalidate that is symptom of the false sharing problem that is happening.

Time	NUMA Node 0						NUMA Node 1							
	Core 0		Core 1		Bus transactions	NUMA transactions	Core 2		Core 3	Bus transactions	NUMA transactions			
	Inst.	Cache state	Inst.	Cache state			Inst.	Cache state	Not used			Directory state	Sharers list	
		flag/count		flag/count				flag/count				flag / count	1	0
0	**count=0**	M		--	BusRdX	WrReq		--		--	Dreply	M	0	1
1	**Flag=1**	M		--	--	--		--		--	--	M	0	1
2	**count++**	M		--	--	--		--		--	--	M	0	1
3		I	**count++**	M	BusRdX/Flush	--		--		--	--	M	0	1
4		I		S	BusRd/Flush	Dreply	**ll flag**	S		BusRd	Fetch	S	1	1
5	**count++**	M		I	BusRdX	WrReq		I		BusRdX	Dreply (*)	M	0	1
6		I		I	a) BusRd/Flush + b) BusRdX	a) Dreply + b) --	**sc 1, flag**	M		BusRdX	a) Fetch + b) Invalidate	M	1	0
7		I		I	--	--	**ll flag**	M		--	--	M	1	0
8		I	**count++**	M	BusRdX	WrReq		I		BusRdX/Flush	Dreply	M	0	1
9		I		I	a) BusRd/Flush + b) BusRdX	a) Dreply + b) --	**sc 1, flag**	M		BusRdX	a) Fetch + b) Invalidate	M	1	0
10	**flag=0**	M		I	BusRdX	WrReq		I		BusRdX/Flush	Dreply	M	0	1
11		S		I	BusRd/Flush	Dreply	**ll flag**	S		BusRd	Fetch	S	1	1
12		I		I	BusRdX	--	**sc 1, flag**	M		BusRdX	Invalidate	M	1	0

(*) Assuming that memory always provides line in S

SURNAME:

NAME:

Time	NUMA Node 0						NUMA Node 1							
	Core 0		Core 1		Bus transactions	NUMA transactions	Core 2		Core 3	Bus transactions	NUMA transactions			
	Inst.	Cache state flag/count	Inst.	Cache state flag/count			Inst.	Cache state flag/count	Not used			Directory state flag / count	Sharers list 1	0
0	**count=0**													
1	**Flag=1**													
2	**count++**													
3			**count++**											
4							**ll flag**							
5	**count++**													
6							**sc 1, flag**							
7							**ll flag**							
8			**count++**											
9							**sc 1, flag**							
10	**flag=0**													
11							**ll flag**							
12							**sc 1, flag**							

(1) Cache line state: M (modified), S (shared) or I (invalid)

(2) Coherence commands inside NUMA node: BusRd, BusRdX and Flush

(3): Line state in node memory: M (modified), S (shared) or U (uncached)

(4) Coherence commands between NUMA nodes: RdReq, WrReq, Dreply, Fetch and Invalidate