PAR – 2^{nd} In-Term Exam – Course 16-17Q1
December 21st, 2016

Problem 1 (4 points) Consider the following main program that calculates the power of any number using an iterative function and a recursive function.

```
#include <stdio.h>
#define MIN_POWER 10

long int getPower_iterative(int b, int p) {
   long int result=1;
   for(int i=0; i<p; i++)
      result = result * b;
   return result;
}

long int getPower_recursive(int b,int p) {
    long int result;
    if(p<MIN_POWER)
       result = getPower_iterative(b,p);
    else
       result = getPower_recursive(b ,p/2) *
                getPower_recursive(b ,p-p/2);

    return result;
}

int main() {
    int base, power;
    long int result;
    ...
    result = getPower_recursive(base, power);
    ...
    result = getPower_iterative(base, power);
    ...
    return 0;
}
```

Note: Each question is independent. **We ask** you:

1. Write two significative different OpenMP versions that implement an iterative task decomposition for `getPower_iterative`, avoiding any synchronization inside the loop and the appearance of false sharing.

2. Write two different OpenMP versions that implement a recursive (divide and conquer) task decomposition strategy for the recursive function (`getPower_recursive`). The two implementations should ideally exploit the parallelism in such a way that $T\infty \rightarrow \log(p)$ (if `MIN_POWER` was 1 and there wasn't any task creation parallel recursive control). The difference between the two versions is how they control task creation overheads:

   - *v*ersion 1: no more tasks are created once a specific parallel recursion depth (`MAX_DEPTH`) is reached.
   - *v*ersion 2: a task is created if and only if the total number created tasks so far is less than `MAX_TOTAL_CREATED_TASKS`.

Problem 2 (4 points) Consider the following data types and function definitions:

```
typedef struct {
    int key;
    int status; // -1 invalid
} tElem;
typedef tElem tVector[MAXELEM];

typedef struct tNode {
    int key;
    struct tNode *next;
} node;
typedef node *tHashTable[MAXHASH];

int valid (tElem e)
    return (!(e.status<0));

int hash_function (int key, int size)
    return (key%size);

void vector2hashTable (tVector v,  tHashTable h)
{
    int index;
    for (int i=0; i<MAXELEM; i++) {
        if (valid (v[i])) {
            index = hash_function (i, MAXHASH);
            insert_elem (v[i], index, h);
        }
    }
}
```

Function `vector2hashTable` inserts the elements from vector `v` into the hash table `h`, invoking function `hash_function` for which we provide a specific implementation in the code above. Notice that the **hash function is applied to the index** of vector `v`. Only valid elements from vector `v` (i.e. those with the `status` field $>=0$) will be inserted into the hash table.

We ask:

1. Write a parallel OpenMP version of function `vector2hashTable` that implements a CYCLIC data decomposition strategy for the OpenMP input (i.e. consecutive elements are mapped to consecutive threads, in a round robin way). Important: a) you must NOT use the OpenMP `for` directive; and b) the number of available threads is `p`, being `MAXHASH` multiple of `p`. Is it necessary to use any kind of thread synchronization during the execution of the loop? Reason your answer.

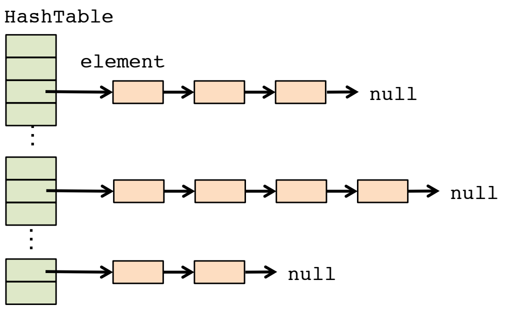
2. Write a parallel OpenMP version of function `vector2hashTable` that implements a BLOCK CYCLIC data decomposition strategy for the input (i.e. blocks of consecutive `MAXHASH/p` iterations are cyclically assigned to threads, in a round robin way). Important: a) you must NOT use the OpenMP `for` directive; and b) the number of available threads is `p`, assuming that `MAXELEM` is multiple of `MAXHASH` and `MAXHASH` is multiple of `p`. Is it necessary to use any kind of thread synchronization during the execution of the loop? Reason your answer.

3. Let's consider now that **the hash function is applied to the `key` field of the `tElem` structure** (i.e. `hash_function(v[i].key, MAXHASH)`). Is the OpenMP parallel code that you have written in the first question still correct?. If not, rewrite the code to make it work correctly.

Problem 3 (2 points) Assume the following definition for a hash table used to store the elements of a list:

```
#define SIZE_TABLE 1048576
typedef struct {
    int data;
    element * next;
} element;

element * HashTable[SIZE_TABLE];
```



And the following parallel version of a code to insert the MAX_ELEM elements in vector ToInsert:

```
#define MAX_ELEM 1024
int ToInsert[MAX_ELEM];
...
int main() {
    int i, index, num_elem;
    ...
    #pragma omp parallel private(index) num_threads(4)
    #pragma omp single
    for (i = 0; i < num_elem; i++) {
        index = hash_function(ToInsert[i], SIZE_TABLE);
        #pragma omp task firstprivate(i, index) depend(inout : HashTable[index])
        insert_elem (ToInsert[i], index);
    }
    ...
}
```

Function hash_function returns the entry of the table (between 0 and SIZE_TABLE-1) where each element in vector ToInsert has to be inserted. Function insert_elem inserts that element in the corresponding position inside the chained list pointed by HashTable[index].

Given the sequence index={5,10,10,10,14,25,8,25} returned by hash_function for a given vector ToInsert with num_elem=8 elements, **complete the timing diagram in the answer sheet** with the parallel execution with 4 threads, assuming that the execution of hash_function lasts 1 time unit, task creation takes 1 time unit, the execution of insert_elem lasts 10 time units and the rest of the operations (including selecting the next task to be executed) can be considered to use a negligible time. **Compute the total execution time** for the parallel region. Note: If more than one task is ready for execution, a thread will choose the one that was first created.

Student name: ....

Timeline diagram to be used to deliver your solution to Problem 3.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Thread 0																																									
Thread 1																																									
Thread 2																																									
Thread 3																																									

Clearly indicate what each thread is executing (Hx: hash_function, Tx: task creation, Ix: insert_elem), being x the value of index.