PAR – Final Exam – Course 2021/22-Q1

January 17^{th}, 2022

Problem 1 (2.5 points)

The following code computes matrix `u[N][N]` by blocks of $BS \times N$ elements, with `N` very large:

```
double u[N][N];

// Compute elements in a block of BS x N elements
void compute_row_block(int ii) {
    for (int i=max(1, ii); i<min(ii+BS, N-1); i++)
        for (int j=1; j<N-1; j++) {
            double tmp = u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j];
            u[i][j] = tmp/4;
        }
}

void main() {
    int NB = 2*P;          // Total number of row blocks
    int BS = N/NB;         // Number of rows per block

    // EVEN loop: traversing all EVEN blocks
    for (int ii=0; ii<NB; ii+=2)
        compute_row_block(ii*BS);

    // ODD loop: traversing all ODD blocks
    for (int ii=1; ii<NB; ii+=2)
        compute_row_block(ii*BS);
}
```

In this code the computation is divided in two parts: the so called *EVEN* loop computing half of the blocks first (blocks 0, 2, ...), and the so called *ODD* loop computing the other half of the blocks later (blocks 1, 3, ...). Before answering the first question below, think about the parallel execution opportunities in this code when defining each iteration of the *EVEN* and *ODD* loops as a task. Could all the tasks in the *EVEN* loop be executed in parallel? Could all the tasks in the *ODD* loop be executed in parallel? And could the execution of tasks in the *ODD* loop be performed in parallel with the execution of tasks in the *EVEN* loop?. Having all that in mind, **we first ask you to:**

1. Write the expression for the contribution to the total parallel time T_P coming from the parallel computation time T_P^{comp} on an ideal machine with P processors, assuming that: 1) $NB = 2 * P$ perfectly divides the problem size `N`; 2) the execution time for a single iteration of the innermost loop body in routine `compute_row_block` takes t_c time units; and 3) no parallelisation overheads should be considered at this point.

Next we consider that the ideal machine has the memory distributed among the P processors, In that machine, matrix u is divided row-wise in $NB = 2 * P$ blocks, where each block contains $BS = N/(2 * P)$ consecutive rows. Pairs of consecutive blocks are assigned to the same processor, so for example blocks 0 and 1 are owned by processor 0, blocks 2 and 3 are owned by processor 1; and so on and so forth. Each processor is in charge of computing all the rows in the blocks that are assigned to it, and therefore it will have to execute one iteration of the *EVEN* loop and one iteration of the *ODD* loop. Before answering the second question below, and having in mind the assignment of blocks and iterations, think about the need for processors to perform any remote access before starting the execution of tasks in the *EVEN* loop, or before starting the execution of tasks in the *ODD* loop; and, if affirmative, how many elements need to be transferred in each case and which processors to do them. Having all that in mind, next **we ask you to:**

2. Write the expression for the contribution to T_P coming from the data sharing overheads $T_P^{data\_sharing}$, if any, as part of the overall $T_P = T_P^{comp} + T_P^{data\_sharing}$. For that you should consider the data sharing model explained in class. Accesses to local memory are performed with zero overhead; accesses to remote memory take $t_{comm} = t_s + m \times t_w$, being t_s and t_w the start–up time for the remote access and transfer time of one element, respectively. At a given moment, a processor can only perform one remote memory access to another processor, and can only serve a remote memory access from another.

Problem 2 (2.5 points)

Given the following code fragment:

```
#define CACHE_LINE_SIZE  128
#define DBSize           (32*1024*1024)
#define NTHREADS         3

int count[NTHREADS];
int DB[DBSize];
int key;

// Initialization loop
for (int i = 0; i < NTHREADS; i++)
    count[i] = 0;
```

```
#pragma omp parallel num_threads(NTHREADS)
{
    int my_id = omp_get_thread_num();
    for (int i = my_id; i < DBSize; i += NTHREADS) {
        // read access to DB[i]
        if (DB[i]==key)
            // read access to count[my_id] followed by a write access
            count[my_id] = count[my_id] + 1;
    }
}
```

Assume a shared–memory UMA parallel system with 3 processors, each one with its own (private) cache memory. Data coherence in the system is maintained using Write Invalidate `MSI` protocol, with a Snoopy attached to each cache memory. Also assume 1) empty caches at the beginning of the program; 2) `count[0]` and `DB[0]` are aligned to the beginning of different memory lines; 3) the rest of variables are stored in registers; and 4) the size for a variable of type `int` is 4 bytes and cache line size is 128 bytes. **We ask you:**

1. Assume thread 0, running on processor 0, starts executing the sequential part of the program until the parallel region starts (i.e. it executes the initialization loop). How many cache lines are used to store the full `count` vector after the execution of the initialization loop? What is the cache coherence state for each cache line storing elements of `count` and in which private cache it is located?

2. Assume the cache states in previous question (after the execution of the initialization loop) and that each thread i runs on processor P_i of the UMA system within the parallel region. Complete the table in the provided answer sheet which represents the first sequence of actions (from top to bottom of the table) executed by the three threads in the parallel region for the first iterations of the loop. State MC_i in each row has to be filled with the state of the cache line after the memory access in the action. For the rest of columns, you should specify the processor event ($PrRd_i$, $PrWr_i$), if there is cache hit or miss, the bus command ($BusRd_i$, $BusRdX_i$, $BusUpgr_i$), and the flush transaction ($Flush_i$), where i is the number of the processor where it is generated. Note: Observe that if you want to leave a cell empty, you can write "-".

last page

3. Assuming the amount of data we are accessing in the program, a system with a main memory of 32 GBytes and 32 Kbytes of private cache memory per processor, what is the total number of bits that an UMA system would use to keep the cache coherence per processor and the overall system? Would this number of bits change if we change the protocol from MSI to MESI?

Problem 3 (2.5 points)

Given the following sequential program that computes the number of times the value stored in variable element appears in a vector of lists data:

```
#define NUM_ELEMS 10000
typedef struct list {
    int elem;
    struct list * next;
} list; // the basic component of a list

list * data[NUM_ELEMS]; // vector of lists, each list with varying number of elements
int element, count = 0; // value to search within data and number of times it appears

// function that returns the number of times element appears in theList
int list_search(list * theList, int element);

void main() {
    ...
    for (int entry = 0; entry < NUM_ELEMS; entry++) {
         int tmp = list_search(data[entry], element);
         count += tmp;
    }
    ...
}
```

Each of the lists may have a different number of elements, so when parallelising the program one should take care of load balancing. A parallel version for the sequential program above is also available, in which the original for loop has been substituted by a parallel region that assigns individual iterations to explicit tasks in such a way that tasks are generated under certain circumstances. One new shared variable active_tasks and two functions to operate it have also been added:

```
...
int active_tasks = 0;
// Functions to atomically add or subtract 1 to memory location pointed by address.
// The operation is saturated to the max or min value (i.e. the result can not be
// greater than max and smaller than min, respectively). They return value in memory
// location before operation
int atomic_inc(int *address, int max);
int atomic_dec(int *address, int min);

void main() {
    #pragma omp parallel
    #pragma omp single
    {
        int workers = omp_get_num_threads() - 1; // one thread focuses on
                                                  // task creation, the rest execute tasks
        for (int entry = 0; entry < NUM_ELEMS; entry++) {
            while (atomic_inc(&active_tasks, workers) == workers);
            #pragma omp task depend(inout: count)
                {
                int tmp = list_search(data[entry], element);
                count += tmp;
                atomic_dec(&active_tasks, 0);
                }
        }
    }
}
```

After compiling the parallel program and executing it with P processors (with $P > 1$) we detect that the program **is not achieving any speed–up**, although it produces a correct result.

1. Assuming the functionality for functions `atomic_inc` and `atomic_dec` explained in the code itself, what are these two functions used for in the program? Which is the number of implicit and explicit tasks that are generated during the execution of the program?

2. In the parallel version provided above, how many of these explicit tasks can be simultaneously executing? Rewrite the program above making the minimum appropriate changes in order to increase this number and, as a consequence, achieve a much better speed–up. Make sure the program generates the correct result. After these minimal changes, which can be the maximum number of explicit tasks that could be simultaneously executing?

3. Do an implementation for function `atomic_inc` making use of load–linked (`ll`) and store–conditional (`sc`) operations, defined as follows:

```
int ll(int *address); // returns the value stored in address
int sc(int *address, int value); // stores value in address if atomicity with ll
                                 // has been accomplished, returning true (1);
                                 // retuns false (0) otherwise
```

4. Finally we found a recursive version alternative to the original sequential code:

```
...
void rec_list_search(list ** data, int size, int element) {
    int tmp = list_search(data[0], element);
    count += tmp;
    if (size > 1)
        rec_list_search(data+1, size-1, element);
}

void main() {
    rec_list_search(&data[0], NUM_ELEMS, element);
}
```

   Write a parallel version for it that implements a *recursive* **leaf** *task decomposition.* This new version should not implement any cut-off mechanism to restrict the number of tasks that are generated.

Problem 4 (2.5 points)

Given the following code fragment computing matrix `m[N][N]`, with `N` much larger than the number of processors P to be used in the parallel execution, and with `N` not necessarily a multiple of P:

```
telem m[N][N];

for (int i=1; i<N-1; i++)
   for (int j=0; j<N; j++)
       m[i][j] = compute (m[i][j], m[i-1][j], m[i+1][j]);
```

We ask you to:

1. Decide the most appropriate *geometric data decomposition strategy* for matrix `m` and write a parallel version of the code above using OpenMP that corresponds to it. Your solution should a) minimize the synchronization overhead among implicit tasks and b) guarantee that the load unbalance is limited to `N` elements (i.e. the number of elements in a row or column of the matrix).

2. Now consider that the program is going to be executed on a parallel machine in which memory lines are 128 bytes long. The allocation in memory for matrix `m` is aligned to the start of a memory line and `sizeof(telem)` is 8 bytes (`N` is not necessarily multiple of `sizeof(telem)`). Decide the most appropriate *geometric data decomposition strategy* in this case and re-write the previous OpenMP parallel code and, if necessary, the definition of matrix `m`. Your solution should a) maximize parallelism among implicit tasks; and b) maximize data locality and reduce coherence traffic.

Student name: ....

Answer sheet for **Question 2.2**.

Parallel Region Execution							
Action	CPU event	Cache Miss/Hit	Bus command	Flush?	State MC_0	State MC_1	State MC_2
P_0 reads DB[0]							
P_0 reads count[0]							
P_1 reads DB[1]							
P_1 reads count[1]							
P_0 writes count[0]							
P_1 writes count[1]							
P_2 reads DB[2]							
P_2 reads count[2]							
P_2 writes count[2]							
P_0 reads DB[3]							
P_1 reads DB[4]							