PAR – Final Exam Theory/Problems– Course 2022/23-Q2

June 21^{th}, 2023

Problem 1 (2.5 points)

Given the following code including *Tareador* task annotations:

```
#define N 3
int I[N][N];
int R[N][N];
...
for (int i=0; i<N; i++) {
    tareador_start_task ("init1");
    I[i][0] = foo(i);   // cost = 1 t.u.
    tareador_end_task ("init1");

    for (int j=1; j<N; j++) {
       tareador_start_task ("init2");
       I[i][j] = j*I[i][0]; // cost = 2 t.u.
       tareador_end_task ("init2");
} }

for (int i=1; i<N-1; i++) {
    tareador_start_task ("comp");
    for (int j=i+1; j<N; j++)
       R[i][j] = i*(I[i-1][j]+I[i][j]+I[i+1][j]); // cost = 6 t.u
    tareador_end_task ("comp");
}
```

Assume: a) the execution time for tasks `init1` and `init2` is 1 and 2 t.u., respectively; b) the execution time for each iteration of the loop body for task `comp` is 6 t.u.; c) the rest of code has negligible cost; d) all scalar variables (i.e. all variables except matrices `I` and `R`) are stored in registers; e) function `foo` does not access any memory location during its execution; and f) `N` with the value defined in the code above. **We ask you** to:

1. Draw the Task Dependence Graph (*TDG*), indicating the cost of each task. Label each task with the values of i, and when necessary, with the pair of values i and j.

2. Based on the *TDG*, compute the values for T_1 and T_∞.

Next consider the data sharing model explained in class based on a distributed memory architecture in which we consider that local memory accesses do not introduce any overhead, but an access to data in different processors introduces a data–sharing overhead, determined by $t_{comm} = t_s + m \times t_w$; t_s is the "start-up" time, t_w is transfer time for one element and m the number of elements transferred during the remote access. Also, according to the data sharing model: at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Assuming a) the number of processors p is equal to N; b) matrices `I` and `R` are already distributed in the memory of each processor when the computation starts, following a *block row data decomposition*; c) the resulting matrices should remain distributed in the same way at the end of the execution; and d) tasks are assigned to processors following the *owner-computes rule* so that a task is executed by the processor that owns the memory that holds the output of that task. **We ask you:**

3. Draw the timeline for the execution with $p = 3$, showing both the computational and remote memory access costs.

4. Obtain the expression that determines the parallel execution time on $p = 3$ processors (T_p), as a function of t_s and t_w, assuming the task durations obtained before.

Problem 2 (2.5 points)
Given the following recursive sequential code:

```
#define N_MAX (1<<29) // 512*1024*1024
struct t_person{
   t_data data; // personal information of bank client
   float balance; // current balance for client
   float interest; // interest for client
};
struct t_person best_client;

void find_best_client(struct t_person *people, int n) {
   int n2 = n/2;
   if (n==1) {
      if (best_client.balance < people[0].balance)
          best_client = people[0]; // copy all info of the person to best_client
   } else {
       find_best_client(people, n2);
       find_best_client(people+n2, n-n2);
} }

int main() {
   struct t_person bank_info[N_MAX];
   ...
   best_client.balance=0.0;
   find_best_client(bank_info, N_MAX);
   ...
}
```

Write an OpenMP version following a tree recursive task decomposition, taking into account the task creation overheads and minimizing both task synchronizations and synchronizations to update shared variables.

Problem 3 (2.5 points)
Given the following sequential C code:

```
#define N 10000000
#define NUMBINS 100
int input[N];
int histogram[NUMBINS];

int findMax(int *v); // returns the maximum value encountered in vector v
int findMin(int *v); // returns the minimum value encountered in vector v
...
int min = findMin(input);
int max = findMax(input);
int binInterval = (max-min)/NUMBINS;

for (int i=0; i<N; i++) {
    int bin = (input[i]-min)/binInterval;
    histogram[bin]++;
}
```

We ask you to:

1. Implement a parallel *OpenMP* version of the code that follows an ***output block geometric data decomposition***, where synchronization overheads are minimized and parallelism is maximized.

2. Assume that the processor's cache line size is 64 bytes, also that the size of the `int` data type is 4 bytes and the initial addresses of `input` and `histogram` vectors are both aligned with the start of a cache line. Write a new parallel *OpenMP* version of the code (without changing the definition of the used data structures) to avoid *false sharing* overheads.

Problem 4 (2.5 points) Assume a multiprocessor system composed of two NUMA nodes, each with 16 GB of main memory. Each NUMA node has two cores (NUMAnode0: core0-1, NUMAnode1: core2-3), each core with a private cache of 4 MB. Cache and memory lines are 32 bytes wide and data coherence in the system is maintained using Write-Invalidate protocols, with a Snoopy attached to each cache memory to provide MSI coherency within each NUMA node and a MSU directory-based coherence among the two NUMA nodes.

Given the following parallel *OpenMP* code excerpt to be executed on the described system:

```
1   #define IMAGESIZE 128*128
2   #define MAXITERS 100
3   typedef float tpixel[3]; // r,g,b values
4   tpixel image[IMAGESIZE], result[IMAGESIZE];
    ...
5   initialization (result); // on core 0
6   #pragma omp parallel num_threads(3)
7   {
8     int id = omp_get_thread_num();
9     for (int iter=0; iter<MAXITERS; iter++)
10        for (int i=0; i<IMAGESIZE; i++)
11           result[i][id] = apply_func (image[i][id],   // apply_func does not affect
12                                       result[i][id]); // any other variables
13  }
```

and assuming that: 1) vectors `image` and `result` are the only variables that will be stored in memory (the rest of variables will be all in registers of the processors); 2) the initial addresses of `image` and `result` vectors are aligned with the start of a cache line; 3) the size of a `float` data type is 4 bytes; and 4) $thread_i$ always executes on $core_i$, where i = [0,1,2], **we ask you to:**

1. Indicate how many entries in the directory structure are needed to store the coherence information of the `result` vector. In addition, for each entry indicate the number of bits needed and their function.

2. Compute the total number of bits that are required to maintain the coherence information at cache level within each NUMA node. Indicate also the number of bits per cache line and their function.

3. Complete the table provided with information for each enumerated memory operation (after it is completed), from the previous code, executed by the threads in the parallel region. You should specify the relative number of the memory line affected (with respect to the start address of the *result* vector, i.e. *result[0][0]* affects line 0), hit or miss, bus transactions of the MSI Snoopy protocol, state of the cache line in the MSI Snoopy protocol, whether there are or not commands from the Directory protocol between NUMA nodes (yes or no), state of the memory line in the Directory and, Presence bits.
4. The speedup achieved when executing the previous parallel code, with three threads, is far below 3x. Explain briefly the reason and suggest any modification to the data structures in order to improve its performance. Re-write the necessary code.

Action	Memory line	Cache Miss/Hit	Bus command	State MC_0	State MC_1	State MC_2	NUMA comands (y/n)	Directory State	Presence bits
Initial state result[0][0:2]				M	-	-		M	01
$thread_1$ reads result[0][1]									
$thread_2$ reads result[0][2]									
$thread_1$ writes result[0][1]									
$thread_0$ reads result[0][0]									