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

**June $21^{th}$, 2023**

**Problem 1** (2.5 points)

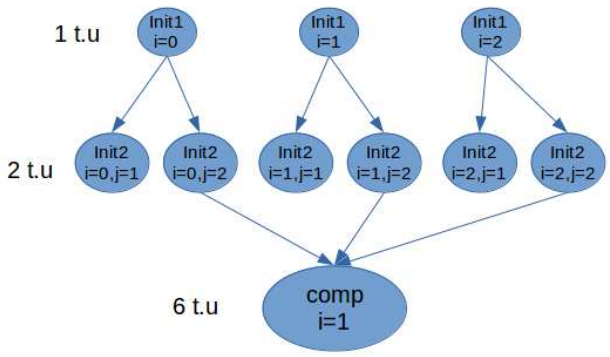
Given the following code including *Tareador* task annotations:

```
#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 $j$.

   **Solution:**




2. Based on the *TDG*, compute the values for $T_1$ and $T_\infty$.

   **Solution:**

   $T_1 = 1 \times 3 + 2 \times 6 + 6 \times 1 = 21$ time units.

   $T_\infty = 1 + 2 + 6 = 9$ time units , determined by the critical path: *init1, init2, comp.*

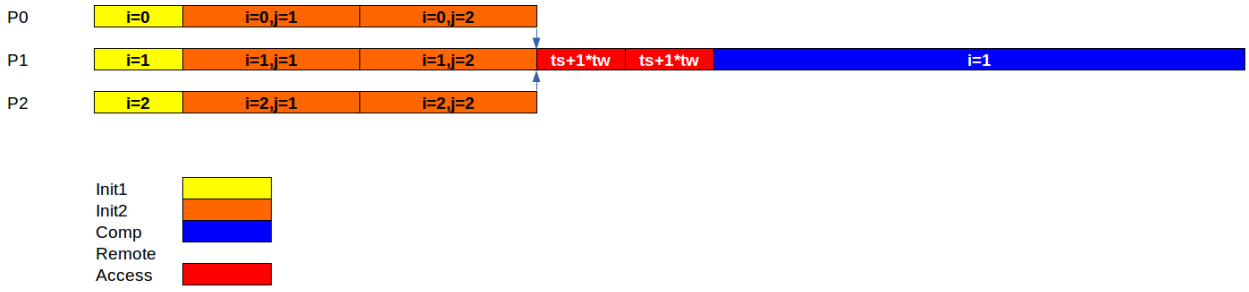
Next consider the data sharing model explained in class based on a distributed memory architecture in which we consider that local memory accesses do not introduce any overhead, but an access to data in different processors introduces a data–sharing overhead, determined by $t_{comm} = t_s + m \times t_w$; $t_s$ 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.

   **Solution:**

   Block row data decomposition of a matrix with a number of rows $N$ equal to $P$ means that each processor $i$ will have row $i$ of the matrix. Therefore, tasks `init1` and `init2` accessing row $i$ of matrix $I$ are assigned to processor $i$ because their own row $i$ of that matrix. Task `comp` access row 1 of matrix $R$. Then, it is assigned to processor 1. Following owner-compute rules, the mapping and execution of the tasks follows:




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.

   **Solution:**

   $T_p = 1 + 2 \times 2 + 2 \times (t_s + 1 \times t_w) + 6 = 11 + 2 \times (t_s + 1 \times t_w)$ time units

**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.

**A Solution:**

```
... // same declarations

#define CUTOFF (4)

void find_best_client(struct t_person *people, int n, int depth) {
   int n2 = n/2;
   if (n==1) {
      if (best_client.balance < people[0].balance) // FIRST TEST (reduce # criticals)
      #pragma omp critical
      {
        if (best_client.balance < people[0].balance) // SECOND TEST (necessary!)
          best_client = people[0]; // copy all info of the person to best_client
      }
   } else {
      if (omp_in_final())
      {
       find_best_client(people, n2, depth);
       find_best_client(people+n2, n-n2, depth);
      } else {
       #pragma omp task final(depth>=CUTOFF)
       find_best_client(people, n2, depth+1);
       #pragma omp task final(depth>=CUTOFF)
       find_best_client(people+n2, n-n2, depth+1);
       // NO TASKWAIT
      }
} }
```

```
int main() {
   struct t_person bank_info[N_MAX];
   ...
   best_client.balance=0.0;
   #pragma omp parallel
   #pragma omp single
   find_best_client(bank_info, N_MAX, 0);
   ...
}
```

**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.

   **Solution:**

```
#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;

#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = NUMBINS / nths;
 int mod = NUMBINS % nths;
 int start = id*BS;
 int end = start + BS;
 if (mod > 0) {
   if (id < mod) {
      start += id;
```

```
      end = start + BS + 1;
    }
    else {
      start += mod;
      end += mod;
    }
  }
  for (int i=0; i<N; i++) {
     int bin = (input[i]-min)/binInterval;
     if (bin >= start && bin < end)
         histogram[bin]++;
  }
 }
```

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.

   **Solution:**

   The cache line size is 64 bytes, so the total number of elements from `histogram` that fit in one cache line is 16. In order to avoid false sharing we propose a block-cyclic data decomposition in such a way that each block has a number of elements that fit in one cache line, so each thread should be allocated a block of size 16.

```
#define N 10000000
#define NUMBINS 100
#define CACHE_LINE_SIZE 64
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;

#pragma omp parallel
{
 int id = omp_get_thread_num();
 int nths = omp_get_num_threads();
 int BS = CACHE_LINE_SIZE / sizeof(int);

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

**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.

   **Solution:**

   Each element of the `result` vector has 12 bytes of total size ($3 \times 4bytes = 12bytes$), the total size of the vector would be: $12 \times (128 \times 128) = 12 \times 2^{14}$

   The number of memory lines needed to allocate the `result` vector is : $(12 \times 2^{14})/2^6 = 12 \times 2^8$

   For each memory line it is necessary to keep the status information (*M,S,U*), which can be stored in 2 bits and the presence bits (one bit per NUMA node, total = 2 bits).

   The total number of bits needed is : $12 \times 2^8 \times (2 + 2)$

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.

   **Solution:**

   Each cache has a total size of 4 MB and the cache line size is 64 bytes, so the total number of lines per cache = $2^{22}/2^6 = 2^{16}$ cache lines per cache.

   For each cache line it is necessary to keep the status information (*M,S,I*), which can be stored in 2 bits.

   So finally, the total number of bits necessary to keep the information of the Snoopy MSI coherence protocol is: $2^{16}2 = 2^{17}$

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] | | | | | | | | | |

| Solution | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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] | 0 | Miss | $BusRd_1$/ $Flush_0$ | S | S | . | n | S | 01 |
| $thread_2$ reads result[0][2] | 0 | Miss | $BusRd_2$ | S | S | S | y | S | 11 |
| $thread_1$ writes result[0][1] | 0 | Hit | $BusUpgr_1$ | I | M | I | y | M | 01 |
| $thread_0$ reads result[0][0] | 0 | Miss | $BusRd_0$/ $Flush_1$ | S | S | I | n | S | 01 |

**Solution:**

Each element of the `result` vector has 3 fields of `float` data type, computing a total size of 12 bytes per element. The size of a cache line is 64 bytes. Given that each element of the result vector is being updated by the three threads, a false sharing situation occurs with high probability (every time more than one thread is updating the same entry at the result vector).

In order to avoid such inefficiency we can apply "padding", that is add some extra bytes to prevent different threads from accessing the same cache line for updating it concurrently. One possible solution could be to add extra bytes following each field of the `tpixel` structure in order to complete a cache line (number of extra bytes = 64 - 4 = 60 bytes, which makes 60/4 = 15 extra elements) In this case we need to substitute line 3 of the code by:

```
1   #define IMAGESIZE 128*128
2   #define MAXITERS 100
3   typedef float tpixel[3][1+15]; // r ,g ,b values with padding
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][0] = apply_func (image[i][id][0],   // apply_func does not affect
```

```
12                                      result[i][id][0]); // any other variables
13   }
```

# PAR – Final Exam Laboratory– Course 2022/23-Q2

**June $21^{th}$, 2023**

**Problem 1: Lab 1** (2.5 points)

Given the following outputs obtained after the interactive execution of `pi_omp` with different number of threads:

```
par@boada-7> OMP_NUM_THREADS=1 /usr/bin/time ./pi_omp 1000000000
Number pi after 1000000000 iterations with 1 threads = 3.141592653589828
Execution time (secs.): 2.368062
2.36user 0.01system 0:02.37elapsed 99%CPU (0avgtext+0avgdata 4320maxresident)k
0inputs+0outputs (1major+275minor)pagefaults 0swaps

par@boada-7> OMP_NUM_THREADS=2 /usr/bin/time ./pi_omp 1000000000
Number pi after 1000000000 iterations with 2 threads = 3.141592653589845
Execution time (secs.): 1.186354
2.36user 0.00system 0:01.19elapsed 198%CPU (0avgtext+0avgdata 4668maxresident)k
0inputs+0outputs (1major+381minor)pagefaults 0swaps

par@boada-7> OMP_NUM_THREADS=4 /usr/bin/time ./pi_omp 1000000000
Number pi after 1000000000 iterations with 4 threads = 3.141592653589845
Execution time (secs.): 1.188191
2.36user 0.01system 0:01.19elapsed 198%CPU (0avgtext+0avgdata 4672maxresident)k
0inputs+8outputs (1major+306minor)pagefaults 0swaps
```

1. Explain the meaning the four first values reported by the `/usr/bin/time` command when executed with a single thread: *2.36user 0.01system 0:02.37elapsed 99%CPU*.

   **Solution:**

   They are the user CPU time, system CPU time, elapsed time and % of elapsed time dedicated to CPU time (user+system CPU time*100/elapsed time), respectively.

2. When executed with 2 and 4 threads, explain why *2.36user* does not change with respect to the value reported for a single thread, and why the *198%CPU* is the same for 2 and 4 threads.

   **Solution:**

   The value *2.36user* corresponds to the addition of the CPU time of each of the 2 or 4 threads during the execution. Each thread lasts of $2.36/nt$ approx, being $nt$ the number of threads.

   The %CPU is close to 200% for 2 threads because the CPU time remains the same while the elapsed time is half the elapsed time of a single thread execution. In the case of 4 threads is still the same because we only have two cores in interactive mode, and the elapsed time remains the same than the case with 2 threads.

**Problem 2: Lab 3** (2.5 points)

Assuming the following task decomposition strategies to parallelize the *Mandelbrot set*, which one do you think would be more efficient? Please reason your answer taking into account the number of tasks generated/executed and their granularity for each strategy.

```
#pragma omp parallel    // Code 1
#pragma omp single
 for (int row = 0; row < height; ++row)
   for (int col = 0; col < width; ++col) {
     #pragma omp task firstprivate(row,col)
     {
             . . .
   } }
```

```
#pragma omp parallel    // Code 2
#pragma omp single
 #pragma omp taskloop
 for (int row = 0; row < height; ++row) {
   for (int col = 0; col < width; ++col) {
             . . .
 } }
```

**Solution:**

Code 2 is much more efficient than Code 1. Code 1 corresponds with the first point strategy implementation that you evaluated in the laboratory session. One of the threads in Code 1 has to create a huge number of very fine-grain tasks sequentially. This incurs a significant overhead of task creation. Code 2 corresponds with the row strategy implementation with taskloop that you also evaluated in the laboratory session. This reduces the number of tasks created, which usually is proportional to the number of threads in the parallel region and much less than the number of tasks created in Code 1. The task creation overhead is significantly reduced with larger task granularities.

**Problem 3: Lab 4** (2.5 points)

Consider the following sequential *multisort* code.

```
 1   void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
 2    if (length < MIN_MERGE_SIZE*2L) { // Base case
 3       basicmerge(n, left, right, result, start, length);
 4    } else { // Recursive decomposition
 5       merge(n, left, right, result, start, length/2);
 6       merge(n, left, right, result, start + length/2, length/2);
 7   }}
 8
 9  void multisort(long n, T data[n], T tmp[n]) {
10     if (n >= MIN_SORT_SIZE*4L) {     // Recursive decomposition
11        multisort(n/4L, &data[0], &tmp[0]);
12        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
13        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
14        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
15
16        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
17        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
18
19        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
20    } else     // Base case
21        basicsort(n, data);
22  }
23  void main() {
24   ...
25   multisort(n,elements,elements_tmp);
36   ...
37  }
```

Let's suppose that you parallelize it following a tree strategy decomposition with a given cut-off level. Assuming the laboratory problem size, 1024K elements to sort, and MIN_SORT_SIZE and MIN_MERGE_SIZE equal to 256. Answer the following questions:

1. Assume you want to use the `final(depth>=cut_off)` clause in the task pragmas, being "depth" the recursion level. Indicate where (i.e, in the sequential code line number above) you will add the task creation control to continue creating tasks or not, and write the code line to do it.

   **Solution:**

   Substitute the code at lines 5-6 by the following code structure:

   ```
   11  if (!omp_in_final())
       {
         Code with tasks
       } else {
         Code without tasks
       }
   ```

Also, substitute the code at lines 11-19 with the same code structure.

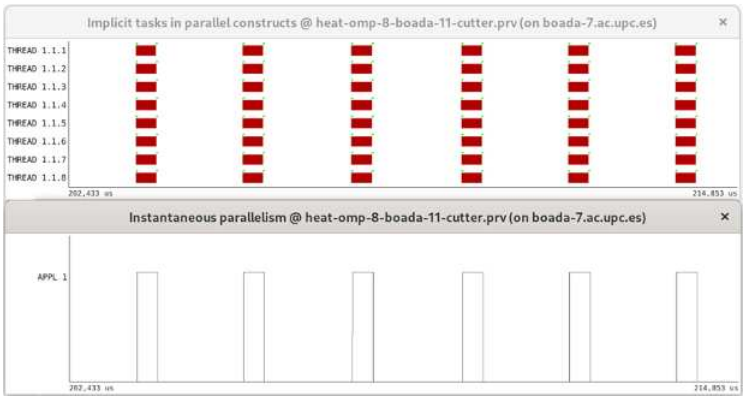
2. Assume now you are using task with depend clauses (with no cut-off control). Indicate the synchronisation (`depends` clauses, `taskwaits` or/and `taskgroups`) that you need to add in your tree-recursive task decomposition (i.e the sequential code line number, before and/or after you will insert them) when using tasks with depend clauses to guarantee the functionality of the *multisort* program.

   **Solution:**

   No solution provided. Look at your laboratory deliverable and feedback.

**Problem 4: Lab 5** (2.5 points)

Assume that you have parallelized *Jacobi* solver function with the data decomposition seen at the laboratory. You run *modelfactors* and use the instantaneous parallelism and implicit tasks in parallel constructs hints with *Paraver* to figure out why *Jacobi* application is not scaling properly. The "zoom in capture" of part of the execution trace for 8 threads of the first implementation version of heat application with *Jacobi* solver follows (top: implicit tasks in parallel constructions, bottom: instantaneous parallelism):




We ask you to briefly explain your answer to the following questions:

1. What do you observe with instantaneous parallelism hint at the view of *Paraver*? What is the instantaneous parallelism achieved during the *Jacobi* solver function and during the execution of the other parts of the code?

   **Solution:** The amount of useful work done (state running) by all the threads at a certain instant of time. That is, if two threads are running, instantaneous parallelism is 2. The instantaneous parallelism achieved by *Jacobi* is about *nt*, being *nt* the number of threads. However, it is 1 for other parts of the code because they are not parallelized.

2. How did you solve the lack of parallelism and performance problem of the initial *Jacobi* implementation to achieve linear scalability?

   **Solution:**

   No solution provided. Look at your laboratory deliverable and feedback.