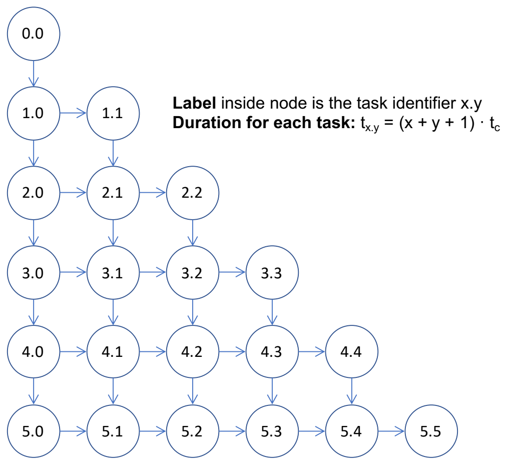
PAR – 1^{st} In-Term Exam – Course 2018/19-Q1

November 7^{th}, 2018

Problem 1 (4 points) Given the following TDG:


0.0
1.0
1.1
2.0
2.1
2.2
3.0
3.1
3.2
3.3
4.0
4.1
4.2
4.3
4.4
5.0
5.1
5.2
5.3
5.4
5.5
Label inside node is the task identifier x.y
Duration for each task: $t_{x,y} = (x + y + 1) \cdot t_c$

for the task decomposition associated to the following sequential program:

```
#define N 6
void do_computation () {
    for (int x=0; x<N; x++)
        for (int y=0; y<=x; y++)
            compute_task (x, y);
}

void main () {
     do_computation();
}
```

in which the duration in time units of each `compute_task` depends on the value of the row (`x`) and column (`y`) it is placed (label `x.y`): $t_{x.y} = (x + y + 1) \times t_c$.

1. (1 point) Compute the values for T_1, T_∞ and $Parallelism$.

2. (1 point) Assuming: 1) a task is created as soon as all its dependences are satisfied; and 2) the overhead for the creation of each task is t_{create}. How the expression for T_∞ would change if you take into account task creation overheads?

3. (1.5 points) Assuming: 1) the same task creation mechanism but with $t_{create} = 0$; 2) there is no overhead due to task synchronisation; and 3) an ideal machine with P=3 processors and the three possible assignments of tasks to processors shown in the following table:

Assignment	p_0	p_1	p_2
block	$x = \{0, 1\}$ and $y = \{0 : x\}$	$x = \{2, 3\}$ and $y = \{0 : x\}$	$x = \{4, 5\}$ and $y = \{0 : x\}$
interleaved	$x = \{0, 3\}$ and $y = \{0 : x\}$	$x = \{1, 4\}$ and $y = \{0 : x\}$	$x = \{2, 5\}$ and $y = \{0 : x\}$
balanced	$x = \{0, 5\}$ and $y = \{0 : x\}$	$x = \{1, 4\}$ and $y = \{0 : x\}$	$x = \{2, 3\}$ and $y = \{0 : x\}$

We ask you to draw a timeline representing the execution of the tasks and compute the speed-up S_3 that could be achieved for each of the proposed task–to–processor assignments above. **Important:** tasks assigned to a processor should be executed in an order that minimizes the overall parallel execution time T_3.

4. (0.5 points) Assuming: 1) only horizontal arrows in the TDG (i.e. dependences between tasks with the same value of x) imply data sharing; and 2) the overhead for data sharing is constant $t_{sharing}$. How the previous expressions for T_3 change when you consider data sharing overheads?

Solution: Since tasks with the same value of x are mapped to the same processor, all accesses to data are local, so data sharing overheads do not have any effect on T_3.

Problem 2 (3 points) Given the following **incomplete recursive** parallel version for the same program specified in the previous problem:

```
#define N 6
int deps[N][N];

void compute_task (int x, int y) {
    compute (x, y);    // actual computation inside task
    if (x<(N-1)) {
        deps[x+1][y]--;
        }
    if (y<x) {
        deps[x][y+1]--;
        }
}

void main () {
    #pragma omp task
    compute_task(0, 0);
}
```

in which each task should be created as soon as all its dependences are honoured; dependences are controlled with matrix `deps` which is initialized as follows:

deps[6][6] — y →

x ↓	0	1	2	3	4	5
0	0					
1	1	1				
2	1	2	1			
3	1	2	2	1		
4	1	2	2	2	1	
5	1	2	2	2	2	1

Observe that `deps[x,y]` is initialized with the number of predecessor tasks each task `x.y` depends on, according to the TDG.

1. (1.5 points) We ask you to complete the program, **including the necessary C code and OpenMP pragmas** so that it follows the task creation strategy described above, i.e. a task is recursively created as soon as all its dependences are satisfied. All necessary **data clauses** in the OpenMP `task` pragmas you add should be made explicit (i.e. you should specify them).

2. (1.5 points) Write an alternative version for the program that makes use of **task dependences**, as defined by OpenMP: task dependences are defined between sibling tasks, i.e. task generated by the same parent task.

Problem 3 (3 points) In this problem we consider an alternative implementation for the TDG in *Problem 2* based on the use of a graph representation, defined as follows in C:

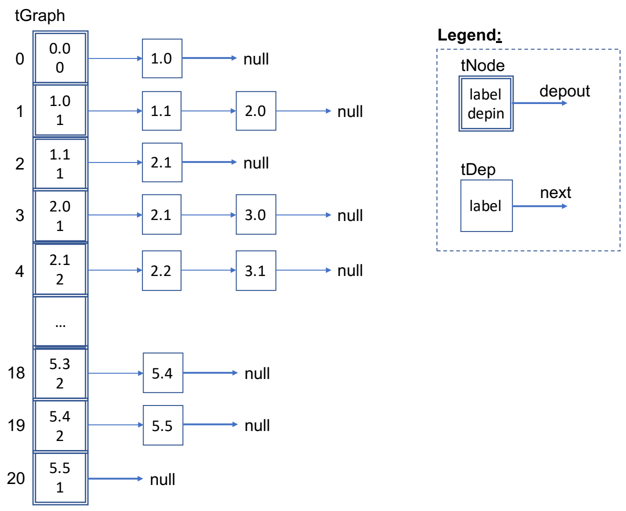
```
#define tLabel int       // not relevant for the problem

struct tDep {
       tLabel label;
       struct tDep *next;
};
```

```
struct tNode {
    tLabel label;
    int depin;
    struct tDep *depout;
};

typedef struct tNode tGraph[MAXNODES];
```

Each node of the TDG (i.e. a task instance) is stored as an element of a vector of type `tGraph`; the size of the graph is limited to `MAXNODES` nodes. For each node of type `tNode` the previous definition stores a `label` that identifies the task, the **number of predecessors** for the task (`depin` – the number of tasks this task depends on) and the pointer to the **list of successor tasks** (i.e. tasks whose execution depends on this one). The following figure shows how the TDG in *Problem 1* would be represented (assuming the label could be of type `tLabel`):



The TDG is first created using function:

```
void instantiate_task(tGraph g, tLabel label, tDep *predecessors);
```

The function receives as arguments the TDG, the label of the task being instantiated and the list of its predecessor tasks (i.e. tasks that have to be executed before it). The execution of this function will add a new task in the TDG and update the nodes for all predecessor tasks so that they are aware of their new successor task.

Once the whole TDG is created, then tasks are executed according to the TDG. When a task finishes it executes the following function:

```
void terminate_task(tGraph g, tLabel label, tDep **ready_tasks);
```

The function receives as arguments the TDG, the label of the task being terminated and returns a list (`ready_tasks`) with all successor tasks that due to the termination of the task become ready to be executed. Therefore, the execution of this function should decrement the `depin` counter for every node in its `depout` successors list, adding to `ready_tasks` those tasks whose `depin` counter reaches the value 0.

The following auxiliary functions are also available:

- `int map (tGraph g, tLabel label)`: given a task identified by `label` returns its index in the vector that stores the node in `g`. If the task is new then it returns an empty position to store it.
- `void insert_in_list(tLabel label, struct tDep **list)`: this function inserts `label` in the list pointed by `list` (argument by reference, so it returns the pointer to the new list).

- `tLabel remove_from_list(struct tDep **list)`: this function removes the first element from the list pointed by `list` (argument by reference, so it returns the pointer to the new list), returning the `label` of the element removed.

We ask you to complete the implementation of functions `instantiate_task` and `terminate_task` given below, adding the necessary OpenMP synchronisation constructs that you consider necessary, making sure that **multiple simultaneous** invocations of `instantiate_task` can happen during the TDG creation and **multiple simultaneous** invocations of `terminate_task` can happen during the TDG execution. Notes: 1) you should not care about thread and task creation; both happen outside these two functions (so you don't need to add them in your code); 2) you should only care about adding the appropriate synchronisation to ensure that the data structure is properly updated; 3) in case of using locks, you should declare them as part of structure `tNode` and you should not care about their initialisation and destruction; and 4) the `ready_tasks` is a list shared by all threads. Your implementation should maximize the degree of parallelism when updating the TDG.

```
void instantiate_task(tGraph g, tLabel label, struct tDep * predecessors) {
    struct tDep * t;
    int src, dest;
    // insert new node in the TDG
    dest = map(g, label);
    g[dest].label = label;
    g[dest].depin = 0;
    g[dest].depout = NULL;
    for (t = predecessors; t != NULL; t = t->next) {
        // update each predecessor node in the TDG
        src = map(g, t->label);
        insert_in_list(label, &g[src].depout);
        g[dest].depin++;
    }
}

void terminate_task(tGraph g, tLabel label, struct tDep **ready_tasks) {
    int src, dest;
    src = map(g, label);
    for ( ; g[src].depout != NULL; ) {
        // update each successor node in the TDG
        tLabel tmp = remove_from_list(&g[src].depout);
        dest = map(g, tmp);
        g[dest].depin--;
        // and if free of dependences then queue it in ready task list
        if (g[dest].depin == 0)
            insert_in_list(g[dest].label, ready_tasks);
    }
}
```

Solution: