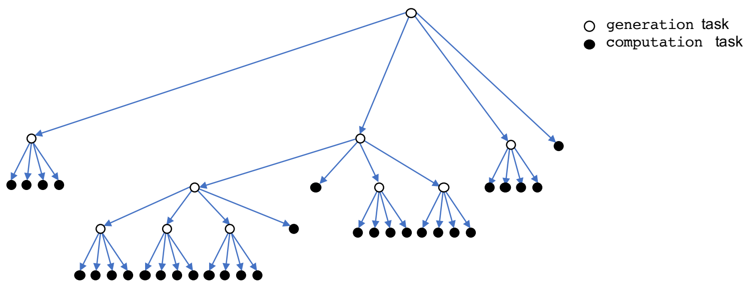
PAR – 1^{st} In-Term Exam – Course 2016/17-Q2

April 19^{th}, 2017

Problem 1 (1.5 points) Given the following task dependence graph for the **parallelizable part** of a program:



in which there are two different kind of tasks: 1) `generation` tasks in charge of generating more tasks by traversing a recursive program; and 2) `computation` tasks in charge of doing the actual computation.

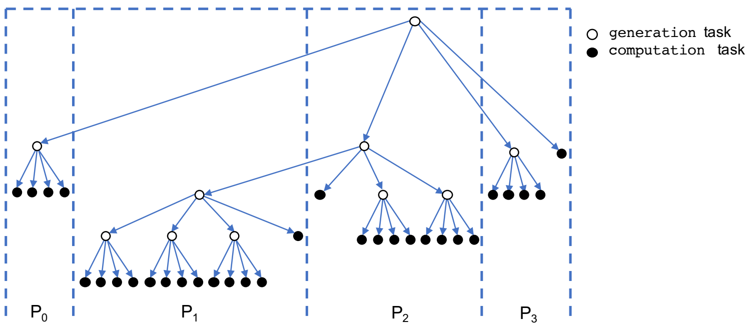
1. Assuming that the execution of each `generation` and `computation` task takes 3 and 12 time units, respectively, **we ask** to compute the values of T_1, T_∞ and *Parallelism* for the parallelizable part of the program. Consider that both kind of tasks are also executed in the sequential version of the program.

   Solution: In the task graph there are 10 `generation` and 31 `computation` tasks. So in total $T_1 = 10 \times 3 + 31 \times 12 = 402$. The critical path in the graph has 4 `generation` and 1 `computation` tasks, so $T_\infty = 4 \times 3 + 1 \times 12 = 24$, and therefore $Parallelism = 16.75$.

2. If the execution of the **non-parallelizable** part of the program takes 38 time units, **we ask** to compute the value of the parallel fraction ϕ for the whole program and the ideal speed–up S_∞ that could be achieved if we are able to scale the parallelizable part to infinite processors.

   Solution: Since the parallelizable part takes 402 time units, the parallel fraction can be computed as $\phi = 402 \div (38 + 402) = 0.91$. According to Amdalh's law the ideal speed–up would be $S_\infty = 1 \div (1 - \phi) = 11.58$.

Problem 2 (1.5 points) Assume that the tasks in the previous task graph are assigned to 4 processors as shown in the following picture:



For the same task durations in the previous problem and duration of the non–parallelizable part, **we ask** to compute the values for T_4 and S_4 **for the whole program**.

Solution: From the assignment of tasks to processors, it is clear that processor P_1 is the one that determines the execution time, who can start its execution after P_2 has executed 2 `generation` tasks. Therefore, $T_4 = 38 + (2 \times 3) + (4 \times 3 + 13 \times 12) = 212$ and $S_4 = T_1/T_4 = 440/212 = 2.07$.

Problem 3 (2.5 points) Assume that each `computation` task performs the computation specified in the following code:

```
void computation (double *A, double *B, int size) {
    for (int i=0; i<size; i++) A[i] += foo(B[i]);
}
```

Observe that during the execution of the task each element of vector `A` accumulates the value of the corresponding element in vector `B` after invoking `foo`.

In order to improve the parallelization efficiency someone has proposed us the following idea to dynamically balance the load (number of tasks) assigned to each processor: *"as soon a processor finishes with all the tasks initially assigned to it, it steals a `computation` task from the most loaded processor at that moment; the process is repeated until all tasks are executed"*. However, stealing the execution of a task incurs an overhead that is associated with 1) the cost of moving the data that is needed to execute that task from the memory of the owner processor to the new processor, and 2) returning the computed result to the memory of the owner processor once the computation is finished. Assuming that each processor has all data associated to the initially assigned tasks and the data sharing model explained in class ($t_{overhead} = t_s + m \times t_w$, being $t_s = 1$ and $t_w = 0.1$ the start-up time and transfer time for each data element, respectively), **we ask:**

1. Compute the value of the overhead associated with the **remote execution of one `computation` task**, assuming that `size=20` for all `computation` tasks and that all the elements of a vector are transferred in a single message, each vector in a different message.

   Solution: Since `size=20`, then the overhead of each individual remote access takes $t_{overhead} = 1 + 20 \times 0.1 = 3$ time units. For each remote task execution there are two reads and one write, so the total overhead is 9 time units.

2. Using the attached solution sheet, draw a possible temporal diagram for the execution of the previous task graph, considering the initial assignment of tasks to processors shown in **problem 2** and the use of the stealing mechanism that has been proposed to dynamically balance the load initially assigned. **Important:** 1) assume that tasks are extracted from the task pool first choosing the one that is at a lower recursion level (i.e. close to the root of the tree), and second, if more than one is at the same recursion level, choosing from left to right; 2) if several threads attempt to steal tasks from the same processor at the same time, the steals will be sequentialized; and 3) according to the data sharing model, a processor can only serve one remote transfer from other processors at a time.

   Solution: See attached solution sheet.

Problem 4 (3 points) Assume that the task graph shown in **problem 1** is generated from the parallel OpenMP version of the following sequential recursive program:

```
#define SIZE_VECTOR 992      // always larger than 4 times MIN_SIZE
#define MIN_SIZE 32

void generation(double * A, double * B, int size) {
    int assigned = 0;
    for (int i=0; i<4; i++) {
        int m = partition(size-assigned, i);
        if (m > MIN_SIZE) generation(&A[assigned], &B[assigned], m);
        else computation(&A[assigned], &B[assigned], m);
        assigned += m;
    }
}
void main() {
      generation(vectorA, vectorB, SIZE_VECTOR);
}
```

In this program, function `partition` randomly returns a value that is multiple of `MIN_SIZE` (i.e. between `MIN_SIZE` and the remaining number of elements `size-assigned`), ensuring that all the `size` elements are partitioned after the four invocations in the loop. **We ask** to write an OpenMP parallelization that could potentially generate the tasks shown in the task graph in **problem 1**. In addition, in order to minimize the overheads of task creation, the code will have to include a cut–off mechanism to ensure that no additional `generation` tasks are created after a certain recursion level (`MAX_LEVEL`) (**important:** the cut-off mechanism should only apply to `generation` tasks, not to `computation` tasks).

Solution: It is not possible to make use of the clause `final` since this would also cut-off the generation of `computation` tasks at the leaves. So the cut-off is implemented with a conditional statement.

```
#define SIZE_VECTOR = 992
#define MIN_SIZE = 32

void generation(double * A, double * B, int size, int depth) {
    int assigned = 0;
    for (int i=0; i<4; i++) {
        int m = partition(size, assigned, i);
        if (m > MIN_SIZE)
            if (depth < MAX_LEVEL)
                #pragma omp task
                generation(&A[assigned], &B[assigned], m, depth+1);
            else
                generation(&A[assigned], &B[assigned], m, depth+1);
        else
            #pragma omp task
            computation(&A[assigned], &B[assigned], m);
        assigned += m;
    }
}

void main() {
    #pragma omp parallel
    #pragma omp single
    // the following initial task could be avoided if we consider the implicit task
    // executing the single
    #pragma omp task
    generation(vectorA, vectorB, SIZE_VECTOR, 0);
}
```

Problem 5 (1.5 points) Function `int foo(int input)` invoked in **problem 3** performs a certain computation that takes a considerable amount of time. Since the `input` may appear multiple times during program execution, we have decided to do an implementation that memorizes previously computed values, assuming that `input` values are always in a range between 0 and `MAX-1`:

```
struct entrada {
    int computed = 0; // 0 to indicate that the result is not already computed
    int result;
} table[MAX]

int foo(int input) {
if (table[input].computed == 0) {
    table[input].result = expensive_comp(input);
    table[input].computed = 1;
    }
return(table[input].result);
}
```

We ask you to complete the implementation of function `foo`, the main program in **problem 4** and the definition of data type `entrada` to guarantee the correct access to `table` while maximizing the parallelism.

Solution: In the solution below one thread waits if another thread is computing the result for a particular `input` vaule at that moment; as soon as the computing thread finishes, the one waiting will get the already computed value. This is better than re-computing the value, which may take longer than the waiting time for the result.

```
struct entrada {
    omp_lock_t locke;
    int computed = 0; // 0 to indicate that the result s not already computed
    int result;
} table[MAX]
```

```
int foo(int input) {
omp_set_lock(table[input].locke);
if (table[input].computed == 0) {
    table[input].result = expensive_comp(input);
    table[input].computed = 1;
    }
omp_unset_lock(table[input].locke);
return(table[input].result);
}
...
void main() {
      for (int i=0; i< MAX; i++)  omp_init_lock(&table[i].locke);
      #pragma omp parallel
      #pragma omp single
      generation(vectorA, vectorB, SIZE_VECTOR);
      for (int i=0; i< MAX; i++)  omp_destroy_lock(&table[i].locke);
}
```

In order to be the execution correct, one should include a `#pragma omp flush` in order to enforce memory consistency among threads. However this is an issue not explained in class, so it has not been considered when evaluating this problem.

Student name: ....

Task numbering and empty temporal diagram for **Problem 3**. Fill each cell with: 1) the number of the task that is executed; 2) R if a read data transfer is occurring; or 3) W if a write data transfer is occurring. **Important:** each slot in the temporal diagram corresponds to 3 time units.

	0	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60	63	66	69	72	75	78	81	84	87	90	93	96	99	102	105	108	111	114	117	120	123	126	129	132
P0		2	6	6	6	6	7	7	7	7	8	8	8	8	9	9	9	9	R +	R	33	33	33	33	W	R	R	36	36	36	36	W	R **	R	40	40	40	40	W						
P1			10	18	19	20	21	21	21	21	30	30	30	30	31	31	31	31	32	32	32	32	34	34	34	34	37	37	37	37	39	39	39	39	41	41	41	41							
P2	1	3	11	11	11	11	12	13	22	22	22	22	23	23	23	23	24	24	24	24	25	25	25	25	26	26	26	26	27	27	27	27	28	28	28	28	29	29	29	29					
P3		4	5	5	5	5	14	14	14	14	15	15	15	15	16	16	16	16	17	17	17	17	R ++	R	35	35	35	35	W	R	R	38	38	38	38	W	*								

+ At this point 32 and 33 could have been interchanged

++ At this point 34 and 35 could have been interchanged

* At this point P3 could have also stolen 29

108	111	114	117	120	123	126	129
40	40	W					
41	41						
R	R	29	29	29	29	W	

** At this point P1 could have also stolen 28

96	99	102	105	108	111	114	117	120	123	126	129
R	R	28	28	28	28	W					
39	39	40	40	40	40						
29	29	29	29	R	R	41	41	41	41	W	
38	38	38	W								