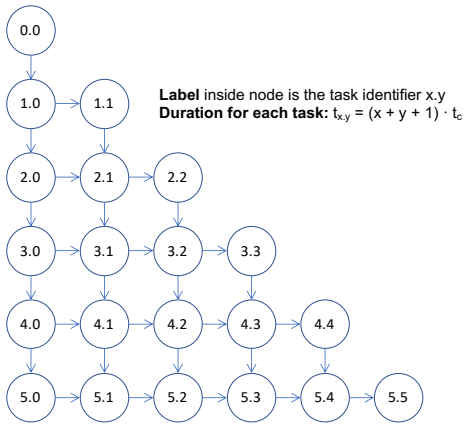
PAR – Final Exam – Course 2021/22-Q2

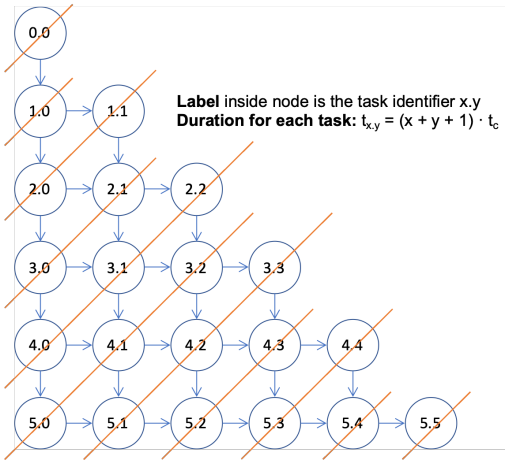
June 20^{th}, 2022

Problem 1 (1.5 points) Given the following Task Dependence Graph associated to the task decomposition applied to a certain program:

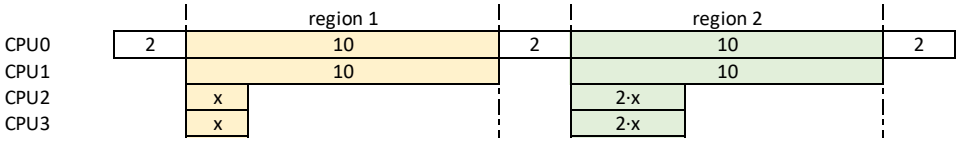


The duration, in time units, of each `compute_task` depends on the value of the row (`x`) and column (`y`) where it is placed (label `x.y`): $t_{x.y} = (x + y + 1) \times t_c$. **We ask you** to compute the values for T_1, T_∞, $Parallelism$ and P_{min}.

Solution: $T_1 = 126 \times t_c$ (sum of the cost of all nodes in the TDG above), $T_\infty = 66 \times t_c$ (defined by the critical path in the TDG, including all nodes in the column on the left and all nodes in the row at the bottom of the TDG above) and $Parallelism = T_1 \div T_\infty = 126 \div 66 = 1.\widehat{90}$. The minimum number of processors to achieve it is $P_{min} = 3$, since we need enough simultaneous processors to execute the widest anti–diagonal in the wavefront execution of the TDG, as shown in the picture below.



Problem 2 (1.5 points) The following diagram plots the timeline for the execution of a parallel application, with two parallel regions, on 4 processors (time evolves from left to right):



Each box represents a burst executed on a processor and the number inside its execution time. There are 3 sequential regions with constant execution time of 2 time units. The execution time is unknown for two of the bursts in $region1$ and in $region2$, both affected by the value x (the timeline shows the case for which

$(2 \times x) < 10$).

- Obtain all the possible values for value x that would lead to an speed-up $S_4 = 2.5$ when the application is executed on 4 processors. Observe that for values $0 < x \leq 5$, $5 < x \leq 10$ and $x > 10$ the two parallel regions have a different contribution in the timeline.

  Solution: In all cases, $T_1 = 2 + (10 + 10 + x + x) + 2 + (10 + 10 + 2 \times x + 2 \times x) + 2 = 46 + 6 \times x$.

  - For x smaller than 5, the execution time for both parallel regions is 10 units, so $T_4 = 2 + 10 + 2 + 10 + 2 = 26$. Therefore from $S_4 = T_1/T_4 = 2.5$ we get $x = 19/6 = 3.16$.
  - For x larger than 5 but smaller than 10, the execution time for the first parallel region is still 10 but for the second one is $2 \times x$. Therefore $T_4 = 16 + 2 \times x$. From $S_4 = T_1/T_4 = 2.5$ it is not possible to get a positive value for x.
  - Finally, for x larger than 10, both parallel regions are dominated by the value of x, being the execution time $T_4 = 6 + 3 \times x$. Again from $S_4 = T_1/T_4 = 2.5$ we obtain $x = 31/1.5 = 20.66$.

- For the values of x in the previous question, obtain the value for $S_{p \to \infty}$, assuming that parallel regions ideally scale with the number of processors.

  Solution: For the two cases above, $T_{p \to \infty} = 6$. Therefore:

  - For $x = 19/6 = 3.16$ we get $S_{p \to \infty} = 65/6 = 10.83$.
  - Similarly, for $x = 31/1.5 = 20.66$ we get $S_{p \to \infty} = 170/6 = 28.33$.

Problem 3 (3.0 points) Given the following data structures that models a graph in which each node can have up to 4 neighbour nodes. Vector `g` holds the information for `N` nodes; for each node the `telem` struct stores the weight of the node `w`, an integer to identify each of the 4 possible neighbours (`north`, `east`, `west` and `south`) and a field that is used to traverse the graph (`visited`) .

```
#define N ... /* large value */

typedef struct {
        int w;
        int north, east, west, south; // -1 value to indicate no neighbour
        char visited;
} telem;
telem g[N];
```

The following code processes all the nodes that are reachable from a given node (0 in the invocation from main):

```
int compute (int label, int weight); // heavy computation, does not access the graph

int traverse_reachable (int label) {
        int ret=0, ret1, ret2, ret3, ret4;
        if (label >= 0) {
            if (!g[label].visited) {
                g[label].visited=1;
                ret1 = traverse_reachable (g[label].north);
                ret2 = traverse_reachable (g[label].east);
                ret3 = traverse_reachable (g[label].west);
                ret4 = traverse_reachable (g[label].south);
                ret = compute(label, g[label].w) + ret1 + ret2 + ret3 + ret4;
            }
        }
        return ret;
}

int main() {
```

```
    ...
    int ret = traverse_reachable (0);
    ...
}
```

We ask you to write an *OpenMP* parallel version of the previous code using a *recursive tree task decomposition* strategy. The implementation should maximize parallelism and minimize the possible synchronization overheads. The implementation must also include a task generation control mechanism based on the recursion level. Use *MAX_ DEPTH* as the value for the maximum recursion level for which tasks must be generated.

Solution:

```
typedef struct {
        int w;
        int north, east, west, south;
        char visited;
        omp_lock_t lock;    // new field to protect the access to the node
} telem;
telem g[N];

int traverse_reachable (int label, int d) { // new argument d to control recursion level
        int ret=0, ret1, ret2, ret3, ret4, tmp;

        if (label >= 0) {
             if (!g[label].visited) {
                omp_set_lock (&g[label].lock);
                if (!g[label].visited) {
                    g[label].visited=1;
                    omp_unset_lock (&g[label].lock);

                    if (!omp_in_final()) {
                        #pragma omp task shared(ret1) final (d>= MAX_DEPTH)
                        ret1 = traverse_reachable (g[label].north, d+1);
                        #pragma omp task shared(ret2) final (d>= MAX_DEPTH)
                        ret2 = traverse_reachable (g[label].east, d+1);
                        #pragma omp task shared(ret3) final (d>= MAX_DEPTH)
                        ret3 = traverse_reachable (g[label].west, d+1);
                        #pragma omp task shared(ret4) final (d>= MAX_DEPTH)
                        ret4 = traverse_reachable (g[label].south, d+1);
                        tmp = compute(label, g[label].w);
                        #pragma omp taskwait     // compute does not need to wait for the graph t
                    } else {
                        ret1 = traverse_reachable (g[label].north, d+1);
                        ret2 = traverse_reachable (g[label].east, d+1);
                        ret3 = traverse_reachable (g[label].west, d+1);
                        ret4 = traverse_reachable (g[label].south, d+1);
                        int tmp = compute(label, g[label].w);
                    }
                    ret = tmp + ret1 + ret2 + ret3 + ret4;
                } else {
                    omp_unset_lock (&g[label].lock);
                }
            }
        }
        return ret;
}

int main() {
    ...
    #pragma omp parallel
```

```
    #pragma omp single
    int ret = traverse_reachable (0, 0);
    ...
}
```

Problem 4 (4.0 points) Given the following sequential code:

```
void saxpy(int n, float a, float * x, float * y) {
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

Assume parameters `x` and `y` point to two completely disjoint vectors, first element of each vector aligned to a memory/cache line boundary. Cache line size is 64 bytes and sizeof(float) is 4 bytes. **We ask you to:**

1. Write a first OpenMP parallel version for the previous sequential function that obeys to an *input block geometric data decomposition* strategy, so that each thread accesses to a single block of consecutive elements. The *block geometric decomposition* should minimise the possible load unbalance that occurs when the size of the vectors `n` is not a multiple value of the number of threads. Assume that `n` is large compared to the number of threads.

   Solution:

   ```
   void saxpy(int n, float a, float * x, float * y) {
     #pragma omp parallel
     {
        int nt;
        int my_id;
        int red;
        int i_start, i_end, n_elems;
        nt    = omp_get_num_threads();
        my_id = omp_get_thread_num();
        n_elems = n/nt;
        red     = n%nt;
        i_start = my_id*n_elems;
        i_start = i_start + (my_id<red)?my_id:red;
        i_end   = i_start + n_elems + (my_id<red);
        for (int i = i_start; i < i_end; ++i)
            y[i] = a*x[i] + y[i];
     }
   }
   ```

2. Write a new parallel version that obeys to an *input/output cyclic geometric data decomposition* strategy. Has the load balance improved with respect to the previous version?

   Solution:

   ```
   void saxpy(int n, float a, float * x, float * y) {
     #pragma omp parallel
     {
       int nt;
       int my_id;
       nt    = omp_get_num_threads();
       my_id = omp_get_thread_num();
       for (int i = my_id; i < n; i+=nt)
         y[i] = a*x[i] + y[i];
     }
   }
   ```

   Both implementantions block and cyclic geometric data decomposition have the same load unbalance: maximum 1 element.

3. Assuming the *input/output cyclic geometric data decomposition strategy* in the previous implementation, and that 1) the parallel system is composed of 2 NUMA nodes, each with a single processor and private cache; 2)

the parallel program is executed with 2 threads (i.e. *thread* i in NUMA node i); 3) the operating system makes use of "first touch" at page level to decide the allocation of memory addresses to NUMA nodes, with one line per memory page; 4) main memory, directories and private caches are empty when the function above starts its execution; and 5) the execution of the iterations assigned to the two threads are interleaved in time, as shown in the two leftmost columns in the table in the answer sheet for the first 4 iterations of the loop. Data coherence across NUMA nodes is provided by a write-invalidate MSU directory-based system. Complete this table with the information of the directory entries where elements of vector `x` and `y` are stored.

4. Write a final parallel version that obeys to an *output block-cyclic geometric data decomposition* strategy, so that the overhead associated with the coherence protocol in a NUMA multiprocessor architecture is minimised, i.e. exploiting the data locality of both input and output data. Is the load balance better/worse than the one achieved in the parallel versions in question 1 and 2?

   Solution:

   We set BS to the number of float elements that fits in a cache/memory line. In this way we avoid extra cache misses accessing both vector `x` and `y` and false sharing accessing `y`.

```
#define BS (64/4)
void saxpy(int n, float a, float * x, float * y) {
  #pragma omp parallel
  {
    int nt;
    int my_id;
    nt    = omp_get_num_threads();
    my_id = omp_get_thread_num();
    for (int ii = my_id*BS; ii < n; ii+=nt*BS)
      for (int i = ii; i < max(n,ii+BS); i++)
         y[i] = a*x[i] + y[i];
  }
}
```

   Block cyclic geometric data decomposition may have `BS` elements of load unbalance. Therefore, it is worse than before.

Student name: .....

Answer sheet for **Problem 4**.

Time	thread	Loop iteration i - vector access	Home NUMA node	Directory entry (sharers bit list)	Directory entry (status)
0	1	1 - read x[1]			
		1 - read y[1]			
		1 - write y[1]			
1	0	0 - read x[0]			
		0 - read y[0]			
		0 - write y[0]			
2	0	2 - read x[2]			
		2 - read y[2]			
		2 - write y[2]			
3	1	3 - read x[3]			
		3 - read y[3]			
		3 - write y[3]			

Solution for **Problem 4**.

Time	thread	Loop iteration i - vector access	Home NUMA node	Directory entry (sharers bit list)	Directory entry (status bits)
0	1	1 - read x[1]	1	10	S
		1 - read y[1]	1	10	S
		1 - write y[1]	1	10	M
1	0	0 - read x[0]	1	11	S
		0 - read y[0]	1	11	S
		0 - write y[0]	1	01	M
2	0	2 - read x[2]	1	11	S
		2 - read y[2]	1	01	M
		2 - write y[2]	1	01	M
3	1	3 - read x[3]	1	11	S
		3 - read y[3]	1	11	S
		3 - write y[3]	1	10	M