PAR – In-Term Exam – Course 2021/22-Q2

April 6^{th}, 2022

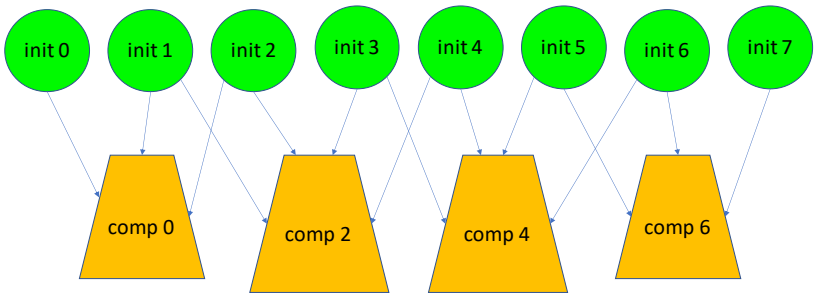
Problem 1 (4 points) Given the following code using *Tareador* for task annotations:

```
#define R  8
#define BS 2
int x[R][N], y[R][N];
...
// initialization phase
for (int i=0; i<R; i++) {
    tareador_start_task ("init");
    for (int k=0; k<N; k++)
        x[i][k] = foo (i); // no other memory accesses inside foo
    tareador_end_task ("init");
}
// computation phase
for (int i=0; i<R; i+=BS) {
    tareador_start_task ("comp");
    for (int ii=max(1,i); ii<min(R-1,i+BS); ii++)
        for (int k=0; k<N; k++)
            y[ii][k] = compute (x[ii][k], x[ii-1][k], x[ii+1][k]);
                       // no other memory accesses inside compute
    tareador_end_task ("comp");
}
```

Assume that the execution time for functions `foo` and `compute` is 2 and 20 time units, respectively, and that the execution cost in time for the rest of the code is negligible. **We ask you to**:

1. Draw the *Task Dependence Graph* (TDG) based on the above *Tareador* task definitions. Include for each task its name followed by the iteration number that generates it and its cost in time units (as a function of `N`).

   Solution:



Where task costs are:
cost (*init i*) = 2 x N where *i=0..7*
cost (*comp 0*) = cost (*comp 6*) = 20 x 1 x N
cost (*comp 2*) = cost(*comp 4*) = 20 x 2 x N

2. Calculate the values for T_1, T_∞ and amount of *Parallelism* (as a function of `N`).

   Solution:

   $T_1 = 8 \times 2 \times N + 2 \times 20 \times 1 \times N + 2 \times 40 \times N = 136 \times N$

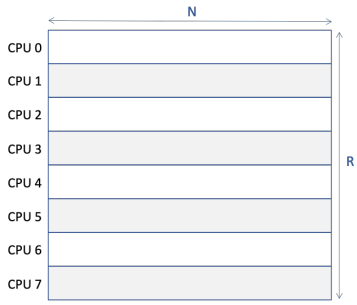
   $T_\infty = 2 \times N + 40 \times N = 42 \times N$, determined by any of the critical paths going through either `comp 2` or `comp 4`. For example, `init 2`–`comp 2`.

   $Parallelism = (136 \times N)/(42 \times N) = 3.2$.

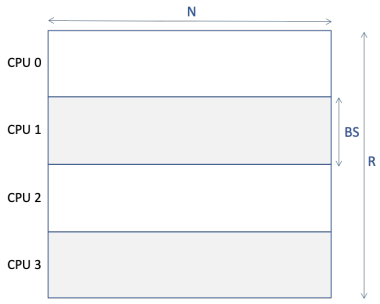
   Although it is not asked in the problem, observe that in this case $P_{min} = 6$. Why?

3. Given the following task allocation for $P = 8$ processors (CPUs) and mapping of matrices `x` and `y` by rows to processors ($R = 8$ and $BS = 2$):

Processor	Task assignment
0	init 0, comp 0
1	init 1, comp 2
2	init 2, comp 4
3	init 3, comp 6
4	init 4
5	init 5
6	init 6
7	init 7



Matrix x mapped to 8 processors



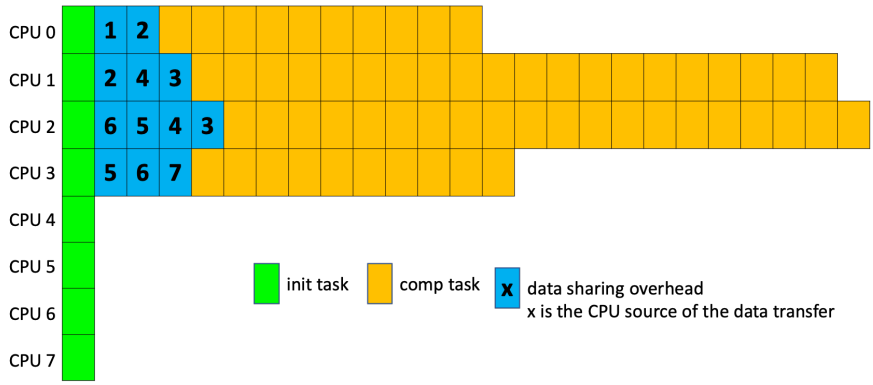
Matrix y mapped to 4 processors

Write the expression that determines the execution time, T_8 as a function of `N`, clearly identifying the contribution of the computation time and the data sharing overheads, assuming the data sharing model explained in class in which the overhead to perform a remote memory access is $t_s + t_w \times m$, being t_s the start-up time, t_w the time to transfer one element and m the number of elements to be transferred; at any time, each processor can simultaneously make one remote access to a different processor and serve one remote access from another processor.

Solution:

First let's find out the critical path considering data sharing overheads. Observe that `comp 0` executed by processor 0 has to perform two remote accesses (to get the row initialized in `init 1` by processor 1 and to get the row initialized in `init 2` by processor 2); `comp 2` executed by processor 1 has to perform three remote accesses (to get the row initialized in `init 2` by processor 2, the row initialized in `init 3` by processor 3 and the row initialized in `init 4` by processor 4); `comp 4` executed by processor 2 has to perform four remote accesses (to get the row initialized in `init 3` by processor 3, the row initialized in `init 4` by processor 4, the row initialized in `init 5` by processor 5, and the row initialized in `init 6` by processor 6); and finally, `comp 6` executed by processor 3 has to perform three remote accesses (to get the row initialized in `init 5` by processor 5, the row initialized in `init 6` by processor 6 and the row initialized in `init 7` by processor 7). Therefore it is clear that the critical path is determined by processor 3 executing `init 2` and `comp 4`,

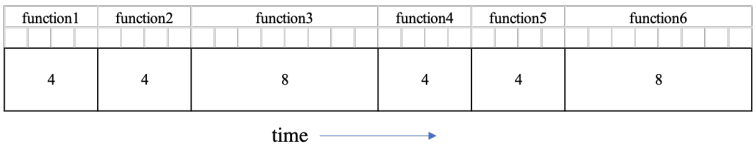
The following temporal diagram shows the execution taking into account the data sharing overheads. There is also indicated a possible scheduling of data transfers to show that it is possible to make it.



The data sharing overhead cost is given by the expression: $T_8^{datasharing} = 4 \times (t_s + t_w \times N)$

so $T_8 = T_8^{comp} + T_8^{datasharing} = (2 \times N + 40 \times N) + (4 \times (t_s + t_w \times N))$

Problem 2 (2 points) The following figure shows the timing diagram for the sequential execution of an application on 1 processor:



The figure has a set of rectangles, each rectangle representing the serial execution of a function with its associated cost in time units. In a first attempt the programmer has been able to parallelize functions 1, 2, 3, 4 and 5, which can be ideally decomposed; **function 6 remains sequential. We ask you to answer the following questions**:

1. What is the parallel fraction ϕ?

   Solution: $\phi = \frac{4+4+8+4+4}{4+4+8+4+4+8} = \frac{24}{32} = \frac{3}{4}$

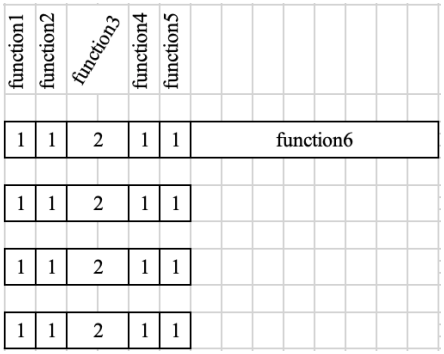
2. Which would be the maximum speedup that you could achieve based on Amdahl's law ($S_{p\to\infty}$)?

   Solution: $S_{p\to\infty} = \frac{1}{(1-\phi)} = 4$

3. Draw the time diagram of an ideal parallel execution of the application to obtain maximum speed-up with 4 processors. Which is the value for S_4 that is achieved?

   Solution:

   The proposed parallel execution is shown below, giving the ideal speed–up for 4 processors: $S_4 = \frac{T_1}{T_4} = \frac{32}{14} = \frac{16}{7} = 2.28$.



In a second attempt the programmer has been able to fully parallelize all funtions in the application (total workload of 32 time units), with ideal scalability. Please **answer the following two questions:**

4. Which should be the workload assigned to each processor when parallelized with $P = 8$ processors and strong scaling?

   Solution: $work\_load\_per\_thread = \frac{32}{8} = 4$ time units.

5. Which should be the workload assigned to each processor when parallelized with $P = 8$ processors and weak scaling?

   Solution: $work\_load\_per\_thread = 32$ time units.

Problem 3 (4 points) Given the following code excerpt, including OpenMP directives, to be executed on a UMA architecture with 8 processors, each one with a 1 MB fully-associative cache memory, and cache coherence maintained with the simplest write–invalidate MSI protocol explained in class, sharing the access to 16 GB of main memory:

```
#define NUM_THREADS 8
#define N 262144                 // N multiple of NUM_THREADS

int a[N], b[N];
struct {
    int zeros[NUM_THREADS];
    int positives;
} count;

#pragma omp parallel num_threads(NUM_THREADS)
{
    int id = omp_get_thread_num();
    int num_elems = N / NUM_THREADS;
    for (int i=id*num_elems; i < (id+1)*num_elems; i++) {
        a[i] = a[i] * b[i];
        if (a[i] == 0) count.zeros[id]++;
        if (a[i]>0)
            #pragma omp atomic
            count.positives++;
    }
}
```

Observe that each implicit task executes a chunk of `num_elems` consecutive iterations of the loop. Assume that processor i executes implicit task with $id = i$. Also assume that each integer (`int`) variable occupies 4 bytes, a memory (cache) line occupies 32 bytes, and that the initial address of vectors `a` and `b` as well as structure `count` are aligned with the start of a cache line. All other variables are stored in registers (not in memory). Processors have no cached copies of `a`, `b` and `count` in their private cache memories before starting the execution of the parallel region. We ask you to **answer the following questions:**

1. How many *BusRd*, *BusRdX*, *BusUpgr* and *Flush* commands will be placed BY EACH ONE of the 8 processors on the shared interconnection network (bus) due to accesses to **vectors `a` and `b`**.

   Solution: Each vector `a` and `b` occupies (262144 elements * 4 bytes/element) = 1 MB. Since elements are equally distributed among the 8 processors, each local cache will need 128 KB for each vector, which correspond to 128 KB / 32 bytes/line = 4096 lines. Therefore each processor will place on the bus 4096 *BusRd* command to read its elements of `a` and the same for `b`; since `a` is also written, 4096 *BusUpgr* commands (not *BusRdX* since the access results in a cache hit) will also be placed on the bus from each processor. Both vectors fit in the caches, so no *Flush* commands are generated for the lines of vector `a`. In total: 8192 BusRd and 4096 BusUpgr, no BusRdX and no Flush.

2. How many memory lines does **structure `count`** occupy? Does the access to `count` in the program causes *true* and/or *false* sharing? Briefly reason your answer.

   Solution: Structure `count` occupies (8 + 1) int * 4 bytes/int = 36 bytes. Therefore it occupies 2 cache lines. The access to `count.positives` causes *true* sharing, since that field is updated by all processors whenever a positive value is computed for `a`. The access to `count.zeros` causes *false* sharing, since each processor is accessing to a different element of the vector `zeros` but all of them reside in the same cache line.

3. In order to reduce the coherence traffic that is caused by the accesses to **vector `count.zeros`** the programmer has changed the definition of `count` and the access to it, as follows:

```
struct {
    int zeros[NUM_THREADS*PADDING];
    int positives;
} count;
...
        if (a[i] == 0) count.zeros[id*PADDING]++;
...
```

Which would be the most appropriate value for constant `PADDING`? How many *BusRd*, *BusRdX*, *BusUpgr* and *Flush* commands will be placed BY EACH ONE of the 8 processors on the shared interconnection network (bus) due to accesses to `count.zeros`?

Solution: Since this access is causing a *false* sharing problem, coherence traffic can only be eliminated if we use padding, so that each element of vector `zeros` occupies a different memory line. We achieve this with `PADDING=8`, so that the elements accessed by different processors are $8 \times 4 = 32$ bytes apart. With this each processor will place one *BusRd* and one *BusUpgr* command in the shared bus during the whole execution of the parallel region.

If the multiprocessor architecture is upgraded to a NUMA system with 8 nodes, each node with a single processor, a private cache memory of 1 MB, and a portion of main memory of 2 GB, and all variables (vectors `a` and `b` and structure `count`) are physically mapped to NUMA node 0 by the operating system (first touch policy) before starting the execution of the parallel region, with no cached copies in any of the nodes. We ask you to **answer the following questions:**

4. How many entries in the directory **of each NUMA node** will be used to store the coherence information for **vectors `a` and `b`**? After the execution of the parallel region, how many bits in the *sharers list* of each of these directory entries will be set to 1 and in which state will those memory lines be?

   Solution: As calculated before, each vector `a` and `b` occupies (262144 elements * 4 bytes/element)/32 bytes/line = 32768 lines. Since all elements are mapped in NUMA node 0, 32768*2=65536 entries will be used in that node. No entries will be used in the rest of nodes. Since each line is only accessed by the processor of a single NUMA node, only one bit in the *sharers list* will be set to one, with M state for vector `a` and S state for vector `b`.

5. After executing the program the programmer realized that the access to `count.positives` was creating a performance bottleneck, so she/he proposed the following program transformation for the parallel region, making use of a per–thread copy `tmppos` of the shared variable `count.positives`, which is accumulated into it before exiting the parallel region:

```
#pragma omp parallel num_threads(NUM_THREADS)
{
    int tmppos = 0;
    int id = omp_get_thread_num();
    int num_elems = N / NUM_THREADS;
    for (int i=id*num_elems; i < (id+1)*num_elems; i++) {
        a[i] = a[i] * b[i];
        if (a[i] == 0) count.zeros[id]++;
        if (a[i]>0) tmppos++;
    }
    #pragma omp atomic
    count.positives = count.positives + tmppos;
}
```

Assume that the processor in NUMA node 0 is the first executing `#pragma omp atomic` and that its execution ensures read/write atomicity in the access to `count.positives` (i.e. while one processor is accessing to it inside the `pragma` no other processors will be able to access it). Which of the following statements is/are true? **(selected wrong statements penalize the mark that you can obtain in this question)**

(a) The proposed transformation can never improve performance since the number of of positives found in vector `a` does not change.

(b) If the number of positive results in vector `a` is much larger than 1, the ONLY improvement in performance comes from the fact that `#pragma omp atomic` has been removed from the loop body.

(c) The last processor x executing `#pragma omp atomic` will first send a $RdReq_{x \to 0}$ command, which will provoke a $Fetch_{0 \to y}$ command to the remote node y that performed the previous update. As a consequence, $Dreply_{y \to 0}$ and $Dreply_{0 \to x}$ will be generated in order to get the updated line in processor x. The line is also updated in main memory of node 0.

(d) After that, the same processor x will send an $UpgrReq_{x \to 0}$ to change the state of the line in the home node and an $Invalidate_{x \to y}$ command to invalidate the copy in cache of node y, which will reply with an $Ack_{y \to x}$ to notify the completion of the command.

(e) At the end of the parallel region the directory entry associated to the memory line containing `count.positives` will be in state M with only the bit associated to node x in the sharers list active to 1.

Note: *$Command_{a \to b}$ refers to a NUMA command sent from node a to node b.*

Solution:

{False, False, True, False, True}