Preparation questions:

1. The main reason for moving to parallel computing is the physical constraints preventing frequency scaling like power consumption in serial computation, therefore in addition to achieve better performance we need to scale in number of cores/processors instead of improving he frequency for a single core/processor

2.
   a. The HAL processor is under the category of MIMD, meaning that it contains several cores that are capable of computing different tasks on different data streams simultaneously.

   b. The HAL processor belongs to the RISC (Reduced Instruction Set Computer) architecture. The general concept of RISC architecture is that such a computer has a small set of simple and general instructions, rather than a large set of complex and specialized instructions.

   c. The first level of parallelism in use will be ILP (Instruction Level Parallelism) since this is a MIMD processor.

      The second level of parallelism in use is DLP (Data Level Parallelism) since it can also be used as SIMD for example when creating multiple instances of the same task on different data and schedule those tasks to different core instead of using a ‘for loop’.

d.

The main problem with shared memory is that scalability beyond thirty-two processors is difficult from the cache-coherence point of view.

The HAL processor doesn't have any cache therefore there is no cache-coherence problem.

This can be achieved efficiently because the HAL processor make it possible for each core to access the memory in a single cycle (unless there are conflicts between 2 cores and then an access can be delayed by another cycle but this happens rarely) therefore there is no need for caches.

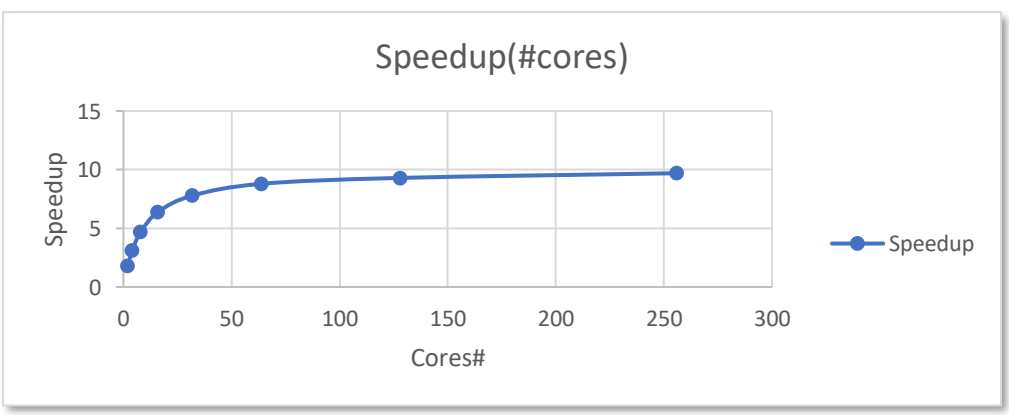
3.

a.

The speedup is given by Amdahl's law $speedup(\#cores) = \frac{1}{\frac{A}{\#cores}+(1-A)}$ when $A = 0.9$ is the part of the program that can be parallelized.

#cores	speedup
2	1.8
4	3.1
8	4.7
16	6.4
32	7.8
64	8.8
128	9.3
256	9.7

b.



c.

We can see that after 56 cores, adding other cores doesn't increase the speedup significantly.

4.

a.

We discuss only 3 types of tasks:

- **Regular tasks:**
  - runs a single thread sequentially on a single core
  - return ‘true’, ‘false’ or ‘don’t care’
- **Duplicable tasks:**
  - can be executed in parallel as multiple instances on multiple cores
  - amount of instances (Quota) is programmable during a previously allocated task
  - each core receives an instance number to identify himself
  - all instances of the task must terminate before a termination token is generated
  - a duplicable task only indicates termination
- **Dummy tasks:**
  - not allocated to a core and terminates immediately when enabled
  - dummy tasks are used to allow more complex task enabling conditions
  - like duplicable tasks, dummy tasks only indicate termination

b.

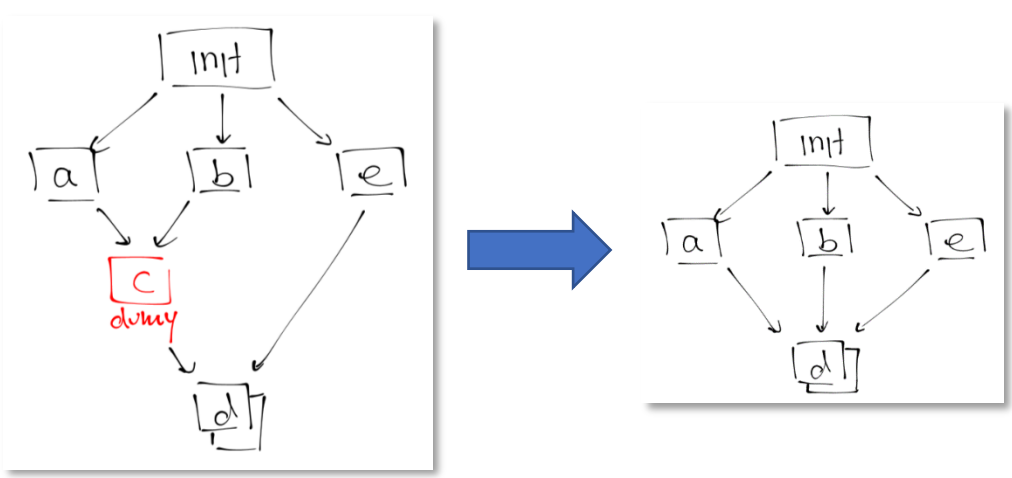
The task is a ‘duplicable task’ named ‘f’ and can start only after the termination of task ‘func’ regardless of its returned value.

c.

This is because a duplicable task has many instances and each one of them may lead to a different returned value, we cannot ensure all instances will be consistence from the returned value point of view.

5.

Dummy tasks are used to simplify complex conditions on other tasks so the second graph is the abstraction of the first one and it is the dependencies graph



6.

```
regular task a ()

dummy task dum1 (d/u & e/u)
dummy task dum2 (c & dum1)
regular task b (a/u | dum2)

duplicable task c (b/u)
regular task d (b/u)
regular task e (b/u)

regular task f (a/u)

dummy task dum3 (g/u & h/u)
regular task g (f/u | dum3)
regular task h (f/u | dum3)
```

7.

a.

This isn't written anywhere but I assume there are:

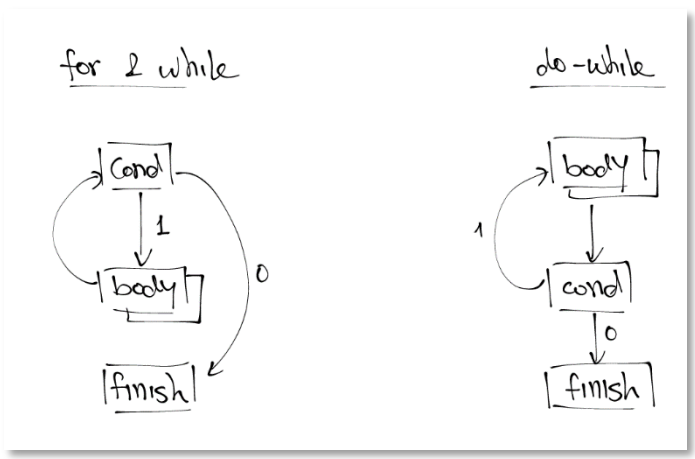
- For loops
- While loops
- Do-while loops

b.

We will create a duplicable task such that each instance will simulate a single iteration of the loop using $HAL\_TASK\_INST \ = loop\ index$ for the loop body and another regular task for the condition checking.

The order between the tasks ('cond' & 'body') vary between the different loop types.

c.




```
regular task cond (init/u | body)
duplicable task body (cond/1)
regular task finish (cond/0)
```

```
duplicable task body (init/u | cond/1)
regular task cond (body)
regular task finish (cond/0)
```