Parallelization and lock-free programming

What properties does the program executed by thread `t` have?

```
AtomicInteger x = new AtomicInteger(0);
```

thread `t`

```
int v;
do {
  v = x.get();
  v = v + 1;
} while (!x.compareAndSet(v - 1, v));
```

1. it is starvation free
2. it is lock free
3. it is lock free, and hence also wait free
4. it is wait free, and hence also lock free

What properties does the program executed by thread `t` have?

```
AtomicInteger x = new AtomicInteger(0);
```

thread `t`

```
int v;
do {
  v = x.get();
  v = v + 1;
} while (!x.compareAndSet(v - 1, v));
```

1. it is starvation free
2. [✓ it is lock free]
3. it is lock free, and hence also wait free
4. it is wait free, and hence also lock free

What properties does the program executed by thread `t` have?

```
AtomicInteger x = new AtomicInteger(0);
```

thread `t`

```
int v;
for (int i = 0; i < 10_000; i++) {
  v = x.get();
  v = v + 1;
  if (x.compareAndSet(v - 1, v))
    break;
}
```

1. it is starvation free
2. it is lock free
3. it is lock free, and hence also wait free
4. it is wait free, and hence also lock free

What properties does the program executed by thread `t` have?

```
AtomicInteger x = new AtomicInteger(0);
```

thread `t`

```
int v;
for (int i = 0; i < 10_000; i++) {
  v = x.get();
  v = v + 1;
  if (x.compareAndSet(v - 1, v))
    break;
}
```

1. it is starvation free
2. it is lock free
3. it is lock free, and hence also wait free
4. it is wait free, and hence also lock free

Note that the increment may fail after trying 10’000 times.

What does function `pm(L)` compute?

```
pm([X|[]]) -> X;
pm([X|[Y|[]]]) -> if X > Y -> X; true -> Y end;
pm(L) -> M = length(L) div 2, {A, Z} = lists:split(M, L),
         Me = self(),
         spawn(fun () -> Me ! pm(A) end),
         spawn(fun () -> Me ! pm(Z) end),
         receive B -> B end, receive Y -> Y end, pm([B, Y]).
```

1. the sum of elements in `L`
2. the maximum of elements in `L`
3. the minimum of elements in `L`
4. a sorted copy of `L`

What does function `pm(L)` compute?

```
pm([X|[]]) -> X;
pm([X|[Y|[]]]) -> if X > Y -> X; true -> Y end;
pm(L) -> M = length(L) div 2, {A, Z} = lists:split(M, L),
         Me = self(),
         spawn(fun () -> Me ! pm(A) end),
         spawn(fun () -> Me ! pm(Z) end),
         receive B -> B end, receive Y -> Y end, pm([B, Y]).
```

1. the sum of elements in `L`
2. the maximum of elements in `L`
3. the minimum of elements in `L`
4. a sorted copy of `L`

How many tasks may execute in parallel when computing the factorial of `n`?

```
class Factorial extends RecursiveTask<Integer> {
  int n; // number to compute factorial of
  protected Integer compute() {
    if (n <= 1) return 1;
    Factorial f = new Factorial(n - 1);
    f.fork();
    return n * f.join();
  }
}
```

1. `n`! (the factorial of `n`)
2. `n`
3. it depends on the number of available cores
4. there is practically no parallelism

How many tasks may execute in parallel when computing the factorial of n?

```
class Factorial extends RecursiveTask<Integer> {
  int n; // number to compute factorial of
  protected Integer compute() {
    if (n <= 1) return 1;
    Factorial f = new Factorial(n - 1);
    f.fork();
    return n * f.join();
  }
}
```

1. n! (the factorial of n)
2. n
3. it depends on the number of available cores
4. there is practically no parallelism

How many processes may execute in parallel when computing the factorial of `n`?

```
fact(1) -> 1;
fact(N) ->
         Me = self(),
         spawn(fun () -> Me ! fact(N-1) end),
         receive F -> N*F end.
```

1. `n`! (the factorial of `n`)
2. `n`
3. it depends on the number of available cores
4. there is practically no parallelism

How many processes may execute in parallel when computing the factorial of `n`?

```
fact(1) -> 1;
fact(N) ->
        Me = self(),
        spawn(fun () -> Me ! fact(N-1) end),
        receive F -> N*F end.
```

1. `n`! (the factorial of `n`)
2. `n`
3. it depends on the number of available cores
4. there is practically no parallelism

How many tasks may execute in parallel when computing the sum of integers from 1 to n?

```
class Sum extends RecursiveTask<Integer> {
  int m, n; // sum integers from m to n
  protected Integer compute() {
    if (m > n) return 0;
    if (m == n) return m;
    int mid = m + (n-m)/2; // mid point
    Sum lower = new Sum(m, mid);
    Sum upper = new Sum(mid+1, n);
    lower.fork(); upper.fork();
    return lower.join() + upper.join();
  }
}
```

1. 2^n (2 to the power of n)
2. n^2 (the square of n)
3. n
4. there is practically no parallelism

How many tasks may execute in parallel when computing the sum of integers from 1 to `n`?

```java
class Sum extends RecursiveTask<Integer> {
  int m, n; // sum integers from m to n
  protected Integer compute() {
    if (m > n) return 0;
    if (m == n) return m;
    int mid = m + (n-m)/2; // mid point
    Sum lower = new Sum(m, mid);
    Sum upper = new Sum(mid+1, n);
    lower.fork(); upper.fork();
    return lower.join() + upper.join();
  }
}
```

1. 2^n (2 to the power of `n`)
2. n^2 (the square of `n`)
3. `n`
4. there is practically no parallelism