1.

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1);
    b[i] = a[i] - a[i-1];
}
```

Assuming a and b are not aliased. There is a true dependence inside the loop and a loop carried true dependence.

This can be mitigated using Loop Splitting:

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1);
}
#pragma omp parallel for
for (i=1; i<N; i++) {
    b[i] = a[i] - a[i-1];
}
```

If a and b are aliased the code simplifies to:

```
a[0] = 0;
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1) - a[i-1];
}
```

which results in a loop carried true dependence. Using a little more analysis this can be simplified to:

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1) - 2.0*(i-1)*(i-2);
}
```

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
```

```
    a[i] = 2.0*(i*(i-1) - (i-1)*(i-2));
}
```

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*(i-1)*(i - (i-2));
}
```

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*(i-1)*2;
}
```

2.

```
a[0] = 0;
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<N; i++) {
        a[i] = 3.0*i*(i+1);
    }
    #pragma omp for
    for (i=1; i<N; i++) {
        b[i] = a[i] - a[i-1];
    }
}
```

This code results in a race condition since the second loop does not wait for the first one to finish and some of the values may not have been written by the time they are used in the second loop. The easiest way to mitigate this would be to remove the 'nowait'.

```
a[0] = 0;
#pragma omp parallel
{
    #pragma omp for
    for (i=1; i<N; i++) {
        a[i] = 3.0*i*(i+1);
    }
    #pragma omp for
    for (i=1; i<N; i++) {
        b[i] = a[i] - a[i-1];
```

```
    }
}
```

Another solution would be to apply the simplification from 1) so all the values can be calculated in a single Loop Iteration.

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 3.0*i*(i+1);
    b[i] = 3.0 * (i-1) * 2;
}
```

3.

```
#pragma omp parallel for
for (i=1; i<N; i++) {
    x = sqrt(b[i]) - 1;
    a[i] = x*x + 2*x + 1;
}
```

This code is correct, even if a and b are aliased they are indexed in the same way which only results in an anti-dependence inside the loop, which is fine.

4.

```
f = 2;
#pragma omp parallel for private(f,x)
for (i=1; i<N; i++) {
    x = f * b[i];
    a[i] = x - 7;
}
a[0] = x;
```

This code is mostly correct, even if a and b are aliased they are indexed in the same way which only results in an anti-dependence inside the loop, which is fine. The final assignment to a[0] is a problem though since x is a private variable inside the parallel region which results in an undefined state after the parallel region.

5.

```
sum = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
```

```
        sum = sum + b[i];
    }
```

This code is correct but could be improved using Reduction.