PARALLELIZATION AND VECTORIZATION IN GCC

1. This is an anti-dependence (write after read). In iteration 0, `a[2]` is read and `a[0]` is written into. In iteration 2, `a[4]` is read and `a[2]` is written into. Therefore, `a[2]` is first read, and then written into. Hence write after read. The distance is 2.

2. The subscript tests are performed from outermost to innermost subscripts. The dependence dump says that there is 'no dependence'. This is because the ZIV test returns the result that the accesses are independent. GCC dependence analysis bails out the instance it figures out that a subscript is independent. So if ZIV is independent in the second subscript, no test will be performed for the first subscript

3. There is a true dependence (read after write). In iteration 0, `a[0]` is read and `a[1]` is written into. In iteration 1, `a[1]` is read and `a[2]` is written into. Therefore, `a[1]` is first written into and then read from. Hence read after write. The code can't be vectorized as the dependence distance is less than vectorization factor.

4. There is a true dependence (read after write). But since the dependence distance (5) is greater than vectorization factor for int (4), the loop can be vectorized. Howerver, if we convert the type of a to 'short int', the vectorization factor becomes 8, and vectorization will be prohibited.

5. Without loop interchange, the accesses are not with deterministic constant stride, and therefore the code is not vectorized. But with loop interchange, the strides become constant and the innermost loop (j loop, where j exhibits anti-dependence) is vectorized.

6. The current loop parallelizes for x = 2. If we change the loop iterations to parallelize the code for x = 4, the parallelization fails. If we change the loop bound to 401, it will be parallelized with x = 4. Therefore, there must be minimum 100 iterations assigned to each.

7. There is actually no dependence as the vectorization factor is less than the dependence distance. However, N is an invariant within the loop, and therefore its scalar evolution is not performed. Banerjee's

dependence implementation cannot determine the absence of dependence, and parallelization fails. However, GRAPHITE handles the invariant parametric inequality and the code is parallelized.

8. In the loop, S1 can be parallelized while S2 cannot be parallelized. But normally we cannot distribute this loop because of the conditional statement. But in GRAPHITE, the conditional code is converted into inequality which is merged with the loop bounds by CLOOG, giving the code

   ```
   for (i=0; i<min(N, M); i++)
      S1
   for (i=M; i<N; i++)
      S2
   ```

   Here, the loop enclosing S1 can be parallelized.