1. (a) 
   \[ \text{for } k = 1, N, 2 \]
   \[ \text{for } j = 1, k \]
   \[ \text{for } i = 1, N \]
   \[ C(j, k) = C(j, k) + A(i, k) \cdot B(i, j) \]
   \[ C(j, k+1) = C(j, k+1) + A(i, k+1) \cdot B(i, j) \]
   \[ \text{end for} \]
   \[ \text{end for} \]
   \[ \text{for } i = 1, N \]
   \[ C(k+1, k+1) = C(k+1, k+1) + A(i, k+1) \cdot B(i, k+1) \]
   \[ \text{end for} \]
   \[ \text{end for} \]

   (b) Strides of access in innermost loop are now N, N, 0, respectively for A, B, C. This will result in poor spatial locality due to the large stride of access. It would be best to make k the innermost loop, i.e. ijk or jik permutations; this would make the strides of access in innermost loop 1, 0, 1, respectively for A, B, C.

2. The cache capacity is 512K words; linesize is 8 words. The cache capacity in blocks is 512K/8 = 64K.

2D Tiling

Considering each tile in execution, i.e. innermost two loops i,j:
Number of misses for C = (512/8)*512 = 32K
Number of misses for B = (512/8)*1 = 64 (one partial row, repeatedly accessed for different i)
Number of misses for A = 1*512 = 512 (one partial column)

So for the first iteration of k, we have 32K+64+512 misses, which is also the number of occupied cache blocks. Since this is less than cache capacity, none of the blocks is displaced from cache. As k is varied from 1 to 4096, the same 32K elements of C are repeatedly accessed for each value of k, but different rows(columns) of B(A) are accessed for different k. Assuming the cache uses an LRU policy, no additional misses will occur for C as k is varied. SO for a fixed (it,jt), we have:
Number of misses for C = 32K*1 = 32K
Number of misses for B = 64 *4096 = 256K
Number of misses for A = 512*(4096/8) = 256K

For a single value of (it,jt), the number of misses is 544K, which means the cache capacity of 64K blocks is greatly exceeded. So no reuse is possible with the outer tiling loops for A or B. The total number of misses for all (it,jt):
Number of misses for C = 32K*2*8 = 512K
Number of misses for B = 256K*2*8 = 4096K
Number of misses for A = 256K*2*8 = 4096K

3D Tiling

Counting misses innermost to outermost loop, consider one tile, i.e. fixed (it,jt,kt):
Number of misses for C = 1*(256/8)*256 = 8K
Number of misses for B = (256/8)*1*256 = 8K
Number of misses for A = 256*(256/8)*1 = 8K

As kt is varied, the same block is C is accessed for each kt, while different parts of A and B are accessed. There will be reuse for C since the total number of blocks accessed for a single kt value is only 24K, much less than cache capacity.

Number of misses for C = 8K
Number of misses for B = 8K*16 = 128K
Number of misses for A = 8K*16 = 128K

As jt is varied, the same sets of blocks of A are accessed again while different blocks of B and C are accessed. However, since the number of cache lines accessed for each jt is 264K, no reuse of A is possible for different jt.

Number of misses for C = 8K*16 = 128K
Number of misses for B = 128K*16 = 2048K
Number of misses for A = 128K*16 = 2048K

Finally, as it is varied, different parts of A and C are accessed. Although the B array does not contain the i index, no reuse is possible since the number of blocks needed (2048K) exceeds cache capacity. So we have:

Number of misses for C = 128K*4 = 512K
Number of misses for B = 2048K*4 = 8192K
Number of misses for A = 2048K*4 = 8192K

So total number of misses for 2D tiling (8704K) is lower than 3D tiling.

3. For sequential execution, the array A is too large to fit in cache, since it has 1024K words. Assuming row-major representation of A, and a dot-product implementation of matrix-vector multiplication, the sequential execution time is:

\[
T_{seq} = 2\times1024\times1024 \text{ [arithmetic ops]} + 1024\times(1024/8)\times32 \text{ [cache miss cost]} = 6291456
\]

On two processors, the portion of A on each processor is still twice as large as cache capacity. Each processor sends/receives 1024/2 words to the other. So total time for a matrix-vector multiply is:

\[
T_2 = 2\times1024\times1024/2 \text{ [arithmetic ops]} + (1024/2)\times(1024/8)\times32 \text{ [cache miss cost]} + 10000 + (1024/2)\times10 \text{ [comm. cost]} = 3160848
\]

So speedup on 2 processors is 6291456/3160848 = 1.99

On four and more processors, the local portion of array A can fit within cache and so will not incur cache miss costs for repeated matrix-vector multiplications.

\[
T_p = 2\times1024\times1024/P + 10000*(P-1) + (1024/P)\times(P-1)
\]

So we have:

\[
T_4 = 555056; \quad S_4 = 11.33 \text{ (highly super-linear!)}
\]
\[
T_8 = 333040; \quad S_8 = 18.89 \text{ (also very super-linear)}
\]
\[
T_{16} = 282032; \quad S_8 = 22.30 \text{ (still super-linear)}
\]

4. (a) For L1, the outer i-loop is not parallelizable; only the inner j-loop can be parallelized. For L2, there is potential parallelism in both loops. So for L2, there will be lower overhead by parallelizing the outer i-loop.
for (i=0;i<1024;i++)
{
t = 0;
#pragma omp for reduction(+:t)
for (j=0;j<i;j++) t = t + a[i][j]*x[j];
x[i] = (b[i]-t)/a[i][i];
}

L1: Lower-triangular Solve

#pragma omp parallel for private(j,t)
for (i=0;i<1024;i++)
{
t = 0;
for (j=0;j<1024;j++) t = t + a[i][j]*x[j];
b[i] = t;
}

L2: matrix-vector multiply

(b) For L1, the sequential run time is 1024 [divisions] + 2*(1024*1025)/2 [multiplications and subtractions] = 1050624
The parallel run time is 1024*100 [scheduling overhead] + 2*1024 [division/subtract by master] + (1/4)*2*(1024*1025)/2 [multiply/adds] = 366848
Speedup = 1050624/366848 = 2.86

For L2, sequential run time is 2*1024*1024 = 2097152
Parallel run time is 100 [scheduling overhead] + 2*1024*1024/4 = 524388
Speedup = 2097152/524388 = 3.99