1. (a)

1. Item	Type	Address
A[0].a	float	0
A[0].b	float	4
A[0].c	float	8
A[0].d	int	12
A[1].a	float	16
A[1].b	float	20
A[1].c	float	24
A[1].d	int	28
A[2].a	float	32
……	……	……
A[6].d	int	108
A[7].a	float	112
A[7].b	float	116
A[7].c	float	120
A[7].d	int	124

(b) It depends on the size of the cache and the size of each cache block. There are 4 types of cache misses: compulsory miss, conflict miss, capacity miss, and coherence miss. Below, I'll discuss when these misses are likely to occur depending on different scenarios:

(1) Compulsory miss:

Say we have a 64KB shared cache with 16-byte cache blocks, then there will be 8 compulsory cache misses. Because the cache block size is so small that each cache block contains the data for only one array element. On each thread access, there is a compulsory miss.

(2) Conflict miss:

For such a small array, conflict misses are not very likely to happen. Since array elements are likely to be held in the same cache block or neighboring cache blocks.

(3) Capacity miss:

For such a small array, capacity misses are not very likely to happen. It will only occur when the size the cache is extremely small that it cannot hold the whole array.

(4) Coherent miss:

Will not happen since we only have one shared cache.

(c) Normally, the number of cache misses will be small. Suppose we have a 64KB shared cache with 16-byte cache blocks, then there will be only 8 compulsory misses And this number decreases to 1 as the size of the cache block increases to 128 bytes, in which case all 8 array elements are loaded in the cache at one time. However, if the cache size is very small, then there might be many conflict misses or capacity misses. One way to deal with this is to rewrite

the array of structure to structure of arrays, (i.e., organize variables a, b, c, d in each array element into corresponding arrays Array[a], Array[b], Array[c], Array[d] and put them into a single structure), under the assumption that the threads are more likely to access the same fields in each element simultaneously. Therefore, each cache block loaded can be best utilized, and the number of compulsory misses, conflict misses, or capacity misses will be effectively reduced.

2. (i) Each core can only execute one instruction flow at a time, many of the threads will sit there waiting for their turn to run even if they are ready.

(ii) Creating and destroying threads both have overhead. If we have a large number of threads, this overhead can be huge.

(iii) Threads may compete for resources and it can lead to deadlocks.

3. (i) If the number of threads is smaller than the number of cores, then you are not utilizing the whole parallelization potential of your hardware.

(ii) If you have only a single thread, then a system call of the thread will block the whole process and CPU may be idle during the block period.

(iii) If instead of using multiple threads, we create multiple processes. Then, these processes require more memory and resources and they cannot share resources in the same way threads share resources within the same process.

4. (a) Depending on whether memory is shared or distributed among cores, the algorithms have slight differences. But the idea for both is to use the Sieve of Eratosthenes algorithm. Before diving into the parallelized code, I would like to first write out the serial code and then transform it into parallelized code by determining which parts in the serial code can be effectively parallelized.

(0) Serial Pseudocode

(i) Initialize a list of integers from 2 to n by setting each element in the list to 0.

(ii) Iterate p from 2 to sqrt(n)

(iii) If the number p is not marked as composite

(iv) Iterate from p to n and mark all multiples of p as composite by setting the value of the corresponding index in the list to 1

(v) Iterate from 2 to n and output as primes the indices whose corresponding values in the list is 0.

(1) Shared memory

Steps (i), (iv) and (v) in the serial code can be parallelized.

(i) The list can be initialized in parallel. The range [2, n] can be equally split into p parts e.g., [2, 2 + n/p], [2 + n/p + 1, 2 + 2n/p] … We can round n/p to the nearest integer if n%p != 0. Then, each thread is responsible for initializing the list element in its responsible range, and this can be carried out in parallel.

(iv) The composite numbers can be marked in parallel. The range [p, n] can be equally split into p parts e.g., [p, p + n/p – 1], [p + n/p, p + 2n/p – 1], … We can round n/p to the nearest integer if n%p != 0. Then, each thread is responsible for marking the multiples in its responsible range, and this can be carried out in parallel.

(v) Printing out the prime numbers can be executed in parallel. The range [2, n] can be equally split into p parts e.g., [2, 1 + n/p - 1], [2 + n/p, 1 + 2n/p] … We can round n/p to the nearest integer if n%p != 0. Then, each thread is responsible for printing out the prime numbers (indices in the list with the corresponding value as 0) in its responsible range, and this can be carried out in parallel.

(2) Distributed memory

Only step (v) can be effectively parallelized. Since each thread has its own list of numbers in its private distributed memory, each thread has to initialize the list and mark the composite numbers individually. However, after all composite numbers has been marked, the rest of the list, which contains prime numbers, can be printed out in parallel as in the shared memory scenario above.

(b)

(1) We can certainly run the algorithm in (a) to find all the prime numbers between 1 and x, and then determine whether x is a prime number.

(2) Or more efficiently, we can use the idea of trial division and split the range [2, sqrt(x)] equally into p parts e.g., [2, 1 + (sqrt(x)-2)/p], [2 + (sqrt(x)-2)/p, 1 + 2(sqrt(x)-2)/p], … Then each thread iterates through its responsible range and determine if there exists an integer that evenly divides x. If any of the thread finds an integer i such that x%i == 0, then x is composite.

Otherwise if none of the thread finds an integer that evenly divides x, then x is prime.

(3) An even more efficient way is similar to method (2), but instead of searching the interval [2, sqrt(x)], we first find out all primes in the range [2, sqrt(x)] and then divides these primes evenly among threads. Since if x can be evenly divided by any composite number, it can also be evenly divided by a prime number which in turn evenly divides this composite number. Then each thread iterates through its responsible split of primes and determine if there exists a prime that evenly divides x. If any of the thread finds a prime number p such that x%p == 0, then x is composite. Otherwise x is prime. Note that however this method requires additional storage to store the primes in the range [2, sqrt(x)], which might in turn incurs access overhead.

5. Using Amdahl's law: If F is the fraction of a calculation that is sequential, then the maximum speed-up that can be achieved by using P processors is $\frac{1}{F+\frac{1-F}{p}}$. Let $\frac{1}{F+\frac{1-F}{p}} >= 3$, we get p >= 6. Thus, for three-fold speed-up, 6 CPUs are required. For five-fold speed-up, it is not possible. Since that means executing the parallel part in 0 unit of time, which cannot be achieved using any number of CPUs.

6. As the level goes from L1 to L3, the hit rate becomes more important and the access latency becomes less important. Since L1 cache is closest to the CPU and is designed to provide fast response (a few CPU cycles), Whereas L3 cache is designed to prevent very expensive memory access, and the hit rate becomes more important (otherwise you will have to look up in the memory).