Xi Liu, xl3504, Homework 1

1.
to compute the sum of n values
method 1: since each core computes a partial sum my_sum and sends that result to the master core to form a global sum, the master need to carry out $n-1$ receives and $n-1$ adds

method 2: to add n values together, the tree like solution has

$$\text{number of receives} = \text{number of additions} = \text{depth of tree} = i$$

$$n/2^i = 1$$

$$n = 2^i$$

$$\log_2 n = i$$

$$\text{number of receives} = \text{number of additions} = \log_2 n$$

reason 1: since $\log_2 n \in \mathcal{O}(n-1)$, method 2 is more time efficient than method 1
reason 2: in method 2, there is more load-balancing among the cores, each core is doing approximately the same amount of work, and the master core are doing less work than the work done by the master core in method 1

2.
reason 1: even load is better than uneven load, because when load is uneven, some cores can be overloaded while other cores are left idle, the overloaded core can be overheated
reason 2: load balancing can exploit the existence of multiple cores. if 1 core is broken, the work of that broken core may be assigned to other cores
reason 3: using load balancing, more overall instructions can be executed in same amount of time

3.
superscalar cores use dynamic multiple issue, in which different threads can use multiple functional units in a multiple issue core at the same physical instant

a) maximum number of threads that can be executed in parallel
$= \text{number of cores} = \boxed{8}$

b) maximum number of threads that can be executed in parallel = (two threads per core due to hyperthreading)*(number of cores) $= 2 \cdot 8 = \boxed{16}$
since two-way hyperthreading allows simultaneous execution of 2 threads per core

c) maximum number of threads that can be executed in parallel
$= \text{number of cores} = \boxed{8}$
performance of the processor in c) may not be as good as the performance of the processor in a) since superscalar is not supported for the processor in c)

4.
speculative execution is needed for superscalar processor to work. since to decide which instructions to execute at the same time, the processor must make a guess first and execute based on the guess. for example:

$sum = a + b;$
$var = *ptr;$ // ptr is a pointer

the system might make the guess that ptr does not hold the address of sum, then the two assignments can be executed in parallel. but if the guess is wrong, the assignment of $var = *ptr$ need to be re-executed

5.
The cost of SIMD is less than MIMD. SIMD has less complexity than MIMD. SIMD systems apply the same instructions to multiple data items when oper-

ate on data streams. since shader functions (in graphics processing pipeline) applied to graphics stream (such as vertices) often results in same flow of control, GPUs can optimize performance by using SIMD parallelism

6.
reason 1: the density of transistors on integrated circuits was increasing
reason 2: as the size of transistor decreases, the speed of the transistors can increase
reason 3: instruction-level parallelism, each core can have simultaneous execution of instructions, pipelining and multiple issue

7.
cache coherence need to be maintained, so coherence protocol is needed. if there is no coherence, a variable can be stored in 2 caches of 2 different cores, then if 1 core updates the variable, another core may not know the change so it may still use the outdated data. but usually coherence protocol affect performance negatively in the sense that it slows the system down
reason 1: snooping cache coherence requires information about cache line updates to be broadcast across across all cores, extra time might be needed to transmit the data through the interconnect
reason 2: directory-based cache coherence uses directory to store each cache line's status. if a variable is updated, there is extra time needed to consult the directory, and cores that store a variable need to be contacted if a cache variable is updated
reason 3: false sharing can happen. since CPU caches operate on cache lines but not variables. suppose that we need to store some computations into a vector in a loop, divide the work of the iterations into multiple cores. the data of the vector is stored in 1 cache line. then each assignment to the vector invalidates the cache line, which leads to increased main memory accesses