Xi Liu, xl3504, Homework 2

1.
yes, the threads can write to the same physical memory address,
the threads contained within the same process share the same virtual address space, so the threads can access the same data. threads within the same process use the same page table, so a given physical frame that is addressable to one thread within a process is also addressable to another thread within the same process

2.
no, the processes cannot write to the same physical memory address,
different processes have different virtual address spaces. for each distinct process, the operating system manages a different page table for the process, so different processes' virtual pages map to different physical frames

3.
the required speedup cannot be reached
based on Amdahl's law:
let F be the fraction of a calculation that is sequential
let S be the speedup
let p be the number of cores available to use

for a linear speedup

$$T_{parallel} = \frac{T_{serial}}{p}$$

$$\frac{T_{serial}}{T_{parallel}} = p$$
$$\frac{T_{parallel}}{T_{serial}} = \frac{1}{p}$$

$$\begin{aligned}
S = \frac{T_{serial}}{T_{parallel}} &= \frac{T_{serial}}{\text{time of parallelized part} + \text{time of unparallelized part}} \\
&= \frac{T_{serial}}{(1-F)T_{parallel} + (F)T_{serial}} \\
&= \frac{1}{(1-F)T_{parallel}/T_{serial} + F} \\
&= \frac{1}{(1-F)/p + F}
\end{aligned}$$

$$\begin{aligned}
\lim_{p\to\infty} S &= \lim_{p\to\infty} \frac{1}{(1-F)/p + F} \\
&= \frac{1}{F} \\
&= \frac{1}{0.4} \\
&= 2.5 < 4
\end{aligned}$$

as number of cores p approaches infinity, the maximum speedup is obtained, the maximum speedup is 2.5 and is less than 4

4.

yes, coherence overhead can be caused if two parallel threads try to update two different variables due to false sharing. cache coherence is enforced at the "cache-line level". the caching system may be unaware of individual variables within each cache line. each time when there is a write in the cache line, if the cache line is also stored in another processor's cache, the entire cache line will be invalidated

5.

if a core runs a task that takes a long time, a lot longer than the time to finish the 3 other tasks, then the other 3 cores with shorter tasks can soon finish and become idle, then using 2 cores can take the same amount of time as using 4 cores (see problem 9)

when the problem size is sufficiently small, running the program with less than 4 cores can give the same speedup as running the program with 4 cores. since CPU computations on small problem sizes can take less time than the time cost of thread creation, message passing, communications, or synchronization involved when using more cores

6.

load imbalance will be more severe as the number of synchronization points increase. a synchronization such as pthread_cond_wait() involves putting the calling thread to sleep and wait for some other thread to call pthread_cond_signal(), then the sleeping thread is idle for some amount of time while some other threads execute for some amount of time, causing load imbalance

7.

no, a sequential implementation with higher efficiency can be slower than the other parallel implementation with lower efficiency

let E be the efficiency

let S be the speedup

let p be the number of cores available to use

$$\begin{aligned} E &= \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} \\ &= \frac{T_{serial}}{p \cdot T_{parallel}} \end{aligned}$$

/* the maximum value of E is 1 since the maximum speedup is the linear speedup $T_{parallel} = T_{serial}/p$,

$p \cdot T_{parallel} = T_{serial}$, then

$$\begin{aligned} E &= \frac{T_{serial}}{p \cdot T_{parallel}} \\ &= \frac{T_{serial}}{T_{serial}} \\ &= 1 \end{aligned}$$

*/
when the implementation uses only 1 core, $p = 1$, then

$$\begin{aligned} E &= \frac{T_{serial}}{p \cdot T_{parallel}} \\ &= \frac{T_{serial}}{1 \cdot T_{parallel}} \\ &= \frac{T_{serial}}{1 \cdot T_{serial}} \\ &= 1 \end{aligned}$$

so sequential code have the maximum efficiency, but parallel code is usually faster than sequential code on sufficiently large problem sizes

8.
no, we cannot assume that the threads will always finish at the same time
reason 1: a thread that encounters more cache misses can become slower since the storage locations of the different data are different, different scenarios of false sharing can be caused since the threads operate on different data
reason 2: since data are different, a thread that operate on a larger data size can take longer time
reason 3: a thread that encounter page faults can become slower. if a page is not present and has been swapped to disk, the operating system will need to swap the page into memory in order to handle the page fault
reason 4: since 2 thread execute on 2 different cores, the instruction set architecture and physical conditions of the cores can be different, the newer core that had been previously used less may perform faster

9.

a.

2 is the smallest number of processes needed to execute in parallel to get the best performance

if 4 processes are used:

time interval (ms)	process 1	process 2	process 3	process 4
$[0, 50]$	S_1			
$[50, 150]$	P_1	P_2	P_3	P_4
$[150, 2050]$				P_4
$[2050, 2100]$	S_2			
$[2100, 2200]$	P_5	P_6	P_7	P_8
$[2200, 2600]$		P_6		
$[2600, 2650]$	S_2			

if 2 processes are used:

time interval (ms)	process 1	process 2
$[0, 50]$	S_1	
$[50, 150]$	P_1	P_4
$[150, 200]$	P_2	P_4
$[200, 300]$	P_3	P_4
$[300, 2050]$		P_4
$[2050, 2100]$	S_2	
$[2100, 2200]$	P_5	P_6
$[2200, 2300]$	P_7	P_6
$[2300, 2400]$	P_8	P_6
$[2300, 2600]$		P_6
$[2600, 2650]$	S_2	

b.

let $S(p)$ be the speedup when p processes are used

$$\begin{aligned} T_{serial} &= \sum(\text{run time of each task}) \\ &= (50 + 3(100) + 2000 + 50 + 3(100) + 500 + 50)ms = 3250ms \end{aligned}$$

based on part a, $T_2 = T_4 = T_8 = 2650ms$
so $S(2) = S(4) = S(8)$

$$\begin{aligned} S(2) = S(4) = S(8) &= \frac{T_{serial}}{T_{parallel}} \\ &= \frac{3250ms}{2650ms} \\ &= \frac{65}{53} \approx 1.2264 \end{aligned}$$

c.
span is the longest path of dependence in the directed acyclic graph, longest sequence of operations that cannot be executed in parallel

$$span = T_{\infty} = 2650ms$$

work is total amount of time spent on all instructions

$$work = T_1 = 3250ms$$

d.

$$\begin{aligned} parallelism &= \frac{T_1}{T_{\infty}} \\ &= \frac{3250ms}{2650ms} \\ &= \frac{65}{53} \approx 1.2264 \end{aligned}$$