CSCI-UA.0480-051: Parallel Computing
Midterm Exam (Oct 12th, 2023)
Total: 100 points

Important Notes- READ BEFORE SOLVING THE EXAM

- **If you perceive any ambiguity in any of the questions, state your assumptions clearly and solve the problem based on your assumptions. We will grade both your solutions and your assumptions.**
- **This exam is take-home.**
- **The exam is posted on Brightspace, assignments section, at the beginning of the Oct 12th lecture.**
- **You have up to 23 hours and 59 minutes from the beginning of the Oct 12th lecture to submit on Brightspace (in the assignments section). That is, you must submit before 2pm Friday Oct 13th.**
- **You are allowed only one submission, unlike assignments and labs.**
- **Your answers must be very focused. You may be penalized for wrong answers and for putting irrelevant information in your answers.**
- **You must upload one pdf file.**
- **Your answer sheet must have a cover page (as indicated below) and one problem answer per page (e.g., problem 1 on a separate page, problem 2 in another separate page, etc.).**
- **This exam has 4 problems totaling 100 points.**
- **The very first page of your answer is the cover page and must ONLY contain:**
  - **You Last Name**
  - **Your First Name**
  - **Your NetID**
  - **Copy and paste the honor code showed in the rectangle at the bottom of this page.**

Honor code (copy and paste to the first page of your exam)

- You may use the textbook, slides, and any notes you have. But you may not use the internet.
- You may NOT use communication tools to collaborate with other humans. This includes but is not limited to Messenger, E-mail, etc.
- Do not try to search for answers on the internet it will show in your answer, and you will earn an immediate grade of 0.
- Anyone found sharing answers or communicating with another student during the exam period will earn an immediate grade of 0.
- **"I understand the ground rules and agree to abide by them. I will not share answers or assist another student during this exam, nor will I seek assistance from another student or attempt to view their answers."**

Problem 1

a. [9 points] Suppose we have k transistors. We can use those k transistors to build either one big core with high frequency or build several smaller cores with lower frequency. Nowadays, designers opt for the second option. State three reasons for why they do that.

[3 points per reason. You need only three reasons]

- Smaller cores with lower frequency consume less power than a big core with high frequency.
- Several smaller cores can exploit task-level parallelism.
- Smaller cores are easier to design and test.
- Executing threads in parallel on several cores can hide memory latency.

b. Suppose we have two threads belonging to the same process and we have three choices on where to execute them: on a single superscalar core, on a two-way hyperthreading core, or on two physical superscalar cores.

- [4] When will it give better performance if we execute the two threads on a single superscalar core?
    - [4] There is a dependency between the two threads. That is, one of them cannot execute till the other is done.
      [Other reasons that make sense will be counted as correct too]
- [4] When will it give better performance if we execute the two threads on the two-way hyperthreading core?
    - The two threads are independent and can execute in parallel.
    - The two threads require the similar data so they will help each other filling the caches if any.
    - The two threads do not require the same execution units. For example, one thread needs int operations while the other needs floating points. Or, one thread is compute bound while the other is memory bound.
      [Only two reasons are needed. Two points for each.]
- [4] When will it give better performance if we execute the two threads on the two physical superscalar cores?
    - Threads are independent and can execute in parallel.
    - Threads do not write the same cache block triggering cache coherence.
    - Not much communication between the two threads.
      [Only two reasons are needed. Two points for each.]

c. [4] Suppose we have eight cores on a shared-memory machine: two of the cores are just pipelined, two of them are superscalar, and the remaining four are two-way hyperthreaded. What is the maximum number of processes that can be executed in parallel on that whole system? Explain how you reached this number.

[correct number 1 point … Justification: 3 points]
Twelve processes can be executed in parallel.
To get the maximum number of processes, each process needs to be single-threaded.
Pipelined cores can each execute one thread: 2 x 1 = 2
Superscalar cores can each execute on thread: 2 x 1 = 2
Each of the four two-way hyperthreading can execute two threads simultaneously:
4 x 2 = 8

This makes the total number of threads = 2 + 2 + 8 = 12.
Since we assumed that each thread is single threaded, then we can have 12 processes executing in parallel.

Problem 2

a. [8] The pipelined core has several stages: fetch, decode, issue, execute and commit. The next generation after the pipelined core is the superscalar core. Superscalar has several execution units. Yet, the superscalar also has fetch, decode, issue, and commit stages like what we had in the pipelined core. Are there any modifications that need to be made to these stages (fetch, decode, issue, and commit) to make the superscalar get higher performance than the pipelined? If not, explain why not. If yes, explain what these modifications are. Assume no speculative execution.

[2] Yes, there are modifications that need to be done to these stages in the superscalar. To get higher performance, we need to make the several execution units in the superscalar as busy as possible. To do so, we need the following modifications to the other pipeline stages in the superscalar.

[1] Fetch phase needs to be able to fetch more than one instruction at a time.

[1] The decode phase needs to be able to decode more than one instruction at a time.

[2] The issue phase needs to be able to detect independent instructions and send them to their execution units.

[2] The commit stage needs to ensure that the instructions write their results in order. The issue phase may lead to out-of-order execution.

b. [8] Suppose we have a sequential program that we need to parallelize. We want the speedup of the parallelized program relative to the sequential program to be two. What is the minimum number of cores we shall use? And what will be the fraction of the sequential part of the parallelized version?

[1] We will use Amdahl's law: $S = 1/(F + (1-F)/p)$

[5]
$S = 2$
$2 = 1/(F + (1-F)/p) \rightarrow F + (1-F)/p = 0.5$
Since one core will never lead to speedup of two, let's try two cores.
$F + (1-F)/2 = 0.5 \rightarrow 0.5F = 0$ And this cannot be true because there is no program where the sequential fraction is 0. So, let's try three cores.
$F + (1-F)/3 = 0.5 \rightarrow$ This makes $F = 0.25$.

[2] So, we need to use three cores. This will give a speedup of two and a sequential fraction of 25%.

c. [3] If a parallel program does not have any synchronization points at all, is load balancing still important? Justify.

[1] Yes, it is.

[2 points: one reason is enough] For two reasons:

- Any program ends with one synchronization point. For example, the main() function will not end till all threads are done. So, with load imbalance, several fast threads will have to wait for slow ones. This affects the performance.
- When a core executes a bigger thread (i.e. more computations) than the others, this core becomes warmer. The hardware will reduce its speed to avoid burning. This leads to lower performance.

d. [6] Why we cannot reach perfect load balancing in parallel programs even if we do our best? State three reasons.

[2 points per reason … three reasons only are needed] Even if we give each core the same threads, we may not reach perfect load balancing, for the following reasons:

- Some threads may get cache misses while the others don't.
- Some threads may get page faults while others don't.
- Some threads may run on weaker cores than others, if we have processor with different cores (e.g., performance cores vs efficiency cores).
- The code may have if-else, and some threads will go in a different path that contains more (or less) computations than others.
- Accessing the memory (and even the large L3 cache) may lead to different latency depending on where the core is relative to the banks (Non-Uniform Memory Access).

Problem 3

a. [4] Cache coherence is not enough to deal with critical sections. This is why we need programming tools like locks, mutex, etc. State two reasons for that.

- [2] The critical section may consist of more than one instruction. Coherence does not deal with that.
- [2] Coherence ensures that no two cores update their similar cache blocks at the same time. But it does not two threads from updating the same data item. It just serializes them, which will still lead to wrong result.

b. [6] Suppose we have the following two pieces of code. We have two threads belonging to the same process. Each thread will execute the same piece of code as the other thread but potentially on different data. Shall we make the two threads execute code#1? Or code#2? Justify your answer. Assume the two arrays A and B are shared between the two threads. Each thread has a unique ID, 0 or 1, stored in a variable, myID, private to each thread. Also assume each thread will execute on a separate superscalar core with its own level 1 cache. Level 2 is shared among the cores and there is no level 3 cache.

Code#1	**Code#2**

for(i = 0; i < N; i++)
  if(myID == 0)
    if(i is odd)
      A[i] += B[i];
  else if(myID == 1)
    if(i is even)
      A[i] += B[i];
	for(i = 0; i < N; i++)
	  if(myID == 0)
	    if(i < (N/2))
	      A[i] += B[i];
	  else if(myID == 1)
	    if(i >= (N/2))
	      A[i] += B[i];

[2] Code #2 is better.
[4] Code#1 makes the two threads update two neighboring elements. For example tread 0 updates A[0] and thread 1 updates A[1]. It is very likely that the two threads belong to the same cache block, triggering the coherence protocol and leading to lower performance. This happens much less frequently in code#2.

c. [15] For each one of the following statements, state if the statement is true or false, and give 1-2 sentences justification.

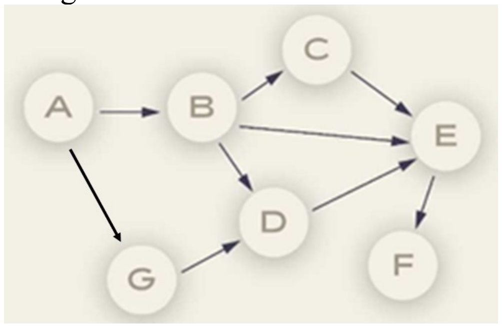
1) For distributed memory machines, when a process consists of two threads, each one of these two threads can execute on a different node.
2) Two processes running on shared-memory machines can access each other's memory.
3) Snoopy coherence protocol can be implemented on a non-bus interconnect.
4) In a distributed memory machine, coherence can be more severe then among processes than in shared-memory machines if the programmer is not careful.
5) Speculative execution (i.e., branch prediction) is the main reason for out-of-order execution in a superscalar core.

[For each statement: T/F: 1 point … Justification: 2 points]

Statement	**T/F**	**Justification**
1	F	Threads of the same process must share the memory. In distributed memory machines, each one of the nodes has its own memory.
2	F	The OS ensures that each process has its own virtual address space. Each process does not know of the existence of the other process. For example, your media player does not know of the existence of your web browser, even though they both run on your laptop which is a shared memory machine.
3	F	Snoopy requires a bus. Otherwise, caches won't be able to "snoop" on the accesses and act.
4	F	There is no coherence among processes because there is no shared memory among them.
5	F	Out-of-order execution happens when the issue phase detects independent instructions and issues them to execution units, potentially out of order. This happens even if there is no branch prediction.

Problem 4

Suppose we have the following DAG:



Each vertex represents a task. The directed arrows represent dependencies among tasks. The following table contains the time taken for each task as well as the number of machine language instructions executed for each task.

Task	Time (in seconds)	# instructions
A	5	10
B	10	30
C	5	10
D	5	20
E	5	10
F	5	15
G	20	50

a. [8] Suppose we have three cores, each one is a superscalar, state which core will execute which task to achieve the minimum execution time. And calculate that execution time.

- [2] One core executes: A, B, C
- [2] The other core executes: G, D, E, F
- [2] The third core remains idle.

There are other correct assignments for the first two cores. The main thing is that one core must execute B and C while the other must execute G.

[2] Given the above, the total execution time will be: 40

b. [6] Some tasks take the same amount of time, yet have a different number of instructions, for example tasks C and D. Why do you think this may happen? State at least two reasons.

[3 points per reason, only two reasons are needed. Other reasons that make sense will be accepted too]

- Some may be executing int instructions while the other floating point instructions take longer.
- Some may be doing add/sub instructions while the others may be doing mul/div instructions which take longer.
- Some may have load/store instructions which take longer than any arithmetic/logic instructions.

c. [5] Is there an arrow we can remove from the DAG that makes the execution faster than the execution you calculated in question a above? If yes, which arrow? What will be the new execution time? And which core will execute which task? If not, explain why not. Assume we still have three cores each of which is a superscalar core.

[3] You need to be careful that a DAG representing a program execution must start with one vertex and end with one vertex. So, removing arrows like G→D or B→C will make the DAG illegal. [You can also show that the other arrows won't lead to higher performance].
[2] Given that, there is no arrow that can be removed and make for a faster execution while making the DAG legal.

d. [6] Can we gain something if we combine tasks E and F? Justify your answer.
[Hint: Thing about the execution of this DAG on multicore processor.]

[2] Yes, we will gain something.
[4] There is an arrow from E to F. This means F depends on data generated by E, which means there is communication between E and F. As we said in class, communication is expensive. If we combine E and F, we will remove this communication as both tasks will be combined into one task (The aggregation step in the PCAM model).
[If you say no because the DAG will still give the same execution time, this means you did not consider the execution on real machine. However, you get 4 points for this problem].