Homework 2

Jack Bosco

February 1, 2024

1. What is one difference between concurrency and parallelism? Give an example of a concurrent task and a parallel task.

   Concurrency is being able to run multiple tasks at the same time. Parallelism is the ability to split up a job into subtasks which are executed at the same time. An example of a concurrent task is a popup from an application while you are working on a different application. An example of a parallel task is when you are running a divide and conquer algorithm like merge sort, and split each division into seperate threads.

2. If a program runs *more than* p times faster on p processors than on a single processor, then it is said to be a program that achieves **superlinear speedup**.

   In *most* circumstances this is impossible to achieve. Why do you think this is the case?

   If I have p processors, and I can make a program run in n steps on one processor then the fastest I can have that program run is $\frac{n}{p}$ using all of the processors.
   This is because I am limited by the number of processors I have. Perhaps in some special cases the program can faster than $\frac{n}{p}$, like $\frac{n}{p^2}$ or something, but I am not aware of such a case.

3. Consider the first program we examined in the course which involved the summation of a lot of numbers. How might parallelizing this task be different on a SIMD, MIMD, or Shared-Memory computer?

   Say there are n items in the list:

   - SIMD: You have a lot of split memory, so try to split up the list into sublists until you have $\frac{n}{2}$ sublists of size 2. Each sublist is given the instruction to take the sum of the two items in the list. In the next step, the sum for each sublist is returned and propagated up so we have $\frac{n}{4}$ sublists. This is repeated until there is only one sublist, where the sum is our answer.
     If we have less data compartments than $\frac{n}{2}$, divide the work into vectors and do the sum as a subprocesses in each vector. If there is a remainder, make a second subprocesses to handle the vector of different size as the others.
   - MIMD: This one is essentially the same as SIMD, but with a lot more overhead because we are sending instructions between nodes in the cluster. Take the same steps as before, but we need to return the results with network sockets or however the nodes are communicating. Realistically this will be slower than SIMD.
   - Shared-Memory: We have to define a new thread for each vector when dividing up the list. The good news is when there is an uneven division, summing up the differently sized vector is essentially the same as all the rest. The bad news is we are limited by the number of processors, which is more expensive than data, so this will likely be more expensive than SIMD.

4. Suppose you have written a long, coherent essay about English literature that spans many paragraphs. You are trying to parallelize the task of displaying this long essay to the screen. What issues could there be?

   The issue is with undefined behavior when you are trying to do multiple things at once. You will need to do a bunch of locks and holds to make sure one line of text doesn't appear out of order. This may be slower than not doing any parallelization, depending on overhead.

5. In real CPU hardware, caches do not store individual variables but cache lines or cache blocks, which consists of enough storage for multiple variables.

   Common cache line sizes are 32, 64, or 128 bytes. Cache lines can only be fetched/written a whole line at a time and CPUs will only keep track of entire cache lines at a time.

   The goal of such a design is to improve performance since it allows one operation to bring in a whole block of memory to the cache.

   However, this means:

   - When loading a variable into cache, the entire line must be loaded
   - When writing a variable from cache into main memory, the entire line will be considered updated.

   In a shared-memory system, how might this affect the problem of cache coherence? What impact does it have on performance?

   Cache coherence means there is a chance that when one processor writes to memory, another processor may have an outdated copy of that region of memory in it's cache. The cache lines design makes cache coherence more complicated because it means that all the caches' lines must be of the same size. When loading a variable into cache, you have to check if you are loading the value of another variable and ask the other caches if there was a change. When writing a variable, you have to see if there are other variable you are writing to and tell other caches you are writing on the other variables as well.

   This will negatively impact performance since when you want to write to one variable, you may inadvetently write over other variables in memory as well. One cache will have to tell other caches where in memory it exactly wrote to, and all the caches must be synced up by blocks.