Cody Adams

CSC478-01

Final Exam

1. [--done

Traditional Memory is responsible for committing successful execution to shared memory in a way that does not negatively affect any other processes/threads. This is possible by flagging a section of code as “transactional”. Once this is done, all execution is done speculatively and then checked for contention. If contention did not in fact occur, the speculative writes are committed, if not, all work is abandoned. TM has specific areas of performance loss; there is overhead in tracking “transactional” code and in checking after it is when if it is to be applied or discarded. Also, there is time lost when the code is marked to be discarded and new code must be fetched and executed. There is idle time whenever “transactional” sections are being log, as this can only be accessed sequentially.

]

2. [--done

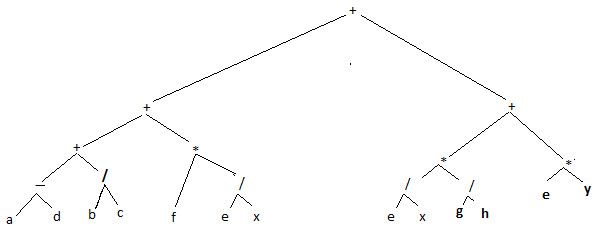
The tree has a depth of four, so requires four time units to execute and only requires five PEs.

1st stage (5 PEs): (a – d), (b / c), (e / x), (e / x), (g / h)

2nd stage (4 PEs): (a – d) + (b / c), [f \* (e / x)], (e / x) + (g / h), (e \* y)

3rd stage (2 PEs): ([(a – d) + (b / c)] + [f \* (e / x)]), (e / x) + (g / h) + (e \* y)

4th stage (1 PEs): ([(a – d) + (b / c)] + [f \* (e / x)]) + (e / x) + (g / h) + (e \* y)



]

3. [--done

Superlinear speedup exists wherever p process can yield a speedup greater than p. Common causes include performance increases yielded by good locality or shared memory design/management. These can be architectural (CUDA). Likewise, architectural advantages in the PEs can result in superlinear speedup. For example, in the Intel NetBurst architecture (more recently seen in the newest Core i-Series CPUs), there are two limited-purpose CPUs that run at least twice as fast as the general purpose core(s) which can be used to quickly execute certain kinds of instructions (more time in faster PEs than slower ones).

]

4.

[--done

This sounds like a good place for the parallel prefix sum algorithm. Essentially, each of the n numbers is summed with its neighbor, and the data is pushed into the element on the right. This is done over and over again until finally the full sum propagates to the right-most element.

For example: to find that 1 + 2 + 3 + 4 + 5 + 6 = 21



]

5. [--done

SIMD:

Using 1 / [ f + ( 1 – f ) / p ]

1 / (.2 + .8/10) = 3.571

MIMD:

Using 1 / SEQ + PAR / p + f / p

1 / (.08 + .12/10 + .8/10) = 5.814

]

6. [--done

Line two has a flow dependency on line 1  
Line four has a flow dependency on line 3  
Line three has a anti-dependency on line 2

There exist no dependencies in the second code section.

]

7. [--done  
  
shared scale, M, x1, x0, x[N], h[M];

private i, j;

scale = M/(x1 - x0);

done := done -1

for i = 1 to N-1

create DOELEMENT(i, j, x[N], h[M]);

i = n;

call DOELEMENT(i, j, x[N], h[M]);

while (!done)

...

procedure DOELEMENT(i, j, x[N], h[M])

j = floor(scale\*(x[i] - x0));

critical

h[j] = h[j] + 1;

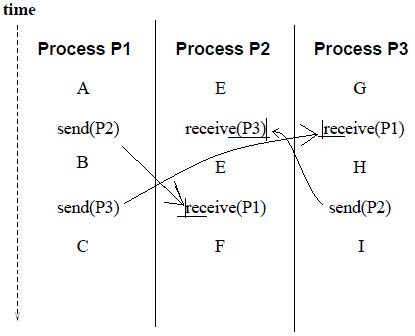
end critical

return

end procedure

]

8. [--done  
(a) I worked up both of these, not sure which is better (feedback?)



(b) Represented above in **BOLDFACE**,   
P1-S3 🡪 P3-R1 🡪 P3-H 🡪 P3-S2 🡪 P2-R3

]

9. [--done

Semaphores – Low level Lock that can protect against race conditions, but cannot help when trying to avoid deadlock. It uses the locks and unlocks to ensure that processes get access to data in the right order so data does not corrupted.

Low-level locks – Examples included semaphores and mutexes. Often reliant on hardware support for efficient implementation, they are difficult to effectively use and suffer from high overhead and a tendency to reduce code to running sequentially.

Critical/end critical – Code that must be atomic is flagged and can only be accessed by one process/thread at a time.

Produce/consume – Code that uses what is produced by previous code can be parallelized using the producer/consumer model. As flag is kept to mark if the producer or consumer should go, based on what data is in the shared memory.

]