Parallel Programming Assignment 1

Madhur Singhal
2015CS10235

February 2018

1 Question 1

1.1 Part 1

I assume that since only seven steps are given to us, they are the only ones that take measurable time. Thus one iteration of the loop will take 9 ns. Accordingly the whole loop will take **9000 ns** since there is no pipelining and everything executes serially.

1.2 Part 2

I assume that we mean a 7 stage pipelined processor which is superscalar (ie can execute two instructions at once). I also assume that the "fetch operands" fetches both operands at once , "add" adds both of them and "store result" stores the result of this. This is assumed since the first two operand names have an s at the end which signifies that they want plural number of operands.
With these assumptions the processor will take 506 cycles to go through the loop since 500 cycles will be taken to accept the whole input into the processor and the last 6 will be taken to process the 12 unfinished inputs present in the pipeline. Since we have a pipeline thus the cycle time will be 2 ns (maximum of the individual operation time). Thus total time = $506 * 2\,ns =$ **1012 ns**.
It is impractical to show the pipeline state graphically so I'll just describe it. For the first cycle we just take two iterations of the loop (maybe 0 and 1) as input and fetch the four values. For each of the next six cycles these inputs keep proceeding to a new stage and new inputs come to the back thus after seven cycles the pipeline is totally filled. For the next cycles two iterations of the loop come in as input and two iterations of the loop go out as having been processed.

2 Question 2

I assume that memory accesses for the result matrix are negligible since they are an order less than those for others. For the **Y** matrix we would require a memory access each time but for the **X** matrix, a memory access is required only oncer per four executions since the word size is 4. We cannot use the cache to store the **Y** matrix since going once through the innermost loop will remove the stored elements by accessing more than the 32 KB of cache we have (since our cache can store 32 KB which is equal to two columns of the matrix).

$$Mem\,Accesses = n^3 + \frac{n^3}{4}$$

$$Num\,Operations = 2n^3$$

We are given than N = 4K (assuming to be equal to 4096).

$$Performace = \frac{Num\,Operations}{Time\,Taken} = \frac{Num\,Operations}{Mem\,Time + Comp\,Time}$$

$$= \frac{2 * (4096)^3}{(100\,ns) * (4096)^3 * \frac{5}{4} + (1\,ns) * 2 * (4096)^3}$$

$$= \frac{2}{127}\,GFLOPS = 15.748\,MFLOPS$$

3 Question 3

The second option is true.

Suppose two barriers are invoked one after the other. Look at the moment when two processes are waiting at the fifth line and the third one goes through second line and changes process_arrived to 3 and unlocks S. One of the three processes will then go through lines 6 to 12 and change process_left to 1. The key problem happens now, this process can go into it's second barrier invocation and acquire the lock on lines 2 to 4 changing proces_arrived to 4. After this another process will go through lines 6 to 12 and 2 to 4 changing process arrived to 5. The last process will go through lines 6 to 12 and 2 to 4 changing process left to 0 and process arrived to 1. So we end up with a situation where all three proceeses are waiting on line 5 while process arrived is 1. This is a deadlock and will stop the program.

To rectify this we can add a "while (process_left != 0);" line after line 12 in the code. This makes sure one barrier has been processed before we enter the next.

4 Question 4

The correct option is (2). We wait till either the other process is one with it's critical section and has set flag(j) to false or the other process has set turn to us and is waiting in his own while loop. This ensures that entering the critical section twice simultaneously is impossible since for one of the processes the above condition has to hold. This is an application of Peterson's algorithm and can be used for parallelism if there is no inbuilt lock in the system. The use of two variables prevents deadlock.

5 Question 5

5.1 Part 1

$$Efficiency = \frac{T_s}{p * T_p}$$
$$Here,\ Eff = \frac{T_s}{T_s + p * T_{over}}$$

We can assume that for a fixed number of threads the overhead is constant with respect to the input size (as $T_{over} = T_s * p$ usually). Thus $T_{over} = c$ and serial time can be assumed to be $T_s = O(n^j)$ for some suitable k¿=1.

$$Eff = \frac{O(n^j)}{k + O(n^j)}$$

Thus it is clear that efficiency increases with increasing problem size.

5.2 Part 2

$$Efficiency = \frac{T_s}{p * T_p}$$
$$Here,\ Eff = \frac{n}{n + p * log(p)}$$

Scalability refers to whether or not we can icrease n and p simultaneously in a way so as to keep efficiency constant. Clearly from the above expression, making $O(n) = p * log(p)$ keeps efficiency constant hence this problem is **scalable**.

5.3 Part 3

A cost optimal version proceeds as follows - divide the array into p segments of equal size, add each segment separately on different processors, use a tree structure to add the p numbers together. Since communication is only needed when the p processors are adding their final numbers so we only need to spend it $log(p)$ times.

$$Thus,\, T_p = \frac{n}{p} + 21 * log_2(p)$$

$$S = \frac{n * p}{n + 21 * p\, log_2(p)}$$

$$E = \frac{n}{n + 21 * p\, log_2(p)}$$

$$Cost = n + 21 * p * log_2(p)$$

The Isoefficiency function is given as follows.

$$n = c * 21 * p * log_2(p)$$