COL 380
Homework 1 Solutions

Author : Aman Bhatia
Dated : 3 feb 2016

Pipelining

(A) There are 1000 floating point addition instruction(i=0 to i=999). As each Instruction takes 9ns to complete, 1000 operations will take 9000 ns if executed one by one i.e. in an unpipelined processor. Hence, the answer is **9000 ns or 9 micro seconds**.

(Neglecting for loop instructions and assuming that x[i] and y[i] both are fetched in Fetch Operand)

(B) We are assuming that we have 5 different pipeline stages for the EX part. Hence, the most expensive stage will be the OF stage taking 2 ns. Now, firstly there is no data dependency in the loop. So, instructions can execute independently. So, the first set of two instructions start executing. The next instruction has to wait for one cycle because both the OF units are busy for 2 cycles. So, the next set of two instructions will start from t=3. The pipeline diagram looks like,

1	2	3	4	5	6	7	8	9	10	11
OF		EX1	EX2	EX3	EX4	EX5	ST			
OF		EX1	EX2	EX3	EX4	EX5	ST			
		OF		EX1	EX2	EX3	EX4	EX5	ST	
		OF		EX1	EX2	EX3	EX4	EX5	ST	
				OF		EX1	EX2	EX3	EX4	EX5
				OF		EX1	EX2	EX3	EX4	EX5
						OF		EX1	EX2	EX3
						OF		EX1	EX2	EX3
								OF		EX1
								OF		EX1
										OF...
										OF...

So, now we have 9 stage pipeline so we should get a 9 fold increase in the execution time. But since the the most expensive operation takes 2 cycles, we should get (9/2=)4.5 fold increase. But, since it is a two way pipelined, we get further 2 fold increase. So the total increase will be (4.5 * 2 =) 9 fold.

Hence, the execution time will get decreased by 9 times i.e. (9000/9=)1000ns. Including, 7ns at the end, we get execution time to be **1007 ns.**

Caches

(A) Assumptions

(i) Assuming the cache is initially empty.
(ii) All memory accesses are from the cache, i.e. if there is cache miss, it is fetched to the cache first which takes 100 ns and then fetched from cache which takes 1 ns.
(iii) Execution time for instruction : 1 FLOP/ns

Let the size of the matrix be “n x n”. Hence, total no. of FLOPs will be $2n^2$(n^2 multiplications and n^2 additions).

- Now, the vector y will be fetched from the memory only once and then will be stored in the cache. This takes (n/4)*100 = 25n.

- Each row of the matrix needs to be fetched from the memory to the cache. This takes n*(n/4)*100 = $25n^2$.

Hence executing $2n^2$ FLOPs will take,	$25n$	{for fetching y to cache}
	$+\ 25n^2$	{for fetching x to cache}
	$+\ n^2$	{for fetching x from cache}
	$+\ n^2$	{for fetching y from cache}
	$+\ 2n^2$	{for execution of FLOPs}

For n = 4K, we get **68.95 MFLOPs/sec**.

Here, I have not considered anything about z because it will not affect the performance as much because its accesses will be less compared to those of x.

(B) Assumptions

(i) Assuming the cache is initially empty.
(ii) All memory accesses are from the cache, i.e. if there is cache miss, it is fetched to the cache first which takes 100 ns and then fetched from cache which takes 1 ns.
(iii) Execution time for instruction : 1 FLOP/ns
(iv) We will calculate performance for execution of the inner for loop i.e. result of multiplying one row of first matrix with the second matrix. This is because all the rows are same in terms of time of execution.

Let the size of the matrices be “n x n”. Hence, total no. of FLOPs will be $2n^2$(n^2 multiplications and n^2 additions).

- Now, the row of the first matrix will be fetched from the memory only once and then will be stored in the cache. This takes (n/4)*100 = 25n.

- Each column of the second matrix needs to be fetched from the memory to the cache. Since we are accessing column where as the matrix is stored in row major form, the cache line is of no use to us and we will have to fetch all the entries of the column individually. This takes

$n*n*100 = 100n^2$.

Hence executing $2n^2$ FLOPs will take,	$25n$	{for fetching row of 1st matrix to cache}
	$+ \ 100n^2$	{for fetching 2nd matrix to cache}
	$+ \ n^2$	{for fetching row from cache}
	$+ \ n^2$	{for fetching 2nd matrix from cache}
	$+ \ 2n^2$	{for execution of FLOPs}

For n = 4K, we get **19.23 MFLOPs/sec**.

Caches++

(A) There are two cases, which are as follows,

If T_0 gets to execute first, then the assignment x=1 will invalidate the value of x. Then when T_1 gets to execute then it will snoop into the cache and gets the updated value which is x=1. So, finally, **y will get the value of 1.**

If T_1 gets to execute first, then already the true value is stored and so **y will get the value of 0.**

(B) No, the answer is going to be the same independent of the cache coherence protocol. Cache coherence protocol does not affect the answers. The nondetermnism is due to the ordering of the thread execution.

(C) The problem is due to the different ordering of the threads. We can resolve te problem by doing thread scheduling and set constraints to either execute T_0 first or T_1 first so that determinism is maintained.

Performance

(A) Efficiency = $T_{serial} / p*T_{parallel}$

Putting $T_{serial} = p*(T_{parallel} - T_{overhead})$, we get efficiency = $(T_{parallel} - T_{overhead}) / T_{parallel}$

or efficieny = $1 - T_{overhead} / T_{parallel}$

If we increase the input problem size, keeping p fixed, then $T_{parallel}$ will surely increase where as $T_{overhead}$ will remain same. This results in decrease in the term $T_{overhead} / T_{parallel}$ and hence increase in the efficiency.

(B) Given T_{serial} = n and $T_{parallel}$ = (n/p) + $\log_2 p$

To check scalability, we increase p to kp and find factor x by which problem size needs to be increased so that Efficiency is unchanged.

Hence, E = $T_{serial} / p*T_{parallel} = n/(n + p*\log_2 p) = xn/(xn + pk*\log_2 pk)$

On solving we will get, $x = k + (k*\log_2 k / \log_2 p)$

Hence, we can keep the efficiency E fixed, by increasing the problem size with above factor while increasing p to kp. More acurately, we can conclude that **the program is weakly scalable.**

(C) We will divide n numbers among p processors. So now, each processor will add its own numbers and then finally in a binary tree manner, they will add their partial sums to get the final sum.

Hence, the time complexity for the local sums will be O(n/p) and for the total sum will be O(log p). However, at each level of the tree, 20 units of time will be taken for communication between the cores and 1 unit for addition. Hence, we get 21*lop p. So,

$T_{parallel} = n/p + 21*\log p$

$S = T_{serial} / T_{parallel} = n / (n/p + 21*\log p)$

$E = T_{serial} / p * T_{parallel} = n / (n + 21*p*\log p)$

$Cost = T_{parallel} = n + 21*p*\log p$

Isoefficiency function :-

$T_0 = p * T_{parallel} - T_{serial}$

$T_0 = 21*p*\log p$

Now, $W = (E/(E-1)) * T_0$

So, $W = (E/(E-1)) * (21 * p * \log p)$

------------------END OF FILE------------------