

Homework Set 1: COL380

Kartikeya Gupta, 2013CS10231

February 3, 2016

1.
 - In an un-pipelined processor, 1000 operations are processed and for each line, a total time of $2ns + 1ns + 1ns + 1ns + 1ns + 1ns + 2ns$ is required.
 \Rightarrow Total time needed = $1000 * 9ns$
 $\Rightarrow t = 9000ns$
 - In the given timings, the first and last stages take $2ns$ the amount of time while the other states take $1ns$ time. Because of this the time spent per instruction is $2ns$.
 The number of stages in the pipeline are 7.
 As there are 2 pipelined processors, the number of lines executed by each will be half of earlier = 500.
 \Rightarrow Time taken for processing 500 lines by a processor = $500 * 2ns = 1000ns$.
 But we have not taken into account the time which the last instruction will spend before leaving the pipeline. This is going to be $1ns + 1ns + 1ns + 1ns + 1ns + 2ns = 7ns$.
 \Rightarrow Total time taken = $1000ns + 7ns = 1007ns$.

Timeline Diagram

The value in a cell represents the operation number which is being executed. A value of – represents that the particular block will idle.

Table 1: Timeline Diagram

StagesCycles	0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8	8-9	9-10	10-11
1 Fetch Operands	I		II		III		IV		V		VI
2 Compare Exponents			I	-	II	-	III	-	IV	-	V
3 Normalize				I	-	II	-	III	-	IV	-
4 Add					I	-	II	-	III	-	IV
5 Normalize Result						I	-	II	-	III	-
6 Round Result							I	-	II	-	III
7 Store Result								I		II	

2.
 - For the peak operation, the entire y will be present in the cache and blocks of x will keep entering and getting evicted by the cache. The present z which is being

needed will also be a member of the cache.

Lets now consider the memory access times as follows

- (a) Time needed to get the entire y in cache from DRAM:
 K amount of cache lines to be retrieved = $K * 100ns$ time.
- (b) Time needed to get the entire z in cache once from DRAM:
 K amount of cache lines to be retrieved = $K * 100ns$ time.
- (c) Time needed to get the entire x in cache repeatedly from DRAM:
 $16K^2/4$ number of cache lines to be retrieved = $4K^2 * 100ns$
- (d) Time needed for retrieving data from the caches for the arithmetic operations:
 $16K^2$ operations taking place and for each of these 3 elements have to be accessed from the cache = $48K^2ns$

- (e) Time needed for processing:
 $2 * 16K^2$ operations require a time of = $32K^2ns$.

\Rightarrow Total time for this = $480K^2 + 200Kns$

Total number of instructions taking place = $32 * K^2$.

\Rightarrow Number of operations per second =

$$\frac{32K^2 * 10^9}{480K^2 + 200K}$$

Approximating K to 1000, \Rightarrow

$$\frac{32 * 10^6 * 10^3}{480 * 10^6 + 200 * 10^3} Mflops$$

\Rightarrow

$$66.68 Mflops$$

- Consider the following code for multiplying the matrices

```

1 for (int i=0; i< SIZE ; i++)
2     for (int j=0 ; j< SIZE ; j++)
3         for (int k=0; k< SIZE; k++)
4             C[i][j]+=A[i][k]*B[k][j];

```

For multiplying 2 dense matrices in given, we have to perform $(4K)^3$ mathematical operations

When the matrices are stored in row major form, for matrix A , we need to keep a given row i of the matrix A in the cache along with the memory in C where the result is to be stored. The elements of matrix B are to be fetched column

wise but if we wish to access a particular element, we will get 3 other elements which are of no use at that time. Hence for a given i and j , the entire column from matrix C needs to be accessed from the DRAM.

Let us calculate the memory access times as follows:

- (a) Time spent in getting Matrix A in the cache from DRAM:
 $4K^2$ number of cache lines to be retrieved = $4K^2 * 100ns$ time.
- (b) Time spent in getting Matrix C in the cache from DRAM:
 $4K^2$ number of cache lines to be retrieved = $4K^2 * 100ns$ time.
- (c) Time spent in getting Matrix B in the cache from DRAM:
 $(4K)^3$ number of cache lines to be retrieved = $64K^3 * 100ns$ time.
- (d) Time needed for retrieving data from caches for arithmetic operations:
 $3 * (4K)^3$ values need to be accessed from the cache = $192K^3$ ns time.
- (e) Time needed for the arithmetic operations to take place:
 $2 * (4K)^3$ number of arithmetic operations take place = $128K^3$ ns time.

\Rightarrow Total time for this = $6720K^3 + 800K^2$ ns.

Total number of instructions taking place = $128K^3$

\Rightarrow Number of operations per second =

$$\frac{128 * K^3 * 10^9}{6720 * K^3 + 800 * K^2} Flops$$

Approximating K to 1000 \Rightarrow

$$\frac{128 * 10^9 * 10^3}{6720 * 10^9 + 800 * 10^6} MFlops$$

\Rightarrow

$$19.045 MFlops$$

3.
 - As the 2 threads are running concurrently, either of the 2 instructions can execute before the other.

Case 1 : T_0 executed before T_1 .

In this case, the value of x gets updated to 1 in the cache corresponding to the thread T_0 . By snooping protocol, the cache of T_1 detects the change and hence when T_1 executes the command, the value of y is set to 1.

Case2 : T_1 executed before T_0 .

In this case the value of y is set as 0 as that is the value which is present in the cache.

- As the 2 threads are running concurrently, either of the 2 instructions can execute before the other.

Case 1 : T_0 executed before T_1 .

In this case, the value of x gets updated to 1 in the cache corresponding to the thread T_0 . By directory based cache protocol, When T_1 executes the command, the value of y is set to 1.

Case2 : T_1 executed before T_0 .

In this case the value of y is set as 0 as that is the value which is present in the cache.

- There is no problem based on cache protocols and coherence in this situation. The problem here is because of lack of synchronisation amongst the threads.

4. • Speedup:

$$\begin{aligned} s &= \frac{T_{serial}}{T_{Parallel}} \\ &= \frac{T_{Serial}}{T_{Overhead} + \frac{T_{Serial}}{p}} \end{aligned}$$

Efficiency:

$$\begin{aligned} e &= \frac{s}{p} \\ &= \frac{T_{Serial}}{p * T_{Overhead} + T_{Serial}} \\ &= \frac{1}{1 + p * \frac{T_{Overhead}}{T_{Serial}}} \end{aligned}$$

On increasing the problem size, the rate of increase of $T_{Overhead}$ is lower than that of T_{Serial} hence the value of the denominator term in efficiency keeps decreasing as the problem size increases.

\implies That the efficiency of a program increases with increase in program size.

- To comment on the scalability of the program, we need to check if on increasing n , the value of efficiency can be kept the same by increasing p .

$$\begin{aligned} e &= \frac{s}{p} \\ &= \frac{T_{serial}}{p * T_{Parallal} \frac{n}{n}} \\ &= \frac{1}{n + p * \log p} \\ &= \frac{1}{1 + \frac{p * \log p}{n}} \end{aligned}$$

Now if we increase n we can increase p as well so that $\frac{p * \log p}{n}$ remains the same and hence the value of efficiency remains the same.

The property which needs to exist hence is:

$$\frac{p * \log p}{n} = c$$

$$\implies p * \log p = c * n$$

- For cost optimal version of prefix sums, we will compute sum of n/p numbers on p different cores in parallel. Then we will add the results in a binary tree fashion such that the tree is of height of $\log p$.

(a) Time needed for computing sum of n/p numbers in parallel : $n/p - 1$.

(b) Time needed for joining sum of 2 precomputed sums is : $20 + 1$

(c) Total time needed for joining results of the n/p computed sums by the tree:
 $21 * \log p$

$$\implies T_{Parallel} = n/p + 21 * \log p - 1$$

If this was executed sequentially, then the time needed is $n - 1$

$$T_{Sequential} = n - 1$$

$$T_{Parallel} = n/p - 1 + 21 * \log p$$

$$S = \frac{T_{Serial}}{T_{Parallel}}$$

$$= \frac{n - 1}{n/p + 21 * \log p - 1}$$

$$Efficiency = \frac{S}{p}$$

$$= \frac{n - 1}{n - 1 + 21 * p * \log p}$$

$$= \frac{1}{1 + \frac{21 * p * \log p}{n - 1}}$$

$$Cost = p * T_{Parallel}$$

$$= n - 1 + p * \log p$$

$$Isoefficiencyfunction \Rightarrow 1 + \frac{21 * p * \log p}{n - 1} = c$$

$$\Rightarrow \frac{21 * p * \log p}{n - 1} = c$$

$$\Rightarrow 21 * p * \log p = c * (n - 1)$$

$$\Rightarrow IsoEfficiencyfunction = \frac{21 * p * \log p}{n - 1}$$