Introduction to Parallel & Distributed Programming
COL380 - Assignment 1

Ankesh Gupta
2015CS10435

Question 1

Common Assumptions:

1. Processor is able to fetch two words from memory in single fetch.
2. All stages mentioned in question are part of processor, one after the other.
3. Loop variables are cached and comparison-adder for loop iterations is being maintained using separate hardware, taking negligible time.

In an unpipelined setting, things are simpler. Each for-loop manipulation of z goes completely through the processor. The clock is unpipelined setting is 1 common clock, with 1 clock cycle period=$(2+1+1+1+1+1+2)ns = 9ns$. The loop executes 1000 times and thus,

$$\boxed{T_{unpipelined} = (1000 * 9)ns = 9000ns}$$

Extra Assumptions:

1. There are 7 stages in pipeline. They are in order as mentioned in question.
2. All stages of pipeline share the same clock. The clock period is dictated by the slowest step, which is $2ns$.
3. Above assumption also ensures no stalls in our execution pipeline.
4. 2-way Pipelined Processor is interpreted as 2 execution pipelines available parallely.

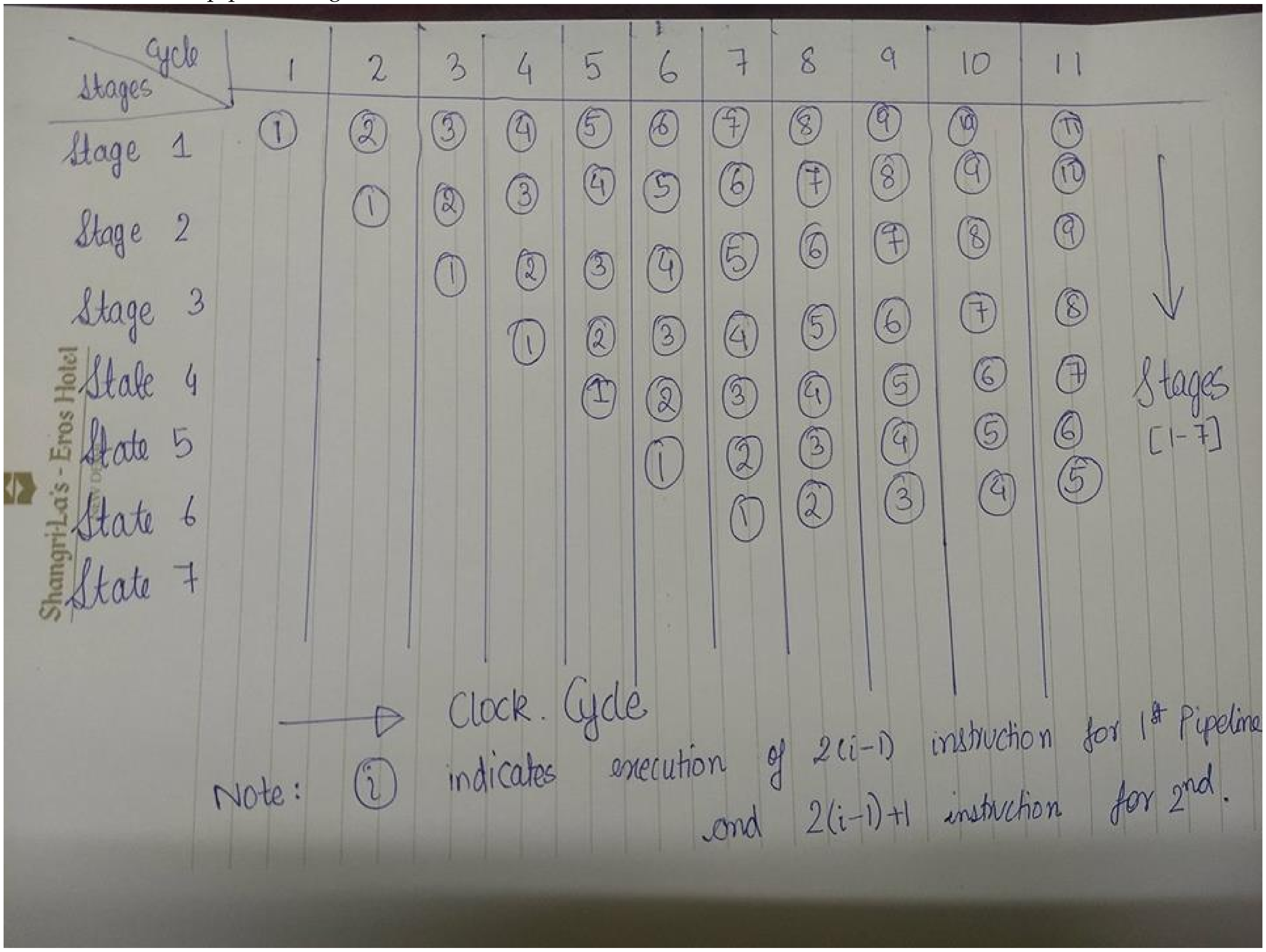
In pipelined setting, $clockcycle = 2ns$, but each instruction travels 7 stages, making in a way, $14ns$ per instruction. But because pipeline architechture exploits ***idealism of stages***, in an amortized setting, we can obtain 1 instruction per cycle. Since each of the $z[i]$ computations are ***independent*** of each other, we can feed 2 instruction for adjacent computations in 2 pipelines available. Hence, in essence, each pipeline executes 500 such instructions. One of them, for odd i's and the other, for even i's.

$$T_{pipelined} = (6 * 2)ns(filling\_pipeline) + (500 * 2)ns$$
$$\boxed{T_{pipelined} = (1012)ns}$$

Note: The assumption of no bubble in extra assumption section may be relaxed. If clock cycle is reduced to= $1ns$, we realise that each fetch and store operation will lead to pipeline stalls. In such a scenario, pipeline filling takes= $7ns$ and after, each instruction execution is completed in $2ns$. Hence, total time in this case:

$$T_{modified\_pipelined} = (5 * 1 + 2)ns(filling\_pipeline) + (500 * 2)ns$$
$$\boxed{T_{modified\_pipelined} = (1007)ns}$$

The pipeline stages are shown for first calculations, which had no stalls.



Stages \ Cycle	1	2	3	4	5	6	7	8	9	10	11
Stage 1	1	2	3	4	5	6	7	8	9	10	11
Stage 2		1	2	3	4	5	6	7	8	9	10
Stage 3			1	2	3	4	5	6	7	8	9
Stale 4				1	2	3	4	5	6	7	8
State 5					1	2	3	4	5	6	7
State 6						1	2	3	4	5	6
State 7							1	2	3	4	5

Stages [1-7]
Clock. Cycle

Note: (i) indicates execution of 2(i-1) instruction for 1st Pipeline and 2(i-1)+1 instruction for 2nd.

Question 2

Assumptions:

1. Since not clear from question, it is assumed that array is $integerarray(size1word)$ of size $4000X4000$.
2. Prefetching's are ignored to keep calculations simpler.
3. Data fetched from DRAM is first loaded in cache, and then fetched from cache in subsequent cycles.
4. Pipeline type fetching from DRAM is ignored.
5. Again, for loop timings are ignored considering separate hardware taking negligible time.

The problem will be simpler to deal with if we analyze each of the array's independently.

For z array, a memory access need to be made *once* every 4 times loop j executes. Rest 3 times,

cache comes in handy.
For x array, a memory access need to be made *once* every 4 times loop k executes. Rest 3 times fetch is made from cache.
For y array, a memory access need to be made *once* for every iteration of loop k. Since the access's are column major wise, no cache locality could be exploited.

Clearly, if we look at k array, then in every 4 iterations of the loop, memory access is made $(4+1) = 5$ times for $y(4\_times)$ array and $x(once)$ array respectively. Cache access is done $(4+4+4) = 12$ times, 4 times for x,y and z respectively. Hence, for doing $4*2 = 8$ floating point operations(FLOP), $512ns$ are required. Hence, in essence, we perform:

$$\begin{aligned} Performance &= (\frac{8}{512})GFLOPS \\ &= (0.0156)GFLOPS \\ &= (15.6)MFLOPS \end{aligned}$$

Note: z DRAM access are kept out of calculation as DRAM access are made once in every ((4*SIZE)*2) floating point operations, to which it has negligible contribution

Question 3

Option **2** is correct. The following sequence of events will lead to a **deadlock** situation between the threads. Execute and Loops at below are used with respect to Statement No. mentioned in the question.

1. Thread 1 executes 1,2,3,4,5. Loops at 5. $process\_arrived = 1$
2. Thread 2 executes 1,2,3,4,5. Loops at 5. $process\_arrived = 2$
3. Thread 3 executes 1,2,3,4,5 ($process\_arrived = 3$,breaks),6,7,8,12,1 *(successive invocation)*,2,3,4,5. Loops at 5. $process\_arrived = 4, process\_left = 1$
4. No other thread ever breaks free of the loop as *processed_arrived* will never be reset again

The sequence of operations above leads to **deadlock**, where no thread is able to make any progress. This situation arise because 2 barrier in invocations are used in immediate succession(noted in point 3)

The following code is one of the ways of resolving the issue pointed out above.

```
void barrier ( void ) {
  L( S ) ;
  process_arrived++;
  UL( S ) ;
  while ( process_arrived!=3) ;
  L( S ) ;
  process_left++;
  UL( S ) ;
  if(process_left==3){
    process_arrived=0;
    process_left=0;
  }
  while(process_left!=0);
}
```

The above code ensures that all the 3 threads executing the barrier leaves the barrier simultaneously. The flaw in previous snippet was that it synchronised threads momentuously, but their exiting the barrier was independent. Now, the barrier will ensure that all variables have been reset, and even the last thread in opertion pipeline is ready to depart.

Question 4

Option (b) is correct.

```
  ****Thread i****                          ****Thread j****
while (true) {                            while (true) {
1. flag[i]=true;                          flag[j]=true;
2. turn=j;                                turn=i;
3. while(flag[j]==true && turn==j);       while(flag[i]==true && turn==i);
   /    enter critical section            /    enter critical section
   perform actions                        perform actions
   exit */                                exit */
4. flag[i]=false ;                        flag[j]=false ;
   }                                      }
```

Justification of Exclusion: Suppost for sake of contradiction, we assume that both the threads execute critical section simultaneously. WLOG, we assume that thread i executed statement 2 first. So, for thread j to execute critical section, flag[i] must have been false in while loop **(..a)**. If we construct a **happen-before** graph, we realise that Thread i statement 2 must happen before Thread j statement 2(assumption), which must happen before Thread j statement 3(sequentiality), which must happen before Thread i statement 1**(from (..a))**.

Hence there is a *happen-before loop, which leads to contradiction*. Notice here we assumed the an order of execution of Statement 2 WLOG, hence this completes the proof.

Justification of Liveliness: Its clear that if only 1 thread is invoking the function, the other threads flag remains false, and thread works uninterrupted. Interesting is when both threads want to access the critical section. Again, the while loop construct ensures that none of the threads executes the critical section in succession, keeping the other thread waiting. Each resetting of *turn* variable is an opportunity for the waiting thread to complete its request of critical section.

Question 5

(a) Lets start by writing the efficiency equation for given system, and doing algebraic manipulations.

$$E = \frac{T_{serial}}{p * T_{parallel}}$$

Substituting from the question, $T_{parallel}$, we get:

$$\begin{aligned} E &= \frac{T_{serial}}{T_{serial} + p * T_{overhead}} \\ &= \frac{1}{1 + \frac{p * T_{overhead}}{T_{serial}}} \end{aligned}$$

We know that T_{serial} is same as W(input size), the workload, which gives us:

$$E = \frac{1}{1 + \frac{p * T_{overhead}}{W}}$$

Assuming that $T_{overhead}$ is a **sublinear** function of Workload(W), the growth in workload dominates the growth in overhead, which decreases the denominator, in turn increasing efficiency. Asymtotically we can write:

$$E = \frac{1}{1 + \frac{p * T_{overhead}}{W}} \approx 1$$

Hence, as input size grows, $Efficiency$ increases, and aymtotically reaches 1. The main assumption is that:

$$W = \Omega(T_{overhead})$$

(b) **Scalability** is defined as the ability of parallel system to keep efficiency, when processing units and workload of the system changes.

Lets analyze the given parallel construct and derive iso-efficiency function.

$$\begin{aligned} E &= \frac{T_{serial}}{p * T_{parallel}} \\ &= \frac{n}{n + p * \log_2 p} \end{aligned}$$

Moving terms, and writing n as function of $p(processingunits)$, we get:

$$n = \frac{E}{1 - E} * p \log_2 p$$

Let initial processing units be p_1, and later it changed to $p_2 = k * p_1$, corresponding n_1, n_2 are:

$$\begin{aligned} n_1 &= \frac{E}{1 - E} * p_1 \log_2 p_1 \\ n_2 &= \frac{E}{1 - E} * k * p_1 \log_2 (k * p_1) \end{aligned}$$
$$\boxed{\frac{n_2}{n_1} = k * (1 + \frac{log_2 k}{log_2 p_1})}$$

Although the system is scalable(because it are able to maintain *isoefficiency*), it is **weakly scalable**. This is because increase in *workload*(n) required to attain iso-efficiency is $super - linear$ with respect to processing units, as visible from the last equation. Hence the system is ***weakly scalable***.

(c) The cost optimal version of parallel-sum algorithm for n numbers on p processing units does the following:

(i) Divide the numbers *equally* and *exclusively* amongst the p cores. Each core get $\frac{n}{p}$ numbers.

(ii) Sum all the numbers each core receives and store the sum locally. This takes $O(\frac{n}{p})$ time.

(iii) Merge the local sums from each of the cores into a global sum in *divide and conquer* style. This takes $O(log_2 p)$ time.

Examining the above algorithm, running time of above algorithm is $T_{parallel} = O(\frac{n}{p} + \log_2 p)$. The best serial algorithm sums up the numbers in $T_{serial} = O(n)$ time. Analyzing the cost:

$$\begin{aligned} C_s &= T_{serial} = n \\ C_p &= p * T_{parallel} = n + p * \log_2 p \\ \frac{C_p}{C_s} &= 1 + \frac{p * \log_2 p}{n} \end{aligned}$$

As long as $n = \Omega(p * \log_2 p)$, the above parallel construct is cost optimal, as then asymtotically:

$$\boxed{C_p = \Theta(C_s)}$$

Deriving expressions for terms asked in question, $T_{parallel}$ has already been written above in runtime analysis. Substituting the values for summing,

$$\begin{aligned} T_{parallel} &= \frac{n}{p} + \log_2 p \\ &= \frac{n}{p} * 1 + \log_2 p * (20+1) (Substituting) \\ \end{aligned}$$

$$\boxed{T_{parallel} = \frac{n}{p} + \log_2 p * (21)}$$

For **speedup(S)**,

$$\begin{aligned} S &= \frac{T_{serial}}{T_{parallel}} \\ &= \frac{n}{\frac{n}{p} + log_2 p} \end{aligned}$$

$$\boxed{S = \frac{p}{1 + \frac{p * log_2 p * 21}{n}}} (Substituting)$$

For **efficiency(E)**,

$$\begin{aligned} E &= \frac{T_{serial}}{p * T_{parallel}} \\ &= \frac{n}{p * (\frac{n}{p} + log_2 p)} \\ &= \frac{n}{n + p * \log_2 p} \\ &= \frac{n}{n + p * \log_2 p * 21} (Substituting) \end{aligned}$$

$$\boxed{E = \frac{1}{1 + \frac{p * \log_2 p * 21}{n}}}$$

Equations for **Cost** had been derived while prooving optimality. What remains is substitution of time.

$$
\begin{aligned}
C_s &= T_{serial} = n \\
C_p &= p * T_{parallel} = n + p * \log_2 p * 21 \\
&\boxed{\frac{C_p}{C_s} = 1 + \frac{p * \log_2 p * 21}{n}}
\end{aligned}
$$

Iso-efficiency can be derived from Efficiency equation, by algebraic manipulations.

$$
\begin{aligned}
E &= \frac{n}{p * (\frac{n}{p} + log_2 p * 21)} \\
n * (1 - E) &= E * p * log_2 p * 21 \\
&\boxed{n = \frac{E}{1 - E} * p * log_2 p * 21}
\end{aligned}
$$