Homework-1

Sriamoghavarsha Beerangi Srinivasa

September 11, 2024

Problem 1

For each of the following, state whether it is a safety or liveness property, or both, and identify the bad or good thing of interest.

Solution:

a) **No process should receive a message, unless the message was indeed sent.**

- **Safety property** - This ensures that something bad (receiving a message that was not sent) never happens.

b) **If a process sends a message to a destination process, then the destination process should eventually receive that message.**

- **Liveness property** - This ensures that something good (the message being received) eventually happens.

c) **Messages exchanged between two processes are neither lost nor duplicated and re-received in the order in which they were sent.**

- **Safety and Liveness properties** - Safety ensures messages are not lost or duplicated, while liveness ensures they are delivered in the correct order.

d) **Two processes with critical sections c_0 and c_1 may not be in their critical sections at the same time.**

- **Safety property** - This guarantees that a bad situation (both processes in the critical section simultaneously) never happens.

e) **If an interrupt occurs, then a message is printed within one second.**

- **Safety and Liveness properties** - Liveness ensures that the message will be printed eventually, and safety ensures that it is printed within the specified time limit.

f) **If an interrupt occurs, then a message is printed.**

- **Liveness property** - This guarantees that the message will eventually be printed after the interrupt.

g) **If a process invokes an operation and never crashes, then the operation eventually completes.**

- **Safety and Liveness properties** - Liveness ensures that the operation will eventually complete if the process doesn't crash. Safety ensures that the system will never leave the operation incomplete indefinitely if the process remains active, avoiding a bad state.

h) **Cars entering a roundabout yield to cars already in the roundabout.**

- **Safety property** - This guarantees that something bad (a collision) never happens.

Problem 2

Solution:

(a) Initial state of the switch is known (off):

We can designate one prisoner as the **counter**. The counter will keep track of how many prisoners have visited the switch room by using the state of the switch. The strategy works as follows:

a) Whenever a prisoner who is not the counter enters the room and sees the switch is **off**, they turn it **on**. However, they can only do this once during the entire process.

b) If a prisoner enters the room and finds the switch is already **on**, they leave it unchanged.

c) Every time the **counter** enters the room and finds the switch is **on**, they turn it **off** and increment their count by 1.

d) Once the **counter** has turned the switch off exactly $(P - 1)$ times (where P is the total number of prisoners), they know that all the other prisoners have visited the room at least once and can safely declare.

This strategy will work because each non-counter prisoner turns the switch on only once, so the counter will count exactly $(P - 1)$ **on** states, which corresponds to all the other prisoners having visited the room at least once.

(b) Initial state of the switch is unknown:

In this case, we modify the strategy slightly to handle the uncertainty about the initial state of the switch:

a) We again designate one prisoner as the **counter**.

b) The first time any prisoner (other than the counter) enters the room, if they see the switch is **on**, they do nothing (this avoids confusion about whether the switch was initially on). If they see the switch is **off**, they turn it **on**. Like before, they can only turn the switch on once.

c) The **counter** follows the same rule as in part (a): whenever they find the switch is **on**, they turn it **off** and increment their count by 1.

d) Once the **counter** has counted $P - 1$ switches from on to off, they can declare that all prisoners have visited the room.

This approach works because prisoners ignore the first time they find the switch on if they don't know the initial state, preventing incorrect counting based on the switch's starting condition.

Problem 3

Solution:

Synchronization Mechanism:

Each person has a can on their window sill, connected by a string to the other person's house. The state of the can (either **UP** or **DOWN**) helps Alice and Bob determine whether they should proceed with their task (releasing pets or putting food) or wait for the other person to act.

Protocol for Alice (Consumer):

a) **Checking the Can:** Alice first checks the state of her can to determine whether it is **UP** or **DOWN**.

b) If the Can is **UP**, Alice waits until the can is **DOWN**, which signals that Bob has placed food for the pet.

c) If the Can is **DOWN**, Alice releases the pet and then resets her own can by turning it **UP**. Alice then knocks down Bob's can indicating Bob to put food again.

Protocol for Bob (Producer):

a) **Checking the Can:** Bob first checks the state of his can to determine whether it is **UP** or **DOWN**.

b) If the Can is **UP**, Bob waits until the can is **DOWN**, which signals that Alice has released the pet.

c) If the Can is **DOWN**, Bob places food for the pet and then resets his own can by turning it **UP**. Bob then knocks down Alice's can indicating Alice to release the pet again.

The synchronization is achieved using the state of the cans (**UP** or **DOWN**), ensuring that:

- Bob does not put food until Alice has captured the pet,
- Alice does not release the pet until Bob has put food.

The alternating states of the cans help prevent both from acting at the same time, ensuring that no race conditions or miscoordination occur.

Problem 4

Solution:

The Amdahl's Law is given by:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

where:

- $S(p)$ is the speedup with p processors,
- f is the fraction of the computation that is parallelizable,
- p is the number of processors.

We are given:

- $S(72) = 80$ (the desired speedup),
- $p = 72$ (the number of processors).

We need to determine the fraction of the computation that can be sequential, which is $(1-f)$.

Rearranging Amdahl's Law, we get:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

Substituting the known values of $S(72) = 80$ and $p = 72$:

$$80 = \frac{1}{(1-f) + \frac{f}{72}}$$

Now, we solve for f:

$$\begin{aligned}
\frac{1}{80} &= (1-f) + \frac{f}{72} \\
0.0125 &= (1-f) + \frac{f}{72} \\
0.0125 \times 72 &= 72(1-f) + f \\
0.9 &= 72 - 71f \\
f &= \frac{71.1}{71} = 1.0014
\end{aligned}$$

Thus, 100.14% of the computation should be parallelized. This implies that more than 100% of the computation would need to be parallelized, which is impossible. Therefore, the desired speedup of 80 with 72 processors cannot be achieved.

Problem 5

Solution:

The inner loop iterates over the index j, modifying $a[j][i]$ based on values from neighboring columns, $a[j][i+1]$ and $a[j][i-1]$. Since there is no dependency between different iterations of j, we can parallelize the inner loop.

- $a[j][i]$ depends on $a[j][i+1]$ and $a[j][i-1]$, but these values are independent for different values of j.
- Therefore, the inner loop is safe to parallelize, as no iteration of j depends on another iteration.

The outer loop iterates over the index i, modifying $a[j][i]$ based on the previous and next columns, $a[j][i+1]$ and $a[j][i-1]$.

- There is a dependency between iterations of i because the value of $a[j][i]$ depends on $a[j][i-1]$.
- This means the calculation for column i depends on the result from column $i-1$.
- As a result, the outer loop cannot be parallelized because of this data dependency.

Let's consider an example with $n = 3$.

For $i = 1$, we update the elements of the matrix as follows:

- $a[0][1] += a[0][2] - a[0][0]$
- $a[1][1] += a[1][2] - a[1][0]$
- $a[2][1] += a[2][2] - a[2][0]$

For $i = 1$, we update the elements of the matrix as follows:

- $a[0][2] += a[0][3] - a[0][1]$
- $a[1][2] += a[1][3] - a[1][1]$
- $a[2][2] += a[2][3] - a[2][1]$

Since each row j operates independently from other rows, the inner loop can be parallelized.

For $i = 2$, the update depends on the values of $a[j][1]$, which were modified in the previous iteration. This dependency means the outer loop cannot be parallelized.

Problem 6

Solution:

Case 1: Uni-processor

The uni-processor can execute 8Z instructions per second, so the execution time for x instructions is:

$$T_{uni} = \frac{x}{8}$$

Case 2: Multiprocessor

The multiprocessor has 16 cores, each capable of executing 1 zillion instructions per second. The execution time for x instructions including parallelizable portion f is:

$$T_{multi} = x \times \left((1 - f) + \frac{f}{16} \right)$$

Calculating the Break-even Point

The break-even point is the point where the performance (execution time) of the uni-processor is equal to the performance of the multiprocessor i.e. $\textit{Speedup} = 1$. We find the break-even point by equating the execution times equal to each other :

$$\begin{aligned}
\frac{x}{8} &= x \times \left((1 - f) + \frac{f}{16} \right) \\
\frac{1}{8} &= (1 - f) + \frac{f}{16} \\
\frac{1}{8} &= 1 - f + \frac{f}{16} \\
0.125 &= 1 - f + \frac{f}{16} \\
-0.875 &= -f + \frac{f}{16} \\
-14 &= -15f \\
f &= \frac{14}{15} = 0.933
\end{aligned}$$

The break-even point occurs when approximately **93.3%** of the program is parallelizable. Therefore, if more than 93.3% of the application can be parallelized, it is beneficial to use the 16-core multiprocessor. If less than 93.3% is parallelizable, the uni-processor is the better option.

Problem 7

Solution:

Original Amdahl's Law:

Amdahl's Law provides the speedup of a program based on:

- f: The fraction of the program that can be parallelized,
- $(1-f)$: The fraction of the program that is sequential and cannot be parallelized,
- p: The total number of processors.

The original formula for speedup $S(p)$ is:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

Modifying Amdahl's Law for $F(i,p)$:

Here, we modify the law to account for the fact that the number of usable processors varies over time, as captured by the function $F(i,p)$. This function tells us the fraction of time exactly i processors are usable out of p total processors.

When i processors are in use, the program runs i times faster. Therefore, the contribution to the speedup from each i processors is:

$$\frac{1}{i}$$

We also know that this configuration of i processors is usable for $F(i,p)$ fraction of the time. The contribution to the overall speedup from using i processors is:

$$F(i,p) \times \frac{1}{i}$$

Since different numbers of processors can be used at different times, we sum up the contributions from all configurations $i = 1$ to p:

$$\sum_{i=1}^{p} F(i,p) \times \frac{1}{i}$$

The final formula for the speedup $S(p)$, considering the varying number of processors and their respective usage fractions, is:

$$S(p) = \frac{1}{\sum_{i=1}^{p} \frac{F(i,p)}{i}}$$

Explanation:

- The term $\frac{1}{i}$ represents the speedup achieved when exactly i processors are in use.
- $F(i, p)$ weights this speedup by the fraction of time that configuration is available.
- By summing over all i from 1 to p, we account for all possible processor configurations.

This formula includes the sequential portion of the program implicitly within the case where $i = 1$, thus covering both the sequential and parallel portions of the program within the same summation.

Problem 8

Solution:

a) Let:

- $f = 0.7$, the fraction of the program dedicated to sorting,
- $(1 - f) = 0.3$, the sequential portion,
- $S(p) = 4$, the desired speedup,
- p be the number of cores.

Using Amdahl's Law, we have:

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

Substituting the known values:

$$4 = \frac{1}{0.3 + \frac{0.7}{p}}$$

$$4 \times \left(0.3 + \frac{0.7}{p}\right) = 1$$

$$1.2 + \frac{2.8}{p} = 1$$

Solving for p:

$$\frac{2.8}{p} = -0.2 \quad \Rightarrow \quad p = \frac{2.8}{-0.2} = -14$$

The result, $p = -14$, shows that achieving a $4x$ speedup with 70% of the program parallelized is impossible because the number of cores cannot be negative. Hence, it is not possible to achieve a 4x speedup in this scenario.

b) **Sequential Merge Sort:**

Sequential merge sort uses the divide and conquer approach to sort an array. The array is divided into halves until the array length is 1, which is always sorted. After this step, all these sub-arrays are merged back. Since the sequential algorithm has to traverse an array of size n in $\log n$ time, the time complexity is asymptotically:

$$T_1 = O(n \log n)$$

Parallel Merge Sort:

In parallel merge sort, the array is divided into halves just like in sequential merge sort, but it invokes the ForkJoinPool framework to handle sorting and merging each sub-array in parallel. Due to this approach, the algorithm needs to traverse the entire n elements. Asymptotically, it takes $O(n)$ time.

The time complexity for the parallel version is given by:

$$T_p = O\left(\frac{T_1}{p} + T_\infty\right)$$

where $T_1 = O(n \log n)$ is the sequential time and p is the number of processors.

Speedup Calculation:

According to Amdahl's law, the speedup is given by:

$$\text{Speedup} = \frac{T_1}{T_p}$$

Substituting the values, we get:

$$\text{Speedup} = \frac{n \log n}{\frac{n \log n}{p} + n}$$

Simplifying:

$$\text{Speedup} = \frac{\log n}{\frac{\log n}{p} + 1}$$

Finally, the upper bound for the speedup of the new parallel merge sort algorithm over the sequential merge sort is:

$$\boxed{\text{Speedup} = \frac{p \log n}{p + \log n}}$$