Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Solution 4: Basic Parallel Programming Concepts
Spring Semester 2017

Assigned on: **14.03.2017** Due by: **20.03.2017**

Task 1 – Amdahl's and Gustafson's Law

Assuming a program consists of 50% non-parallelizable code.

a) Compute the speed-up when using 2 and 4 processors according to Amdahl's law.

Answer: Amdahl's law says: $S_p \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{p}} = \frac{1}{f + \frac{1-f}{p}}$

Therefore we have

$S_2 \leq \frac{1}{\frac{1}{2}+\frac{1}{4}} = \frac{4}{3} \approx 1.33$

and

$S_4 \leq \frac{1}{\frac{1}{2}+\frac{1}{8}} = \frac{8}{5} =\approx 1.6$

b) Now assume that the parallel work per processor is fixed. Compute the speed-up when using 2 and 4 processors according to Gustafson's law.

Answer: Gustafson's Law says: $S_p = p - f(p-1)$

So we have

$S_2 = 2 - \frac{1}{2}(2-1) = \frac{3}{2} \approx 1.5$

and

$S_4 = 4 - \frac{1}{2}(4-1) = \frac{5}{2} \approx 2.5$

c) Explain why both speed-up results are different.

Answer: Amdahl's law sees the percentage of non-parallelizable code as a fixed limit for the speedup. So even if we had an infinite amount of processors, according to Amdahl's law, the speedup would never be greater than 2.
On the other hand Gustafson's law assumes that the parallel part of the program increases with the problem size and the sequential part stays fixed.

~~A more formal explanation:[*]~~

~~The fundamental assumption of Amdahl's law is that the sequential part of the executable work W is given by a fixed ratio f of the **overal** work W provided: $W_s = W \cdot f$, $W_p = W \cdot (1 - f)$.~~

~~Assuming it would take time T to process W on a single core, we compute the time required to process work W_p with p processors as $T_p = T \cdot f + T \cdot (1 - f)/p$. Thus $T_1/T_p = \frac{1}{f+(1-f)/p}$~~

~~The fundamental assumption of Gustafson's law is that the sequential part of the executable work W is given by a fixed ratio f of the work W **before the work size is increased** to $W_p = W \cdot p$ when p processors are available: $W_s = W \cdot f$, $W_p = W \cdot p \cdot (1 - f)$.~~

~~Assuming it would take time T to process W on a single core, we compute the time required to process work W_p with p processors as $T_p = T \cdot f + T \cdot p \cdot (1 - f)/p = T$ while with only one processor we get $T_1 = T \cdot f + T \cdot (1 - f) \cdot p$, thus $T_1/T_p = f + (1 - f) \cdot p$.~~

Task 2 – Pipelining

Bob, Mary, John and Alice share a flat. In this flat they share a washing machine, a dryer and a ironing board. The washing machine takes 50 minutes for one wash cycle. The dryer takes 90 minutes. Everyone of them takes roughly 15 minutes to iron their laundry.

a) Assuming they would do their laundry in strictly sequential order (one person starts only after the other finished ironing), calculate how long would it take to finish the laundry?

Answer: They would take $(50 + 90 + 15) * 4 = 620$ minutes.

b) Are there any better options? If yes, can you describe them and calculate the improved laundry time?

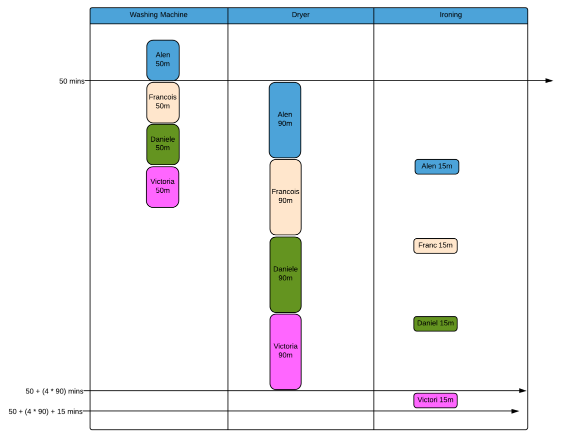


Figure 1: Perfect pipeline

[*]For the following arguments, we assume that, homogeneously over all processors, the time T [e.g. in seconds] to process some work W [e.g. in instructions] is proportional to W. Moreover, we assume parallelism without overheads.

Answer: `(50 + (90 * 4) + 15) minutes = 425 Minutes`
Assuming that three tasks are going in parallel, Speedup is:
$S_3 = T_1/T_3 = 620/425 = 1.46$

c) Can you devise a better strategy assuming that the four roommates bought another dryer? If yes, calculate the new laundry time

Answer: With two dryers, the third person in order can start drying as soon as his or her washing is done. This brings the total time to $(50) * 4 + 90 + 15 = 305$ minutes.

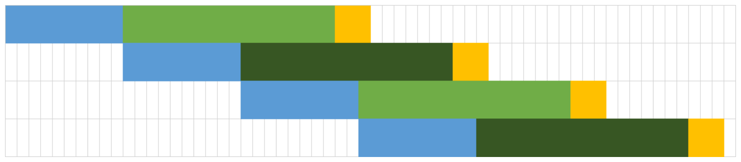
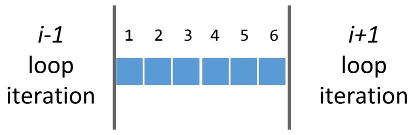


Figure 2: Perfect pipeline with two dryers

Task 3 – Pipelining II

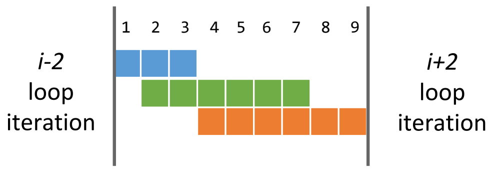
a) Each loop body has to perform one multiplication instruction before we can move to the next loop iteration. Since multiplication instruction has latency 6 cycles and we need to wait until it fully finishes, the whole loop will require `data.length * 6` cycles.

```
   for (int i = 0; i < size; i++) {
🟦   data[i] = data[i] * data[i];
   }
```



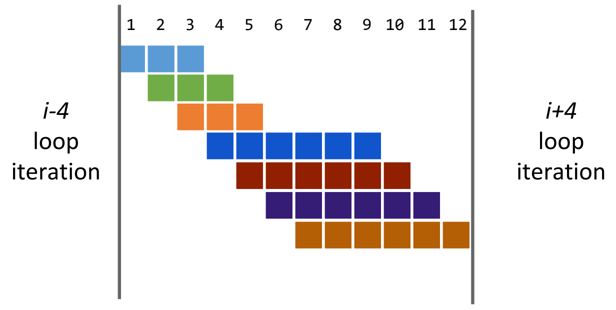
b) In the first cycle the computation of addition `i + 1` is issued. In the next cycle, the multiplication `data[i] * data[i]` can be issued as there is no dependency on the value of `j` that is still being computed. However, the last multiplication needs to wait until the execution of addition is finished as it uses the value `j` to index into the array. The total execution time is `data.length * 9 / 2` cycles as illustrated below:

```
   for (int i = 0; i < size; i += 2) {
🟦   j = i + 1;
🟩   data[i] = data[i] * data[i];
🟧   data[j] = data[j] * data[j];
   }
```



c) In the last example we simply execute all instructions, issuing one each cycle. This is possible as unrolling the loop four times provides enough work to hide the dependency between index computation (via addition) and the corresponding multiplication. The total execution time is therefore `data.length * 12 / 4` cycles as illustrated below:

```
   for (int i = 0; i < size; i += 4) {
🟦   j = i + 1;
🟩   k = i + 2;
🟧   l = i + 3;
🟦   data[i] = data[i] * data[i];
🟥   data[j] = data[j] * data[j];
🟪   data[k] = data[k] * data[k];
🟫   data[l] = data[l] * data[l];
   }
```



Task 4 – Identify potential parallelism

a) Inspect the code snippets for two loops below. For each loop explain if it is possible to parallelize the loop, and how would you do it?

(a) Loop-1

```
    int i = 1;
    while ( i < size) { // Loop for all values till the size
        if (data[i-1] > 0) // If the previous value is positive
            data[i] = data[i] * -1; // toggle the sign bit for this value
        i = i + 1; // increment the counter
    } // end while loop
```

Answer: This loop can not be parallelized as each iteration depends on the answer of previous iteration:

```
if (data[i-1] > 0)
```

Here the computation of current `data[i]` depends on the value of `data[i-1]` which will be available only after the previous iteration is finished. Due to this dependency, we can't parallelize the loop.

(b) Loop-2

```
    i = 0;
    while ( i < size) { // Loop for all values till the size
        data[i] = Math.sin(data[i]); // Calculate sin of the value
        i = i + 1; // increment the counter
    } // end while loop
```

Answer: In this loop, there is no dependency on any previous loop computations, so we can compute each iteration of the loop in parallel.