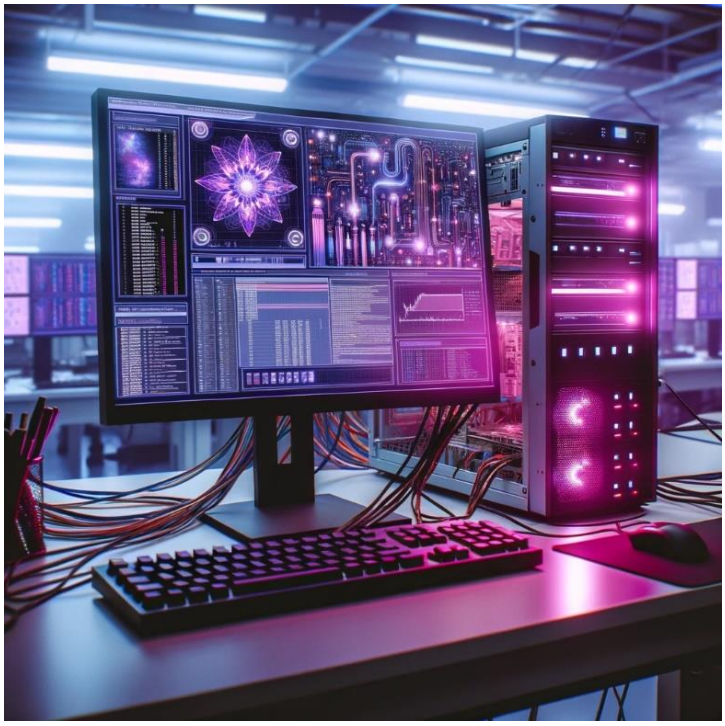


IC PRACTICE 2

Searching Candidates for Parallelization

Components of the Group:

Cristian Andrés Córdoba Silvestre (05988721G)

Nizar Nortes Pastor (74444345S)

Professor: Antonio Martínez Álvarez

3/11/2023

Escuela Politécnica Superior

University of Alicante

INDEX

Introduction to the problem

The aim of this task was to find, analyse and execute an issue demanding considerable computational power. We believed that an image filter which alters the image to appear pixelated and shifts the hues based on specified intervals to a 16-color set would be suitable and interesting for the practice. The algorithm, in order to read and write imaged, uses a C++ toolkit called Magick++. For the pixelation effect, the program determines the median shade of a defined segment (size can be selected via parameter) and then designates this color to all corresponding pixels, ensuring the entire segment appeared as a uniform color or a "bigger pixel". During the color computation it deduces its counterpart in the 16-color set, which then becomes the final assigned hue. This outlines how the retroGaming.cpp procedure adjusts any image, making it resemble visuals from retro video game.

Introduction to Magick++

Magick++ is a C++ library used by several projects such as Gimp, Octave, and various GNU Linux operating system families. The part that interests us is the Image class, one of its attributes being Pixels, which acts like a pixel map accessed by row and column. Each of these pixels contains a PixelPacket which has three numbers in percentage (from 0 to 1) representing the amount of red, green, and blue required to achieve that RGB color. The Image class allows for direct reading from a file and writing to save the image that's worked on, facilitating its manipulation for the program.

Configuration and management of the program

Here are the commands we used:

$ apt-cache search dev | grep magick *// verifies the package's existence*
$ sudo apt-get install libmagick++-dev *// installs the package*
$ sudo apt-get install libgraphicsmagick1-dev *// fetches a vital library (dependency)*

Using the above instructions we fully installed Magick++ which is key for our algorithm. Now in order to compile our program we should use the following command:

$ g++ Magick++-config --cxxflags --cppflags <Program_Name>.cpp -o <Executable_name> Magick++-config --ldflags –libs

Implementation and explanation of the code

Main: First, the code evaluates the count of arguments, initializes "Magick++" for image operations, and captures the pixel count intended for our enlarged pixel (this input is given as an argument). Following this setup, an introductory "for" loop processes the InputFile/OutputFile combinations, ensuring arguments appear in pairs. Utilizing "Image imagen(input_file_name)" the image is loaded, employing the Image constructor from the Magick++ toolkit. Then the image's dimensions are recorded and we loop to scan the image invoking “subSector” with the image, the boundaries and the pixel size for each enlarged pixel (called “tamBigPix”). When the loop is completed, the image is saved using the designated name with "imagen.write(output_file_name)". We use a try catch structure to manage any possible problem, and finally we calculate the time elapsed in the execution of the algorithm.

```
int main(int argc, char ** argv)
{
    long double tiempoTotal_1, tiempoTotal_2, tiempoTotal, tiempo1, tiempo1fin;
    InitializeMagick(*argv);

    if(argc<4)
    {
        cerr << "-----------------------------------------------------\n";
        cerr << "Error: prueba con " << argv[0] << " <Num_Pixels> <Nombre_Entrada> <Nombre_Salida> ... \n";
        cerr << "-----------------------------------------------------\n";
    }
    else if(argc>=4)
    {
        int tamBigPix = atoi(argv[1]);
        tiempoTotal_1=tiempo();
        for(int i=2; i<argc; i+=2)
        {
            tiempo1=tiempo();
            if(argc%2!=0 && i==argc-1)
                cerr << "ERROR: Falta archivo de destino para " << argv[argc-1] << endl;
            else
            {
                try
                {
                    Image imagen(argv[i]);

                    int ancho = imagen.columns(), alto = imagen.rows();
                    for(int j=0; j<ancho; j+=tamBigPix)
                    {
                        for(int k=0; k<alto; k+=tamBigPix)
                            subSector(imagen, j, k, tamBigPix, ancho, alto);
                    }

                    imagen.write(argv[i+1]);
                }
                catch(Exception &error_)
                {
                    cout << "Ha habido algún problema: " << error_.what() << endl;
                }
            }
            tiempo1fin=tiempo();
        }
        tiempoTotal_2=tiempo();
        tiempoTotal = tiempoTotal_2 - tiempoTotal_1;

        return 0;
```

```
void subSector(Image &ima, int x, int y, int rango, int ancho, int alto)
{
    double r = 0, g = 0, b = 0;
    ColorRGB auxColor;

    for (int i = x; (i < rango+x) && (i < ancho); i++)
    {
        for (int j = y; (j < rango+y) && (j < alto); j++)
        {
            auxColor = ima.pixelColor(i, j);
            r = r + (auxColor.red()*255);
            g = g + (auxColor.green()*255);
            b = b + (auxColor.blue()*255);
        }
    }
    r = r/(rango*rango);
    g = g/(rango*rango);
    b = b/(rango*rango);

    ColorRGB color(to16Palette(r, g, b));

    for (int i = x; (i < rango+x) && (i < ancho); i++) {
        for (int j = y; (j < rango+y) && (j < alto); j++) {
            ima.pixelColor(i, j, color);
        }
    }
}
```

SubSector: This segment is tasked with determining the mean color value of the larger pixels, painting these pixels uniformly while switching from a 256-color spectrum to a 16-color palette. Initially, it establishes variables for "r" (red), "g" (green), and "b" (blue) alongside an auxiliary ColorRGB variable that streamlines the average computation process. With the loops we scan the subsector and the values of "r", "g", and "b" are accumulated and divided by the sector's total pixel count to deduce the average. Using the mean value we call the "to16Pallete" function is invoked which outputs a value of type ColorRGB. Following this step, we determine the exact color to employ.

```
ColorRGB to16Palette(double r, double g, double b)
{
  if(0 <= r && r <= 255 && 0 <= g && g <= 255 && 0 <= b && b <= 255)
  {
        if(r <= 66)
        {
            if(g <= 66)
            {
              if(b <= 66)
              {
                  ColorRGB color16b(0, 0, 0); // 0 - Black
                  return color16b;
              }
              else if(b <= 198)
              {
                  ColorRGB color16b(0, 0, 132);   // 1 - RedBrown
                  return color16b;
              }
              else // 199 - 255
              {
                  ColorRGB color16b(0, 0, 255);   // 9 - OrangeRed
                  return color16b;
              }
            }
            else if(g <= 198)
            {
              if(b <= 66)
              {
                  ColorRGB color16b(0, 132, 0);   // 2 - Green
                  return color16b;
              }
              else // 67 - 255
              {
                  ColorRGB color16b(0, 132, 132); // 3 - BrownGreen
                  return color16b;
              }
            }
```

To16Palette: This function estimates a value from the 256-color spectrum to a 16-color variant. It uses specific intervals for each hue and if the computed average lands within a certain interval, the matching color from the 16-color palette is returned. These intervals are based on the red, green, and blue (RGB) values. In the 16-color palette, each hue uses 8 bits, amounting to 24 bits overall (in contrast to the original 8 bits used to encapsulate the entire color). And with this each value spans from 0 to 255, as we can see in the image of the code.

Examples of usage on images

Task 2.1

Given the CFGs attached in the delivery with this documentation, we can note that our issue is indeed parallelizable. This can be seen in the Main and in the function to16Palette where we make multiple choices, and these can be executed simultaneously by parallelizing the software. In these functions, several parallelizable chunks can be found in the form of concatenated if-else structures.

Task 2.2

Access of Read/Write to Variables

Most of the variables in the program are used for temporary storage or iteration:

- **argc** and **argv** are parameters of the main function, providing the number of command-line arguments and their values.
- tiempoTotal_1, tiempoTotal_2, tiempoTotal, tiempo1, and tiempo1fin are used for time measurement. They are read from and written into.
- tamBigPix is read once from the command line arguments and is used throughout the **main** function.
- Image imagen represents an image object. The program reads from and writes to it.
- Variables **ancho** and **alto** are used to store the image's dimensions.
- **r, g, b,** and **auxColor** in subSector are accumulators. They get modified frequently.

Type of Variables

Large Data Matrices: The main data matrix in the program is the Image imagen. This matrix stores pixel values of the image.

Rarely Accessed Data: Command-line parameters like argv[1] for the pixel size are read once and then remain constant throughout the program. Similarly, ancho and alto (width and height) are only read once per image processing.

Automatic Variables: There are several automatic variables like loop counters i, j, k, and accumulators r, g, b in the subSector function. These get created and destroyed frequently during the program execution.

Improvements in the program's flow

Image Decomposition:

Instead of processing the image pixel by pixel (or block by block), we could divide the image into a small number of big chunks or tiles. Each tile can then be processed independently and in parallel. For instance, if we have 4 processing units, we could divide the image into 4 roughly equal sections, and each section could be processed by a separate unit.

Eliminate Nested Conditionals in the Color Conversion:

The to16Palette function consists of many if-else conditions. This can be restructured using lookup tables or formulas to compact the logic and reduce branching, making it more efficient. A lookup table, for instance, might store precomputed values or boundaries to identify which color in the 16-color palette a given RGB value is assigned to.

Pixel Summation in subSector:

Instead of two nested loops in the subSector function where we sum the red, green, and blue values, and then another two nested loops to set the color, a good option would be to consider using parallel reduction for the summation part.

Preallocate Memory:

Instead of creating new ColorRGB objects in every call to to16Palette, we can consider preallocating a buffer of ColorRGB objects and simply updating their values. This can reduce the overhead of frequent memory allocations.

Offload Computation:

If available, consider offloading computation to a GPU, especially if working with large images. GPUs are well-suited for the kind of parallel processing required for image processing.

Problems with the cache

The cache can be a concern in two primary ways: temporally (accessing a memory location many times within a short period) and spatially (accessing memory locations that are close to one another within a short period). In a program, tight loops that process arrays can often benefit from the cache depending on how they're structured.

Pixel Processing in subSector Function

The image pixels are processed in a nested loop. This could cause cache misses, especially if the width (ancho) of the image is large. The reason being that the data might be contiguous in memory by rows and by iterating over the y coordinate in the inner loop, we jump to different rows, potentially causing cache misses.

Temporal Locality in Main Loop

In the main loop where images are processed, the entire image is loaded, processed, and then written out before the next image is processed. This means the program doesn't take advantage of temporal locality for cases where multiple images are processed. But this might not be a big concern unless the processing of each image is quite fast compared to the I/O times.

Task 2.3

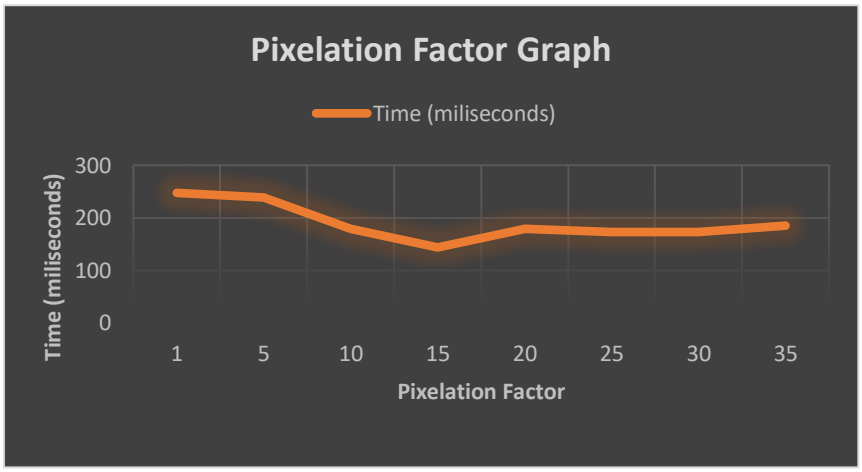
To adjust the problem's workload, there are several approaches. We might replace the image, affecting the program, with a bigger or smaller one, increase or decrease the pixelation factor taken by argument, or handle several images at once by making minor code changes.

Next, we will analyze the performance fluctuation as the problem's load changes; in this instance, by escalating the count of enhancements applied to the original image.
There are three main ways to alter the problem's load: varying the image's pixelation factor, changing the image size, and altering the number of images.

Varying the image's pixelation factor

We have used an image of a Husky of size 1200x675
8 Tests were done wit the following pixelation factors: 1,5,10,15,20,25,30,35
Here are the results:

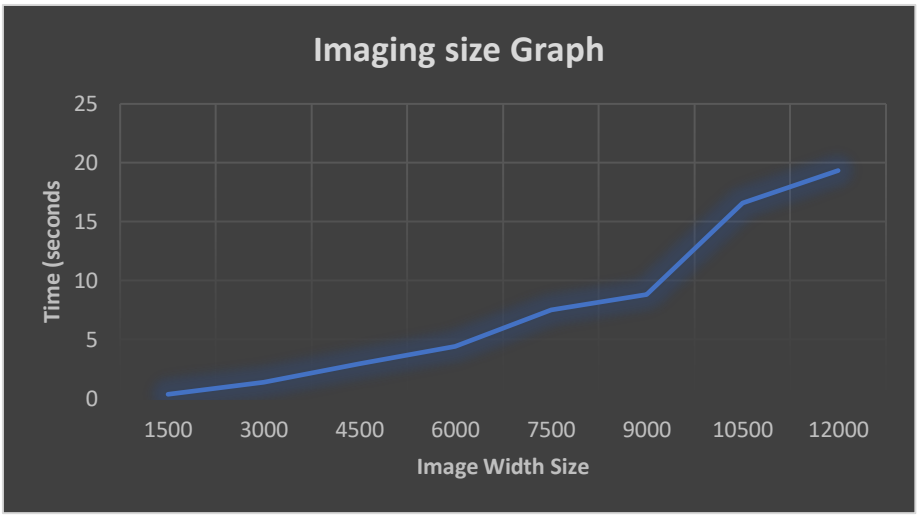
Pixelation Factor	1	5	10	15	20	25	30	35
Time (miliseconds)	247,31	239,08	178,83	143,49	178,48	172,83	172,2	184,9



Varying the image' size

We have used the Husky Image again. We used the following sizes:
1500x844,3000x1687,4500x2531,6000x3375,7500x4219,9000x5062,10500x5906,120
00x6750. Here are the results:

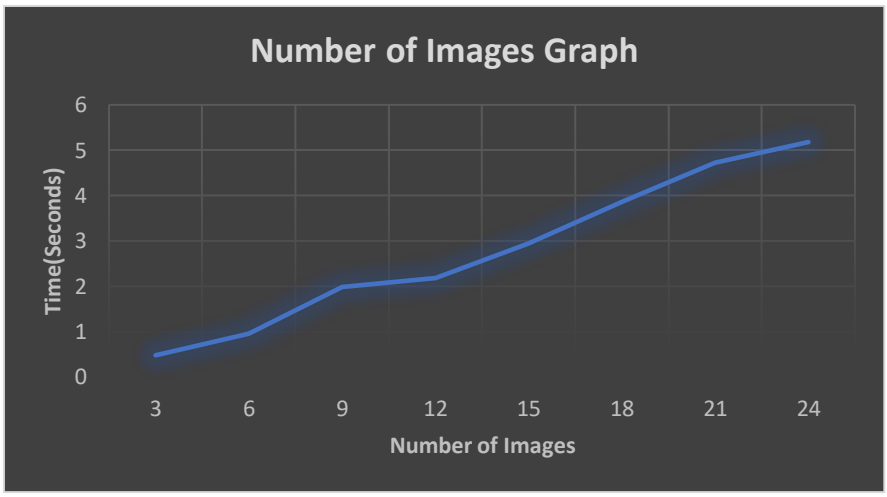
Image Size (Pixels)	1500x844	3000x1687	4500x2531	000x337	7500x4219	9000x5062	10500x5906	12000x6750
Time (Seconds)	0,31028	1,3104	2,88451	4,3736	7,48625	8,81246	16,58014	19,3333



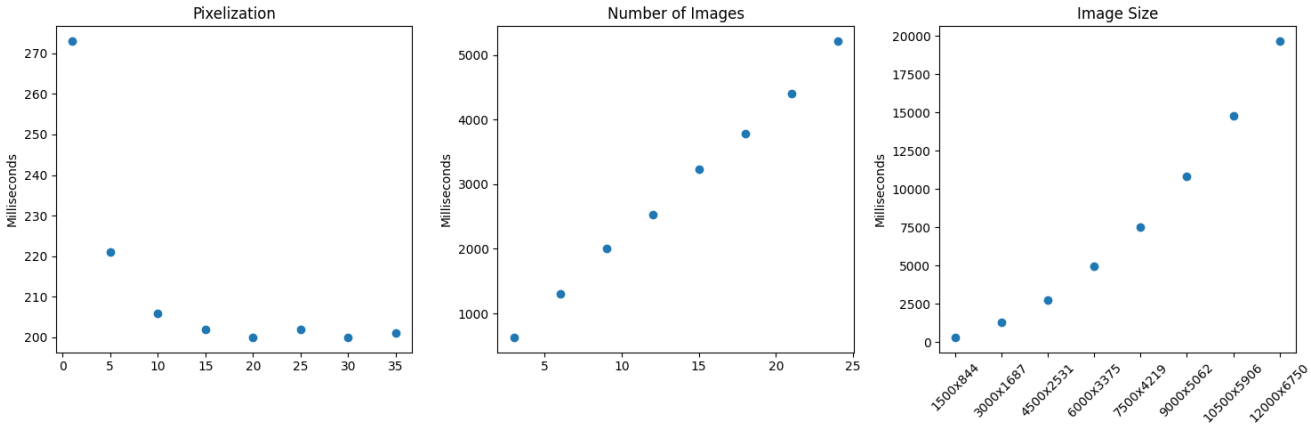
Varying the number of images

We used again the original Husky image. We have tested with the following number of images: 3,6,9,12,15,18,21,24. Here are the results:

Number of Images	3	6	9	12	15	18	21	24
Time (Seconds)	0,4774	0,94791	1,97755	2,1747	2,93821	3,86781	4,72214	5,17941



Here we have the executions performed in another computer for a more general comparison of the performances:



Task 2.4

Optimization Parameters -Ox:
GCC's standard optimization setting is -O0, where the code remains unoptimized. The elementary optimization level is -O1, where the compiler seeks to generate efficient code in a short compilation duration, as a rudimentary process. -O2 follows this, serving as the advised optimization tier unless there are specific system requirements since it introduces a few more options than what -O1 offers. Here, the compiler strives to boost code efficiency without sacrificing size or significantly extending the compilation period. Now, -O3 represents the maximum of optimization. It initiates certain optimizations that demand more compilation time and memory. Yet, using -O3 doesn't assure enhanced speed, it might decelerate a system because of extensive binaries and significant memory consumption.

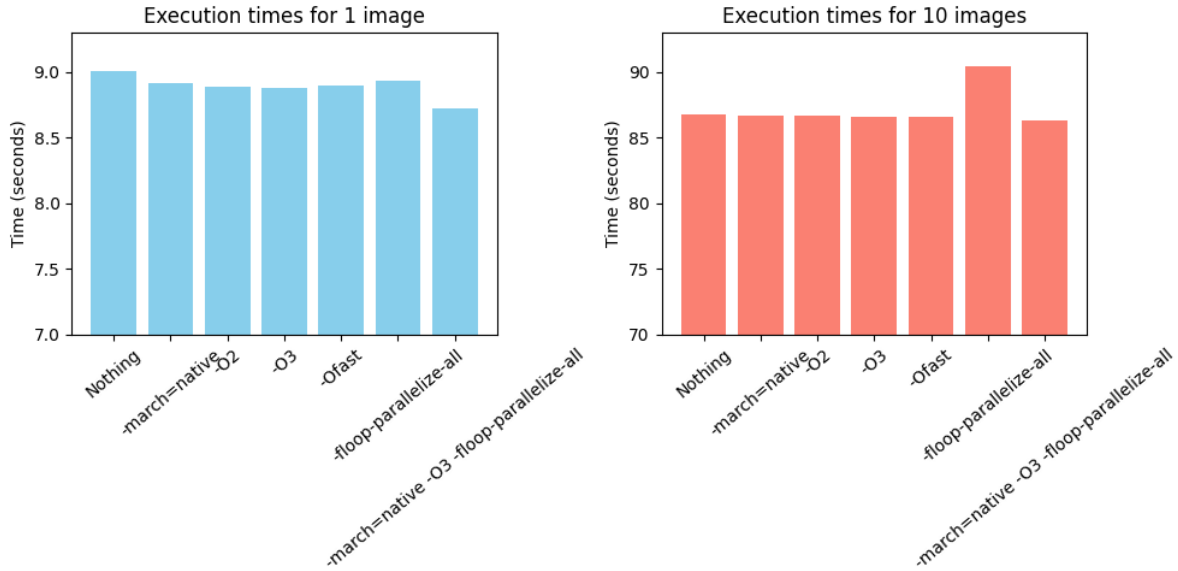
The –march Setting:
The -march setting dictates the kind of code suitable for a particular processor layout. CPUs vary in attributes, endorsing distinct command sets and code execution methodologies.
When uncertain about the CPU kind or the right configurations, one can employ the -march=native. In doing so, GCC tries to recognize the processor and automatically selects fitting options.

The -Ofast Setting:
The -Ofast setting encompasses all the -O3 enhancements and some that might not fit all compliant applications.

The -floop-parallelize-all Setting:
The -floop-parallelize-all configuration attempts to run every loop in parallel, which can be assessed to ensure no loop-dependent interactions, without verifying its efficiency.

Task 2.5 and 2.6

From the the test we have performed, we can see that various compiler flags and optimizations affect the execution time of the image pixelation algorithm, although it doesn't change much. It executes better with the correct use of parameters, but it does not affect our problem enough to be considerable and key for performance. Now we can see the graphs of the execution times for different cases (-O2, -O3…) and examples:



We can see that our problem is not very optimizable by the results of the different tests, maybe because of how the image management is performed in part with the help of an external library and not 100% inside the code.

Task 4

Problem 1:
If one tap takes 4 hours and the other takes 20, we can calculate the time for the two taps to fill the tank using the formula:

$$\frac{1}{t} = \frac{1}{4} + \frac{1}{20}$$

$$t = \frac{20}{6} = 3.\overline{3}\ hours$$

And the speedup and efficiency would be:

$$S = \frac{t^{o}}{t} = \frac{4}{3.\overline{3}} = 1.2$$

$$E = \frac{S}{n} = \frac{1.2}{2} = 0.6$$

The time with both taps is 3.33 hours and we got 1.2 of speedup and 0.6 of efficiency gain.

Problem 2:
If one tap takes 4 hours and the other takes 4, we can calculate the time for the two taps to fill the tank using the formula:

$$\frac{1}{t} = \frac{1}{4} + \frac{1}{4}$$

$$t = \frac{4}{2} = 2\ hours$$

And the speedup and efficiency would be:

$$S = \frac{t^{o}}{t} = \frac{4}{2} = 2$$

$$E = \frac{S}{n} = \frac{2}{2} = 1$$

The time with both taps is 2 hours and we got 2 of speedup (since is the double of throughput) and 1 of efficiency gain (because it's the double, the efficiency is of 100% more).

Problem 3:
If one tap takes 20 hours and the other takes 20, we can calculate the time for the two taps to fill the tank using the formula:

$$\frac{1}{t} = \frac{1}{20} + \frac{1}{20}$$

$$t = \frac{20}{2} = 10\ hours$$

And the speedup and efficiency would be:

$$S = \frac{t^{o}}{t} = \frac{20}{10} = 2$$

$$E = \frac{S}{n} = \frac{2}{2} = 1$$

The time with both taps is 10 hours and we got 2 of speedup and 1 of efficiency gain (just like the one before, it makes sense since we have the double of taps).

Problem 4:
If one tap takes 4 hours, another one takes 20 and the last one takes also 20, we can calculate the time for the three taps to fill the tank using the formula:

$$\frac{1}{t} = \frac{1}{4} + \frac{1}{20} + \frac{1}{20}$$

$$t = \frac{20}{7} = 2.86\ hours$$

And the speedup and efficiency would be:

$$S = \frac{t^{o}}{t} = \frac{4}{2.86} = 1.4$$

$$E = \frac{S}{n} = \frac{1.4}{3} = 0.4\bar{6}$$

The time with the three taps is 2.86 hours and we got 1.4 of speedup and 0.47 of efficiency gain.

The big.LITTLE is an ARM architecture that combines different types of processor cores within a single system. It is designed with two types of cores: the “big" cores for high computational power demanding tasks and "LITTLE" cores for less demanding operations.

In the context of this task we can see the analogy between the different taps and the big.LITTLE architecture, since the taps with different filling speeds represent the "big" and "LITTLE" cores. Utilizing both types of cores for different types of tasks to optimize performance and power consumption is similar to how using slow-filling taps along with the ones that fill the tank in less time can actually make a difference in the performance time and can benefit the speed of the process.