Main
It starts by initializing variables. The main function then looks at the argument passed using the getopt_long function. If the argument is t then tasks will be enabled. If argument is v then the initial values of x0,x1,y0,y1 are passed to the scaleandshift() function to be scaled and shifted. The default option is to present a list of options for the user.

It runs the serial implementation of mandelbrot by calling mandelbrotSerial(). It runs it 3 times and outputs the time required. It outputs the image of the mandelbrot by creating a ppm image file by calling writePPMImage().

It then runs the ispc implementation of mandelbrot by calling mandelbrot_ispc(). It runs it 3 times then outputs the time required. Using writePPMImage it outputs the mandelbrot created by using a ppm image file.

Once both implementations are done it uses verifyResult() to make sure that both gave the same results. If not then it deletes all data.

Main also starts running tasks but we will not go into that as asked.

static inline int mandel
This code commutes the results of on pair of the mandel set at a time. In a program that's supposed to be parallel this sounds counter productive. But ispc is a single program multiple data design. There can be multiple executions happening at the same time in different processors.

export void mandelbrot_ispc
The program initially computes the distances between the coordinates x and y. It has to do this first because these values are constant and needed for the foreach loop later. In the foreach j goes from 0 to height -1 and i goes from 0 to width -1. The foreach executes in parallel. Multiple instances are created that run together. Values i and j have a ranges of values depending on the instance. In this foreach loop the program maps over a 2d plain and performs the necessary calculations. It finds the complex plain needs and passes them as parameters to mendel(). The foreach loops calls mandel() which is not coded to be parallel but since ispc is SPMD multiple instances use different processors to compute the output. The output is stored in the output array.

void mandelbrot Serial
This is the serial implementation of the mandelbrot set. Nothing in this code is designed to run in parallel. Everything instruction is computed one after another. The program starts by finding the distances between the coordinates x and y. A main difference in this code compare to ispc is that it used a double nested for loop to iterate through every position. The for loop must compute each loop separately and wait for it's completion before moving on to the second loop. It computes the complex positions to pass on to mandel() in the loop. The results are stored in dex.

Describe the parts of the image that present challenges for SIMD execution. Why is this so? What adverse phenomenon/characteristic do they add to the execution (what is the name of this phenomenon/characteristic)?
The final value of each foreach loop needs to be written in output. Each gang of instances needs to finish writing into output before the next set of instances can start. According to ispc documentation you can't have a break inside of a foreach loop. The code in mandel() can't be written inside the foreach loop since it uses a break state. The mandel() code computes arithmetic and is not designed to run in parallel. But since it's simple arithmetic we can use SPMD, simple program multiple data. Different processors can be running mandel on different sets at the same time.

What is the maximum speedup you expect in theory (and ideally) given the specifications of your allocated CPUs? Why might the number you observe be less than this ideal

We have a 4 core cpu. We expect a speed up of about 9 to 10 times. The number might be less because not all processes are constantly busy. To run the next gang of instances all current instances might to finish writing to output. Some instances might takes longer to finish then others.

Part 1b

Assuming that the machine we are using has at least n cores, using n tasks should give us an a peak speedup of n times. This is because ISPC tasks are distributed among processor cores to run simultaneously. So if we have, say 4 cores, and we launch 4 tasks with an evenly distributed amount of work, each task will be assigned to a processor core, and will all run simultaneously. Launching more tasks than processor cores available will not continue to achieve speedups. No additional parallelism will be achieved then, because once all cores are working on their tasks, an additional task launched will run concurrently and asynchronously but not simultaneously, detracting from the performance of another task. There may be use case in which it is beneficial to launch more tasks than cores available, perhaps for other concurrency reasons, but in this case there is no efficiency to be gained.