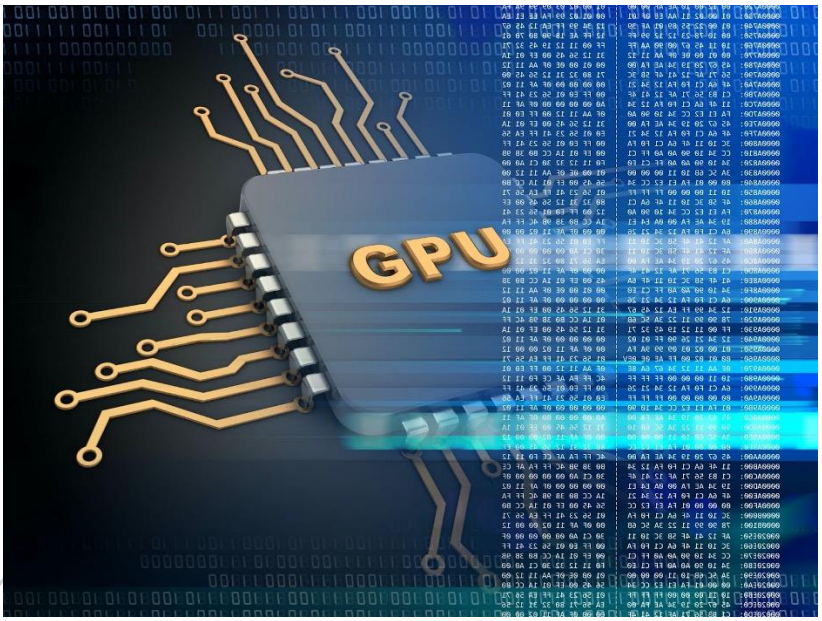
Universitat d'Alacant

8-12-2023

IC Practice 3

Parallelization via OpenMP and asynchronous scheduling of digital photo processing



Components of the group

Cristian Andrés Córdoba Silvestre (05988721G)

Nizar Nortes Pastor (74444345S)

Professor: Antonio Martínez Álvarez

University of Alacant

DEGREE IN COMPUTER SCIENCE

INDEX

Cristian Andrés Córdoba Silvestre (05988721G) Universidad de Alicante
Nizar Nortes Pastor (74444345S)

Task 0

Task 0.1 OpenMP pre-training:

0.1.1 What is the chunk variable used for?

The chunk variable is used to define the size of the chunks or blocks of iterations that are distributed among the threads in the parallel region. This concept is crucial in OpenMP's loop scheduling, particularly when using dynamic scheduling.

How It Works in the Code:

- **Declaration and Initialization:**

int chunk; declares the variable.
chunk = CHUNKSIZE; initializes chunk with the value defined by CHUNKSIZE (which is 100 in this case).

- **Usage in #pragma omp for Directive:**

The line #pragma omp for schedule(dynamic, chunk) uses the chunk variable in the scheduling clause schedule(dynamic, chunk) tells OpenMP how to distribute the iterations of the loop across the available threads.

Each thread, upon completing its current chunk of work, will dynamically request another chunk of chunk iterations to process until all iterations are complete. In this specific example, each thread will process 100 iterations of the loop at a time (since CHUNKSIZE is 100), then request the next 100 iterations, and so on, until all 1000 iterations are processed.

In summary, the chunk variable in the OpenMP code is used to define the size of the blocks of iterations that are dynamically distributed among the threads during the execution of a parallel loop. This helps in balancing the workload across threads and can impact the performance of the parallel execution.

0.1.2 Fully explains the pragma :

- The purpose of declaring these variables as shared is to ensure that all threads can access and modify the same memory locations for these variables.

- For a, b, and c, this means that all threads can read from and write to the same arrays, which is essential for the program's goal of adding elements of a and b into c.

- The chunk variable is shared because its value determines the chunk size for the dynamic scheduling in the parallel loop, and this size needs to be consistent across all threads.

Why is the variable i labeled as private in the pragma?

- The private clause ensures that each thread has its own separate instance of the variables listed.
- The variable i is used as a loop counter in the parallel for loop.
- Making i private is crucial to prevent race conditions. If i were shared, all threads would modify the same counter, leading to incorrect loop execution and unpredictable results.
- By declaring i as private, each thread gets its own copy of i, which it independently increments. This ensures that each thread processes a unique set of iterations in the loop.

0.1.3 What is the schedule for? What other possibilities are there?

The schedule clause in the OpenMP #pragma omp for directive specifies how loop iterations are distributed among the threads in the parallel region. In the provided code, the schedule(dynamic, chunk) clause is used. Let's see an explanation in more detail:

- **Dynamic Scheduling:**
  - In dynamic scheduling, the loop iterations are divided into chunks of a specified size (chunk in this case), and these chunks are dynamically assigned to threads as they become available.
  - The chunk size, set to CHUNKSIZE (100), dictates the number of iterations each thread will take at a time.
  - Dynamic scheduling is beneficial when loop iterations take varying amounts of time to execute, as it can lead to better load balancing. However, it can incur more overhead due to the dynamic allocation of iterations.

- The Role of chunk in Dynamic Scheduling:
  - In the code, chunk is set to 100, meaning that each thread will attempt to execute 100 iterations of the loop at a time before requesting more work. This can help reduce scheduling overhead compared to a scenario where each thread picks only one iteration at a time.

- **Other Scheduling Options in OpenMP**
  - **Static Scheduling (schedule(static, chunk)):**
    - In static scheduling, the total number of loop iterations is divided into chunks of approximately equal size, and these chunks are assigned to the threads at the start of the parallel region.
    - If the chunk size is not specified, OpenMP divides the iterations into as many chunks as there are threads.
    - Static scheduling is efficient when all iterations take approximately the same amount of time.

  - Guided Scheduling (schedule(guided, chunk)):
    - Guided scheduling is a variation of dynamic scheduling. Initially, large chunks are distributed to the threads, and the size of the chunks decreases over time.
    - The chunk parameter sets the minimum size of the chunks. If not specified, it defaults to a small number.
    - This approach can offer a balance between load balancing and scheduling overhead.

  - **Auto Scheduling (schedule(auto)):**
    - This allows the OpenMP runtime to choose the scheduling method.
    - It is useful when you are unsure about which scheduling method to choose.

  - **Runtime Scheduling (schedule(runtime)):**
    - The scheduling decision is deferred until runtime, and the decision is based on the OpenMP environment variable OMP_SCHEDULE.
    - This provides flexibility, as the scheduling can be changed without recompiling the code.

0.1.4 What times and other performance metrics can we measure in parallelized code sections with OpenMP?

- **Execution Time**
  - **Wall Clock Time:** This is the actual real-world time taken for a section of code to execute, often the primary metric of interest. It reflects the total time from the start to the end of the parallel region.
  - **How to Measure:** In OpenMP, you can use omp_get_wtime() to get the current time at the start and end of a parallel region and calculate the difference.
- **CPU Time**
  - **CPU Time:** This is the total time spent by all CPUs (or cores) on a task. It can be more than the wall clock time in a parallel execution due to multiple cores working simultaneously.
  - **How to Measure:** CPU time is not directly measured via OpenMP, but you can use system-specific tools or functions to obtain it.
- **Speedup**
  - **Speedup**: A comparison of execution time of the serial version of the code versus the parallel version. It's calculated as Speedup = Time_Serial / Time_Parallel.
  - **Ideal vs. Actual Speedup**: Ideal speedup is equal to the number of processors (or threads), but actual speedup is often lower due to overhead and non-parallelizable parts of code (as per Amdahl's Law).
- **Efficiency**
  - **What is it? ->** A measure of how effectively the parallel computing resources are used. It's calculated as Efficiency = Speedup / Number_of_Processors.
  - **High Efficiency:** Indicates a good balance between the computational workload and the number of processors. Lower efficiency can indicate overhead or imbalanced workload distribution.
- **Scalability**
  - **Strong Scalability:** Measures performance improvement with a fixed problem size as the number of processors increases.
  - **Weak Scalability:** Measures the change in performance when both the problem size and the number of processors are increased proportionally.
- **Load Balancing**
  - **Load Balancing:** This assesses how evenly the work is distributed across threads. Poor load balancing can lead to some threads doing more work than others, resulting in inefficiencies.
- **Overhead**
  - **Overhead Time:** Time spent on activities that are not part of the actual computation, like thread creation, synchronization, communication between threads, etc.
  - **How to Measure:** It's often calculated indirectly by comparing total execution time with actual computation time.

- **Tools for Performance Measurement**

  - **Profiler Tools:** Tools like Intel VTune, gprof, or Valgrind (with Callgrind) can be used for more detailed performance analysis.
  - **Built-in OpenMP Functions:** Functions like omp_get_wtime() for timing, and environment variables for controlling thread behavior, can be instrumental.

- **Practical Considerations**

  - Baseline Measurement: Always measure the performance of the serial (non-parallel) version of the code for comparison.
  - Varying Conditions: Test under different conditions, such as varying numbers of threads, different compilers, and optimization levels, to fully understand performance characteristics.
  - Bottlenecks Identification: Use these metrics to identify and address bottlenecks, such as synchronization delays or unbalanced workloads.

Task 0.2: Pre-training std::async

0.2.1 What is the std::launch::async parameter used for?

- The std::launch::async parameter in C++'s Standard Library is used with the std::async function to control how the associated task is executed in terms of threading. When you pass std::launch::async as a parameter to std::async, it enforces the following behavior:

  - **Forces Asynchronous Execution:** The task is executed on a new thread, separate from the calling thread. This means the task runs concurrently with the calling thread. If std::launch::async is not specified, the runtime has the freedom to choose whether to perform the task asynchronously or to defer its execution until the result is needed (lazy evaluation).

  - **Immediate Start:** The task starts executing immediately. Without std::launch::async, there's a possibility that the task execution could be deferred until the future's get() or wait() method is called.

In summary, when you use std::launch::async with std::async, you are explicitly stating that you want the task to run asynchronously on a separate thread as soon as possible. This is useful when you need to ensure that a task is executed in the background, without blocking the calling thread.

0.2.2 Calculate the time the program takes with std::launch::async and std::launch::deferred. What is the reason for the time difference?

- Using std::launch::async:

- std::launch::async forces the tasks to be executed asynchronously, i.e., on separate threads.
- In your code, task1 sleeps for 2000 ms and task2 for 3000 ms. Because they are executed concurrently (simultaneously on different threads), the total time taken by the program is dictated by the longer of the two tasks. Since task2 takes 3000 ms, the entire program execution time is roughly around 3000 ms.
- This behavior is ideal when the tasks are independent and can be run in parallel, effectively reducing the overall waiting time to the duration of the longest task.

- Using std::launch::deferred:

- When std::launch::deferred is used, the execution of the tasks is deferred until the future's get() or wait() method is called.
- In this mode, task1 and task2 are not started immediately. Instead, they are executed sequentially when their results are demanded.
- In your program, when task1.wait() is called, it triggers the execution of task1, which completes in 2000 ms. Then, when task2.get() is called, task2 starts and completes in 3000 ms. The total execution time becomes the sum of both task durations, which is 5000 ms.
- This sequential execution is useful when deferred work is preferable or when tasks depend on the results of previous tasks.
- In summary, std::launch::async results in parallel execution of tasks, leading to a shorter combined execution time, while std::launch::deferred leads to sequential execution, where the total time is the sum of individual task times.

0.2.3 What is the difference between the wait and get methods of std::future?

- **wait Method:**
  - The wait method blocks the calling thread until the asynchronous operation is complete.
  - It does not return the result of the asynchronous operation.
  - After wait returns, the std::future still holds the result of the asynchronous operation, and you can call get later to retrieve this result.
  - It can be called multiple times; subsequent calls to wait after the first one have no effect.

- **get Method:**
  - The get method also blocks the calling thread until the asynchronous operation is complete, just like wait.
  - It returns the result of the asynchronous operation.
  - Once get has been called, the std::future object becomes invalid, and you cannot call get again on the same std::future object. If you do, the program will throw an exception (std::future_error with the error condition std::future_errc::no_state).
  - get effectively consumes the std::future; after a call to get, the std::future no longer has an associated shared state.

- **In the context of our program:**
  - task1.wait(); waits for task1 to complete but does not retrieve its result. After task1.wait() completes, task1 is still a valid std::future object holding the result of the task.
  - int taskId = task2.get(); waits for task2 to complete and retrieves its result. After task2.get() has been called, task2 is no longer valid and cannot be used to wait for or retrieve the task's result again.

0.2.4 What advantages does std::async offer over std::thread?

Here we show the main advantages that we have found:

- **Ease of Use and Higher Abstraction Level:**
  - std::async provides a higher level of abstraction compared to std::thread. It allows you to focus on the task to be executed rather than the details of thread management. This makes std::async easier to use and more suitable for straightforward parallel tasks.

- **Automatic Management of Thread Lifecycle:**
  - When using std::async, you don't need to worry about joining or detaching threads. The std::future object returned by std::async takes care of the thread's lifecycle, ensuring that resources are properly released when the task is complete.

- **Return Values and Exception Handling:**
  - std::async simplifies the process of retrieving return values from asynchronous operations through the std::future object. Moreover, exceptions thrown in the asynchronous task are captured and can be rethrown in the context of the calling thread, simplifying exception handling.

- **Deferred Execution Option:**
  - With std::async, you have the option to defer the execution of a task until the result is needed (std::launch::deferred). This flexibility allows you to write code that can either run asynchronously or be deferred, depending on the situation.

- **Potential for Optimizations by the Implementation:**
  - std::async may benefit from optimizations made by the compiler or runtime, such as using a thread pool or other mechanisms to manage the execution of asynchronous tasks efficiently. This can lead to better performance and resource utilization in some cases.

- **Simpler Syntax for Single-Use Cases:**
  - For single-use, fire-and-forget tasks, std::async is simpler as it requires less boilerplate code compared to setting up a std::thread. This makes your code cleaner and more focused on the task at hand.

Task 0.3: Pre-training std::vector

Look at the following c++ program where an stl vector is initialized and filled with values:

```
#include <vector>
#include <iostream>
#include <chrono>
int main() {
// default initialization and add elements with push_back
auto start = std::chrono::high_resolution_clock::now();
std::vector<float> v1;
for (int i = 0; i < 10000; i++)
v1.push_back(i);
auto end = std::chrono::high_resolution_clock::now();
auto elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-
start);
std::cout<<"Default initialization:"<<elapsed.count()<<"ms"<<std::endl;
// initialized with required size and add elements with direct access
start = std::chrono::high_resolution_clock::now();
std::vector<float> v2(10000);
for (int i = 0; i < 10000; i++)
v1[i] = i;
end = std::chrono::high_resolution_clock::now();
elapsed = std::chrono::duration_cast<std::chrono::milliseconds>(end-start);
std::cout<<"Initialization with size:"<<elapsed.count()<<"ms"<<std::endl;
```

Answer the following questions:

0.3.1: Which of the two ways of initializing the vector and filling it is more efficient?

Why?

Initializing a std::vector with the required size and then setting each element's value is more efficient than using push_back on a default-initialized vector. This is especially true for large vectors or in performance-critical scenarios, as it minimizes memory reallocations and maximizes memory usage efficiency. Here's why:

- **Using push_back with Default Initialization:**
  - In this method, the vector starts empty and grows dynamically as new elements are added with push_back.
  - Each time an element is added, the vector might need to reallocate memory to accommodate the growing size. This reallocation involves allocating new memory, moving existing elements to the new space, and then freeing the old space.
  - This dynamic resizing can lead to multiple memory allocations and potentially inefficient use of memory, especially for large vector

- Initializing with Required Size and Direct Access:
  - Here, the vector is initialized with the specified size (10000 elements in this case), allocating all the required memory upfront.
  - Directly setting the value of each element avoids the need for reallocations since the vector already has sufficient space allocated for all its elements.
  - This method is more efficient in terms of both time (due to fewer allocations and no reallocations) and memory usage (as it avoids the potential over-allocation that can happen with push_back).

0.3.2 Could a problem occur when parallelizing the two for loops? Why?

Parallelizing the loop that uses push_back on v1 is problematic due to thread safety issues with std::vector's dynamic resizing. However, parallelizing the loop that assigns values to elements in the pre-sized v2 is generally safe and can be more efficient.

- **Parallelizing the push_back Operation in v1:**
  - This could lead to significant problems. The push_back method of std::vector is not thread-safe, meaning concurrent execution by multiple threads can lead to data races, corrupted data, or other undefined behavior.
  - std::vector does not automatically manage synchronization when its elements are modified or when it's resized. Concurrent push_back calls would likely interfere with each other as they modify the size and potentially trigger reallocations of the underlying array.
  - To safely parallelize this, you would need to add some form of synchronization, like mutexes, but that could negate the benefits of parallelization due to the overhead of locking.
- **Parallelizing the Element Assignment in v2:**
  - This is generally safe to parallelize because v2 is pre-sized, and each thread would be modifying a different part of the vector. There is no risk of reallocation or size modification during the operation.
  - Each loop iteration writes to a separate and distinct element, avoiding data races. As long as no two threads write to the same element (which they won't in this loop), this operation is thread-safe.
  - Parallelizing this loop could lead to performance improvements, especially on systems with multiple processors or cores.

Task 2: Parallelization of digital photo development

Task 2.1: Analyze the code and identify the different processes

In this task you will have to analyze the code using the strategies seen in the previous practice to identify the different processes that are applied on the RAW image and elaborate a dependency diagram. Identifies if some processes can be executed in parallel and the dependencies between them.

- As we can see in the dependence diagram, all the individuals functions can be parallelized, except for denoise, equalization and bloom.
- The reason for this is that all the functions, except for that three one, contains for loops or nested for loops, which an be easily parallelized using OpenMP clausures.
- Moreover, enhanceDetails function and bloom function van be executed in parallel using
- asynchronous programming as they are independent.
- They are independent because they don't modify directly the input image, and bloom function is executed immediately after enhanceDetails.

Task 2.2: Analyze the code of each process

In this task you must analyze the code corresponding to each process identified in task 3.1. Make a flow diagram of each one and identify possible parallelizable points.

- As we have said in task 2.1, the parallelizable points will be in the for loops.
- The for loops are good candidate for parallelization because each channel's operations are independent of the others. In some cases, we will able to parallelize across both the row (i) and column (j) dimensions of the image.
- Finally, as we have said before, enhanceDetails and bloom functions can be executed in parallel with asynchronous programming. The rest of the processes are not independent because all of them modify the input image.

Task 2.3 Parallelization

Is better performance obtained by parallelizing as many parts as possible?
No. It's not useful to parallelize the processes than only use OpenCV's functions, such as denoise or bloom.

Is performance degraded by parallelizing certain parts? If so, to what do you think this degradation is due?
The performance is degraded in the mentioned cases because OpenCV functions are already fully optimized, including potential multithreading. Trying to optimize these functions manually will lead to a decrease in the performance of the program probably due to the cost of creating and synchronizing multiple threads that do not offer us a big enough improvement over the already optimized code. Also, the problems may be related to multiple memory accesses generating a bottleneck.

Aspects that define the size of the problem.
The size of the problem is mainly given by the size of the raw. That's the case because the most time-consuming tasks involved the management of each pixel of the raw image, so a image with bigger size will have more pixels and, for that reason, it will increase the total execution time

Code control structures of special interest in the solution to the problem.

It is important to justify in as much detail as possible the changes that have been made with respect to the sequential version in order to parallelize the solution.OpenMP or std::async instructions and blocks used for code parallelization.
Let's comment each of the changes done in the parallelized solution:

- **gammaCurve**
    - The #pragma omp parallel for private(r) is a compound directive in OpenMP. It tells the compiler to parallelize the execution of the subsequent for loop. When this directive is encountered, OpenMP divides the loop iteration space among the available threads in the thread pool.
    - The private(r) clause specifies that each thread should have its own private copy of the variable r. In the loop, r is calculated based on the loop variable i.
    - Making r private ensures that each thread has its own instance of r, which prevents race conditions and data corruption that could occur if multiple threads were to access and modify the same r variable concurrently.
- The for loop is iterating 65,536 times (from 0 to 0x10000), and each iteration is independent of the others. By parallelizing this loop, the work can be divided among multiple CPU cores, allowing the loop to be executed in less time than it would in a sequential manner.

- **Debayer**
    - The #pragma omp parallel for collapse(2) directive s used with nested loops. In this case, there are two nested loops (one iterating over y and the other over x).
    - The collapse(2) directive instructs OpenMP to collapse these two nested loops into a single. His is particularly beneficial when dealing with large images, as the computation can be significantly sped up by utilizing multiple cores of the CPU.
    - The outer loop iterates over the rows (height), and the inner loop iterates over the columns (width) of the image. Collapsing these loops into a single loop allows the workload to be more evenly distributed across threads, especially if the image is not square (i.e., if width and height are significantly different).

- **EnhanceDetails**

- In this function, asynchronous programming is used to improve performance through parallel processing. This is achieved by dividing the image processing task into smaller sub-tasks and executing them concurrently across multiple threads. Here's a detailed explanation of how asynchronous programming is used:
- **Asynchronous Task Creation:**
  - The function enhanceDetails first prepares the image by converting it to float format and applying a Gaussian blur.
  - It then calculates the number of threads to use based on the hardware concurrency (std::thread::hardware_concurrency()) which typically returns the number of CPU cores.
  - The image is divided into slices (sliceHeight), with each slice representing a portion of the image rows. The division is based on the number of threads available.

- **Launching Asynchronous Tasks:**
  - For each slice of the image, an asynchronous task is created and launched using std::async with the std::launch::async policy. This policy ensures that each task is executed on a separate thread, concurrently with other tasks.
  - The function processSliceEnhanced is called by each asynchronous task. This function processes a slice of the image (from startRow to endRow), applying the enhancement algorithm on each pixel.

- **Parallel Processing:**
  - Each asynchronous task independently processes a different part of the image. This parallel processing allows the CPU to utilize multiple cores, significantly speeding up the computation compared to processing the image in a single thread.
  - This is particularly effective for computationally intensive tasks like image processing, where each pixel or group of pixels can be processed independently.

- **Synchronization and Completion:**
  - After launching all the asynchronous tasks, the enhanceDetails function waits for all of them to complete. This is done using a loop that calls get() on each future in the futures vector.
  - The call to get() blocks the main thread until the associated asynchronous task is completed, ensuring that all image processing is finished before proceeding.

- **GammaCorrection**
  - The **pragma omp parallel for** directive parallelizes the outer for loop (for (int i = 0; i < in.rows; ++i)), which iterates over the rows of the input image. This is especially beneficial for large images or computationally intensive tasks.
  - Since each thread can run on a separate processor core, the total execution time is reduced.

- **ColorBalance and ScreenMerge**
  - Again, the pragma omp parallel for in an analogous way.

- **MainFunction**
  - Finally, the enhanceDetails function and the bloom function are executed inn parallel using asynchronous programming., taking advantage of the independence of both functions.
  - Here, it’s important to use std::async with lambda functions to explicitly specify the arguments.
  - This way, the compiler can infer the correct types and function calls.

On the exploited parallelism

Reviewing the documentation for "Unit 3. Parallel Computing". State the following with respect to your practice:

Types of parallelism used. What communication alternatives (explicit or implicit) your program employs?

The solution uses explicit parallelism. That's because the programmer must specify explicitly the directives used or the clauses in reference to std::async

Parallel programming mode.

The parallel programming mode used is the Single Program Multiple Data (SPMD). That's because there's only one program, in which there are multiple functions used to achieve the functionality of convert a raw image to JPEG format.

Parallel programming style used.

Shared variables (explicit libraries within the sequential code, collective communication...)

Type of parallel program structure

It's master slave, because most of the individual functions are dependent between them (except for enhanceDetails and bloom). This fact discards the option of divide and conquer structure.

On the results of the tests performed and their context

Characterization of the parallel machine on which the program runs (e.g. number of compute nodes, cache system, memory type, etc.).

On Linux use the command: cat /proc/cpuinfo to access this information or lscpu.

NOTE: The content of the document can be seen in the folder of the delivery of this practice

What does the word ht in the output of the above command invocation mean? Does it appear on your practice computer?

- The term "ht" in the flags section stands for Hyper-Threading, a technology used by Intel to improve parallel processing performance.
- Hyper-Threading allows each physical core of the CPU to behave like two logical cores, essentially allowing it to handle two threads at a time. This can significantly improve performance in certain applications.
- Since "ht" is listed in the flags, it means that the CPU supports Hyper-Threading. Given the output, it's likely that our CPU is configured to use Hyper-Threading since it shows 4 logical processors for a 4-core CPU, suggesting two threads per core.

Calculation of the following (with associated graphs and brief explanation)

Gain in speed (speed-up) as a function of the number of computation units (threads in this case) and as a function of the parameters that modify the size of the problem (e.g. dimensions of a matrix, number of iterations considered, etc. ....). Comment on the results in a reasoned manner. What is the theoretical maximum speed gain?
Note: 2D or 3D graphics can be delivered as illustrative.

The Speed up can be calculated with the following formula: T_S/T_P, Where T_S stands for the execution time of the program without any parallelization and T_P stands for the execution time of the parallelized solution. Taking this into account, we will calculate the Speed up in each case. We always will take the time calculated by doing the average between 10 same measurements. In this way we ensure we obtain statistical accuracy

speed-up as a function of the number of computation units
We will express the computation units n in terms of the modules parallelized in the code
n=0:
$T_S=T_P$ = 26.374 s
Speed Up = 1.000

n=1 (gammaCurve parallelized):
T_S=26.374 s
T_P= 26.314 s
Speed up = 26.374/26.314 = 1.002

n=2 (debayer parallelized):
T_S=26.374 s
T_P= 25.997 s
Speed up = 26.374/25.997 = 1.014

n=3 (enhanceDetails parallelized):
T_S=26.374 s
T_P= 24.032 s
Speed up = 26.374/24.032 = 1.097

n=4 (gammaCorrection parallelized):
T_S=26.374 s
T_P= 23.896 s
Speed up = 26.374/23.896 = 1.099

n=5 (colorBalance parallelized):
T_S=26.374 s
T_P= 23.054 s
Speed up = 26.374/23.054 = 1.144

n=6 (screenMerge parallelized):
T_S=26.374 s
T_P= 22.865 s
Speed up = 26.374/22.865 = 1.153

n=7 (enhanceDetails and bloom executed in parallel):
T_S=26.374 s
T_P= 17.347 s
Speed up = 26.374/17.347 = 1.520

- As we can see, the more modules we parallelize, the more speed up we obtain.

- The biggest increases in the speed up are obtained when the most time-consuming processes are parallelized (such as colorBalance).

- Finally, the biggest increase is obtained when enhanceDetails and bloom functions are executed in parallel. That's because these are the two functions that take the most time to finish. Without any parallelization, enhanceDetails takes 7-8 seconds and bloom takes 8-10 seconds to complete.

- When using asynchronous programming, these two functions take only 8-10 seconds to execute (the time of the most time-consuming function). Therefore, between 7-8 seconds of speed-up are achieved in this way.

Speed Up as a function of the size of the problems

The size of the problem will depend of the size of the input raw image. It's obvious that with bigger image sizes we'll obtain larger execution times.
For this tests, we have used the following raw images:

1.NEF (14.6 MB)
2.NEF (21.7 MB)
3.NEF (25.6 MB. This is the original cat image of the practice)
Let's see the results

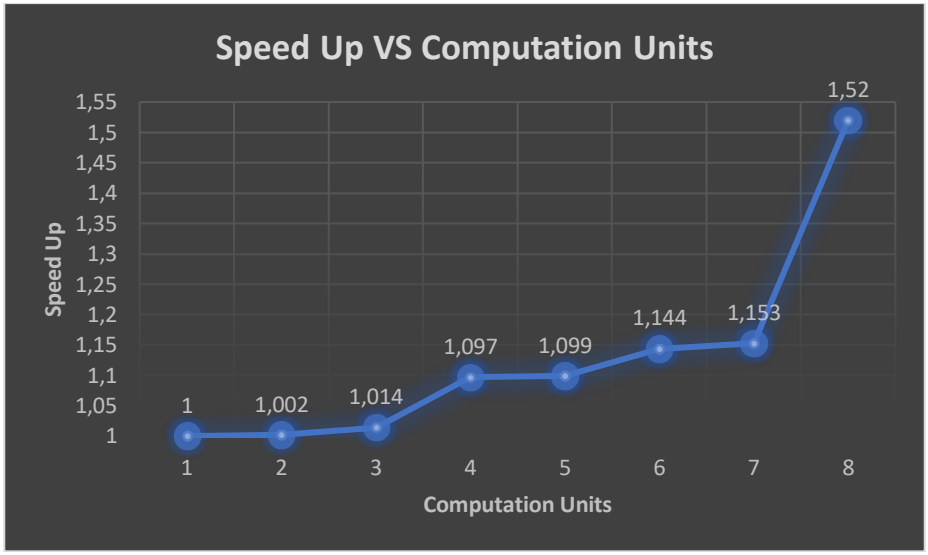
A) For 1.NEF
T_S=13.514 s
T_P= 8.862 s
Speed up = 13.514/8.662=1.524

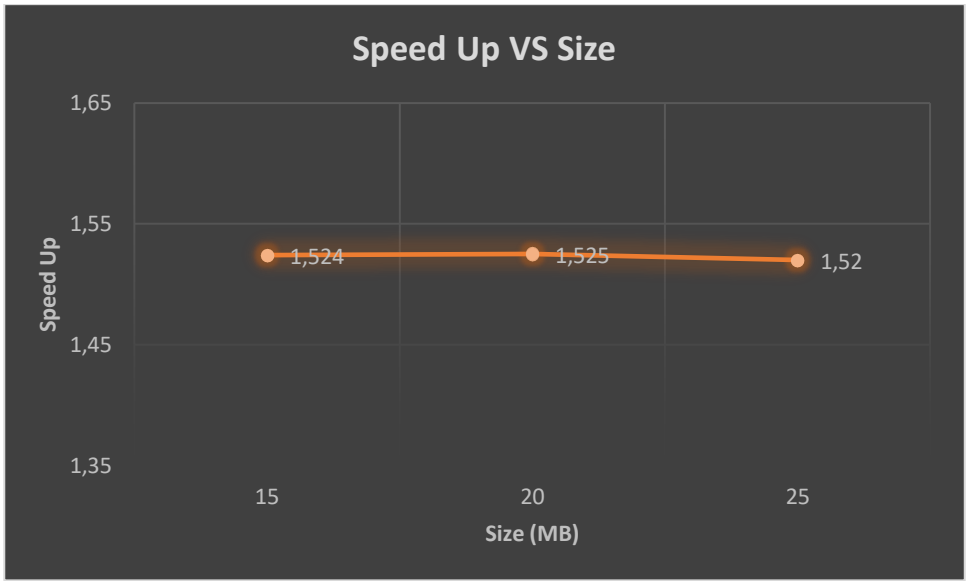
B) For 2.NEF
T_S=17.387 s
T_P= 11.203 s
Speed up = 13.514/8.662=1.525

C) For 3.NEF
T_S=26.374 s
T_P= 17.347 s
Speed up = 26.374/17.347=1.520

As we can see, the speed-up is practically the same for different raw image sizes.

Answer with justification: Which is the more efficient implementation of the 2?





To conclude, we can state that parallelizing the solution is the best option for every case, because for every size, we will achieve the same gain in speed by parallelizing the mentioned modules and by executing in parallel enhanceDetails and bloom using asynchronous programming. Finally, the ideal speedup is linear to the number of processors. Revising again the cpuinfo file, we observe that it shows 4 logical processors for a 4-core CPU. Therefore, the theoretical maximum speed gain would be:

T_S=26.374 s
Number_of_Cores = 4
$T_P = T_S$/Number_of_Cores = 26.374/4 s (application fully parallelized)

Theoretical maximum Speed Up = 26.374/(26.374/4) = 4

Annex: Individual Part

Cristian Part

Task 1: Study of the OpenMP API

First of all, I enumerate the examples that I've used:

- **Loops:** One of the most common structures in programming. Parallelizing loops, especially those that perform independent operations on different iterations, is a primary use case for OpenMP.
- **Data Structures:** Operations on arrays, matrices, trees, graphs, etc., can often be parallelized. For instance, parallel processing can be used for matrix multiplication or searching through a large array.
- **Algorithms:** Different algorithms, especially those that can be broken down into independent tasks, like sorting algorithms (merge sort, quicksort), numerical algorithms, or graph algorithms.
- **Task-Based Decomposition:** Software can be structured around tasks or jobs that can be executed in parallel. This is common in scenarios like processing multiple independent requests in a server application.
- **Concurrent Data Processing:** Applications that involve processing large datasets, like in big data analytics or scientific computing, where data can be partitioned and processed in parallel.
- **Pipeline Parallelism:** In some software architectures, processes or operations are structured in a pipeline manner, where different stages of the pipeline can be executed in parallel.
- **Object-Oriented Structures:** Parallelism can also be applied in object-oriented programming, for instance, by parallelizing methods of different objects that don't share state or depend on each other.
- **Recursive Functions:** Some recursive problems can be parallelized, especially those that follow the divide-and-conquer approach.

In each of these structures, OpenMP can be used to distribute the work across multiple threads, thereby improving the performance of the software on multi-core processors.
For each of these examples, I have created a test module in the file called OpenMPTests.cpp. Each of these modules can be tested freely. All these examples must be compiled with the -fopenmp flag of the GNU GCC compiler. Finally, we will explain each of the OpenMP pragmas used in the modules. These directives are fundamental in OpenMP programming, allowing developers to control parallel execution, task management, and data sharing between threads in a fine-grained manner.

- **#pragma omp parallel for:**
  - This directive is used to parallelize a for loop.
  - The iterations of the loop are divided among the available threads.
  - It's commonly used for data parallelism where each iteration of the loop can be executed independently.
- **#pragma omp parallel for reduction(max:maxVal):**
  - This extends the parallel for directive by adding a reduction clause.
  - It performs a reduction on the variables listed with a specified operation, in this case, max.

- maxVal is the variable being reduced; each thread will have its own copy of this variable. After all iterations, the maximum of all these copies is computed and stored in the original maxVal.

- **#pragma omp task:**
  - This directive defines a unit of work that is executed by a thread.
  - Tasks are independent chunks of work that can be executed concurrently.
  - The runtime system decides when and where to execute the tasks.

- **#pragma omp parallel:**
  - This creates a parallel region where the enclosed code block is executed by multiple threads.
  - It's a way to explicitly define a region of code that should be executed in parallel.

- **#pragma omp single:**
  - This is used inside a parallel region.
  - It specifies that the enclosed code block should be executed by only one thread (not necessarily the master thread).
  - Other threads skip this block and do not execute it.

- **#pragma omp parallel for reduction(+:sum):**
  - Similar to the reduction(max:maxVal) example, but here it performs a summation reduction on the sum variable.
  - Each thread calculates a partial sum, and at the end, these partial sums are added together to get the final result.

- **#pragma omp task shared(a):**
  - This creates a task similar to #pragma omp task, but with an explicit data-sharing attribute.
  - shared(a) indicates that the variable a is shared among the tasks.
  - This means that the tasks will access the same memory location of a, rather than having private copies.
- **#pragma omp taskwait:**
  - This is used within a parallel region to specify a wait point where the thread waits until all tasks created in the current task region are completed.
  - It ensures that all tasks generated before this point are completed before proceeding.

Nizar Part

Task 1: Study of the OpenMP API

In first place, I'm going to briefly explain each of the directives I used in my demonstration of the OpenMP API functionality:

- **Parallel**:

The "parallel" directive creates a parallel region where a block of code can be executed by multiple threads simultaneously. It is used to parallelize sections of code to leverage multiple processors or cores.

- **For Reduction**:

The "for" directive in combination with the "reduction" clause is used to parallelize loops while performing a reduction operation on a variable shared among threads, useful for parallelizing loop iterations with a reduction operation.

- **Critical**:

The "critical" directive creates a critical section, ensuring that only one thread at a time can execute the enclosed block of code. Can be used to protect shared resources or sections of code that must be executed serially to avoid race conditions.

- **Section**:

The "section" directive is used within a parallel region to define individual sections of work that can be executed in parallel by different threads. Useful when different parts of a parallel region can be executed independently.

- **Task**:

The "task" directive creates a task, which is a unit of work that can be executed asynchronously by any available thread. Allows for a parallelism by specifying tasks that can run independently.

- **Single**:

The "single" directive specifies a block of code that should be executed by only one thread in the parallel region, useful for code that needs to be executed by a single thread, such as initialization tasks.

- **Taskwait**:

The "taskwait" directive is used to synchronize the execution of tasks. It ensures that a thread waits until all its child tasks are complete before proceeding, used to control the flow of execution in parallelism based on tasks.

- **Atomic**:

The "atomic" directive ensures that a specific operation is executed atomically, which means that is executed without interruption from other threads of without accessing at the same time. Useful to avoid problems when updating shared variables.

- **Barrier**:

The "barrier" directive creates a synchronization point that all threads must reach before any of them can go on beyond that point. Ensures that all threads have completed their work before moving to the next phase of execution.

- **Private**:
The “private” clause is used with directives like “parallel” or “for” to create private copies of variables for each thread, preventing interference when modifying its value or accessing to the memory of the variable. This is great for avoiding data conflicts when each thread needs its own copy of a variable.

- **Ordered**:
The “ordered” directive is used within a loop to assure the sequential order of iterations even in a parallel context, used when maintaining the order of execution is crucial such as in algorithms where order matters or in loops where the output has a specific format.

Now, for the code demonstration of this directives I’ve developed 7 simple functions that use all previously explained calls for different structures of software. The functions are the following:

1. **vectorLoop()**: Demonstration of the usage of the “parallel for reduction” when adding all values of a vector through a loop.

2. **criticalRegion()**: Here I test the use of a “critical” for a region of code in a parallelization.

3. **parallelSections()**: Demonstrating the usage of the directive “section” in order to define a section of code that can be parallelized with “parallel sections”.

4. **taskParallelization()**: Creating two different functions declared as “task” and parallelizing them with the directive “single”. Also here is demonstrated the use of “taskwait” for resynchronizing the threads.

5. **forAtomic()**: Here I show the functionality of “atomic” on an operation in a parallel for loop.

6. **barrierThreads()**: Demonstration of the use of the directive “barrier” in a parallelized code, where all treads wait for the rest to end their execution.

7. **threadPrivate()**: A function where I parallelize with a number of threads (“num_threads”) and use a private variable for each thread, outputting the number of thread in order.

This would be a solid demonstration of the possibilities of the OpenMP API on different structures and types of operations and pieces of code. All of this can be found and tested in the file “**Task1-Nizar.cpp**” on the directory of this practice delivery.

Work done by each person

Task	Person/people in charge
0.1	Cristian
0.2	Cristian
0.3	Nizar
2.1	Both
2.2	Both
2.3	Both

Thank You!