Continue

In this tutorial, we’ll analyze two common processing techniques used by the operating system (OS). We usually need these two concepts when we have multiple processes waiting to be executed. 2. Key Definitions Before getting into too much detail about concurrency and parallelism, let’s have a look at the key definitions used in the descriptions of these two processing methods: Multiprocessing: The employment of two or more central processing units (CPUs) within a single computer system is known as multiprocessing. Multithreading: This technique allows a single process to have multiple code segments, like threads. These segments run concurrently within the context of that process. Distributed Computing: A distributed computing system consists of multiple computer systems that are run as a single system. The computers in a system can be physically close to each other and connected by a local network, or they can be distant and connected by a wide area network. Multicore processor: It’s a single integrated processor that includes multiple core processing units. It’s also known as chip multiprocessor (CMP). Pipelining: It’s a technique where multiple instructions are overlapped during execution. 3. Concurrency Concurrency actually means that multiple tasks can be executed in an overlapping time period. One of the tasks can begin before the preceding one is completed; however, they won’t be running at the same time. The CPU will adjust time slices per task and appropriately switch contexts. That’s why this concept is quite complicated to implement and especially debug. 3.1. How Does Concurrency Works? The main aim of concurrency is to maximize the CPU by minimizing its idle time. While the current thread or process is waiting for input-output operations, database transactions, or launching an external program, another process or thread receives the CPU allocation. On the kernel side, the OS sends an interrupt to the active task to stop it: If two or more jobs are running on the same core of a single-core or multi-core CPU, they can access the same resources at the same time. Even though data read operations are performed in parallel and are safe, during write accesses, programmers must maintain data integrity. Efficient process scheduling has a crucial role in a concurrent system. First-in, first-out (FIFO), shortest-job-first (SJF), and round-robin (RR) are popular task scheduling algorithms. As we mentioned, it can be complicated to implement and debug concurrency, especially at the kernel level, so there can be starvation between processes when one of the tasks gets the CPU for too long. In order to prevent this situation, interrupts are designed, and they help the CPU allocate other processes. This is also called preemptive scheduling. The OS, like any other application, requires CPU time to adjust concurrent tasks. 4. Parallelism Parallelism is the ability to execute independent tasks of a program in the same instant of time. Contrary to concurrent tasks, these tasks can run simultaneously on another processor core, another processor, or an entirely different computer that can be a distributed system. As the demand for computing speed from real-world applications increases, parallelism becomes more common and affordable. 4.1. How the Parallelism Works? The figure below represents an example of distributed systems. As we previously mentioned, a distributed computing system consists of multiple computer systems, but it’s run as a single system. The computers that are in a system can be physically close to each other and connected by a local network, or they can be distant and connected by a wide area network: Parallelism is a must for performance gain. There’s more than one benefit of parallelism, and we can implement it on different levels of abstractions: As we can see in the figure above, distributed systems are one of the most important examples of parallel systems. They’re basically independent computers with their own memory and IO. For example, we can have multiple functional units, like several adders and multipliers, managed by one instruction set. Process pipelining is another example of parallelism. Even at chip level, parallelism can increase concurrency in operations. We can also take advantage of parallelism by using multiple cores on the same computer. This makes various edge devices, like mobile phones, possible. 5. Concurrency vs Parallelism Let’s take a look at how concurrency and parallelism work with the below example. As we can see, there are two cores and two tasks. In a concurrent approach, each core is executing both tasks by switching among them over time. In contrast, the parallel approach doesn’t switch among tasks, but instead executes them in parallel over time: This simple example for concurrent processing can be any user-interactive program, like a text editor. In such a program, there can be some IO operations that waste CPU cycles. When we save a file or print it, the user can concurrently type. The main thread launches many threads for typing, saving, and similar activities concurrently. They may run in the same time period; however, they aren’t actually running in parallel. In contrast, we can give an example of Hadoop-based distributed data processing for a parallel system. It entails large-scale data processing on many clusters and it uses parallel processors. Programmers see the entire system as a single database. 5.1. Potential Pitfalls in Concurrency and Parallelism As we noted earlier in this tutorial, concurrency and parallelism are complex ideas and require advanced development skills. Otherwise, there could be some potential risks that jeopardize the system’s reliability. For example, if we don’t carefully design the concurrent environment, there can be deadlocks, race conditions, or starvation. Similarly, we should also be careful when we’re doing parallel programming. We need to know where to stop and what to share. Otherwise, we could face memory corruption, leaks, or errors. 5.2. Programming Languages That Support Concurrency and Parallelism Simultaneously executing processes and threads is the main idea that concurrent programming languages use. On the other hand, languages that support parallelism make programming constructs able to be executed on more than one machine. Instruction and data stream are key terms for the parallelism taxonomy. These languages include some important concepts. Instead of learning the language itself, it would be better to understand the fundamentals of these subjects: Systems programming: It’s basic OS and hardware management that can include system call implementation and writing a new scheduler for an OS. Distributed computing: As we mentioned earlier, it’s a must for parallel CPUs to be utilized. Performance computing: This concept is necessary for CPU resource optimization. Now let’s categorize the different languages, frameworks, and APIs: Shared memory languages: Orca, Java, C (with some additional libraries) Object-oriented parallelism: Java, C++, Nexus Distributed memory: MPI, Concurrent C, Ada Message passing: Go, Rust Parallel functional languages: LISP Frameworks and APIs: Spark, Hadoop These are just some of the different languages which we can use for concurrency and parallelism. Instead of the whole language itself, there are library extensions, such as POSIX thread library for C programming language. With this library, we can implement almost all of the concurrent programming concepts, such as semaphores, multi-threads, and condition variables. 6. Conclusion In this article, we discussed how concurrency and parallelism work, and the differences between them. We shared some examples related to these two concepts and explained why we need them in the first place. Lastly, we gave a brief summary of the potential pitfalls in concurrency and parallelism and listed the programming languages that support these two important concepts. I believe this answer to be more correct than the existing answers and editing them would have changed their essence. I have tried to link to various sources or wikipedia pages so others can affirm correctness. Concurrency: the property of a system which enables units of the program, algorithm, or problem to be executed out-of-order or in partial order without affecting the final outcome 1 2. A simple example of this is consecutive additions: 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 Due to the commutative property of addition the order of these can be re-arranged without affecting correctness; the following arrangement will result in the same answer: (1 + 9) + (2 + 8) + (3 + 7) + (4 + 6) + 5 + 0 = 45 Here I have grouped numbers into pairs that will sum to 10, making it easier for me to arrive at the correct answer in my head. Parallel Computing: a type of computation in which many calculations or the execution of processes are carried out simultaneously 3 4. Thus parallel computing leverages the property of concurrency to execute multiple units of the program, algorithm, or problem simultaneously. Continuing with the example of consecutive additions, we can execute different portions of the sum in parallel: Execution unit 1: 0 + 1 + 2 + 3 + 4 = 10 Execution unit 2: 5 + 6 + 7 + 8 + 9 = 35 Then at the end we sum the results from each worker to get 10 + 35 = 45. Again, this parallelism was only possible because consecutive additions have the property of concurrency. Concurrency can be leveraged by more than just parallelism though. Consider pre-emption on a single-core system: over a period of time the system may make progress on multiple running processes without any of them finishing. Indeed, your example of asyncronous I/O is a common example of concurrency that does not require parallelism. Confusion The above is relatively straightforward. I suspect people get confused because the dictionary definitions do not necessarily match what was outlined above: Concurrent: occurring or existing simultaneously or side by side 5. Concurrency: the fact of two or more events or circumstances happening or existing at the same time From searching on google: "define: concurrency". The dictionary defines "concurrency" as a fact of occurrence, whereas the definition in the computing vernacular is a latent property of a program, property, or system. Though related these things are not the same. Personal Recommendations I recommend using the term "parallel" when the simultaneous execution is assured or expected, and to use the term "concurrent" when it is uncertain or irrelevant if simultaneous execution will be employed. I would therefore describe simulating a jet engine on multiple cores as parallel. I would describe Makefiles as an example of concurrency. Makefiles state the dependencies of each target. When targets depend on other targets this creates a partial ordering. When the relationships and recipes are comprehensively and correctly defined this establishes the property of concurrency: there exists a partial order such that order of certain tasks can be re-arranged without affecting the result. Again, this concurrency can be leveraged to build multiple rules simultaneously but the concurrency is a property of the Makefile whether parallelism is employed or not. The terms concurrency and parallelism are often used in relation to multithreaded programs. On the surface, it may seem as if concurrency and parallelism may be referring to the same concepts. However, concurrency and parallelism actually have different meanings. In this concurrency vs. parallelism tutorial I will explain what these concepts mean. Just to be clear, in this text I look at concurrency and parallelism within a single application - a single process. Not among multiple applications, processes or computers. Concurrency vs Parallelism Tutorial Video If you prefer video, I have a video version of this tutorial here: Concurrency vs Parallelism Tutorial Video Concurrency Concurrency means that an application is making progress on more than one task - at the same time or at least seemingly at the same time (concurrently). If the computer only has one CPU the application may not make progress on more than one task at exactly the same time, but more than one task is in progress at a time inside the application. To make progress on more than one task concurrently the CPU switches between the different tasks during execution. This is illustrated in the diagram below: Parallel Execution Parallel execution is when a computer has more than one CPU or CPU core, and makes progress on more than one task simultaneously. However, parallel execution is not referring to the same phenomenon as parallelism. I will get back to parallelism later. Parallel execution is illustrated below: Parallel Concurrent Execution It is possible to have parallel concurrent execution, where threads are distributed among multiple CPUs. Thus, the threads executed on the same CPU are executed concurrently, whereas threads executed on different CPUs are executed in parallel. The diagram below illustrates parallel concurrent execution. Parallelism The term parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time. Thus, parallelism does not refer to the same execution model as parallel concurrent execution - even if they may look similar on the surface. To achieve true parallelism your application must have more than one thread running - and each thread must run on separate CPUs / CPU cores / graphics card GPU cores or similar. The diagram below illustrates a bigger task which is being split up into 4 subtasks. These 4 subtasks are being executed by 4 different threads, which run on 2 different CPUs. This means, that parts of these subtasks are executed concurrently (those executed on the same CPU), and parts are executed in parallel (those executed on different CPUs). If instead the 4 subtasks were executed by 4 threads running on each their own CPU (4 CPUs in total), then the task execution would have been fully parallel. However, it is not always easy to break a task into exactly as many subtasks as the number of CPUs available. Often, it is easier to break a task into a number of subtasks which fit naturally with the task at hand, and then let the thread scheduler take care of distributing the threads among the available CPUs. Concurrency and Parallelism Combinations To recap, concurrency refers to how a single CPU can make progress on multiple tasks seemingly at the same time (AKA concurrently). Parallelism on the other hand, is related to how an application can parallelize the execution of a single task - typically by splitting the task up into subtasks which can be completed in parallel. These two execution styles can be combined within the same application. I will cover some of these combinations below. Concurrent, Not Parallel An application can be concurrent, but not parallel. This means that it makes progress on more than one task seemingly at the same time (concurrently), but the application switches between making progress on each of the tasks - until the tasks are completed. There is no true parallel execution of tasks going in parallel threads / CPUs. Parallel, Not Concurrent An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel. However, each task (+ subtask) is completed before the next task is split up and executed in parallel. Neither Concurrent Nor Parallel Additionally, an application can be neither concurrent nor parallel. This means that it works on only one task at a time, and the task is never broken down into subtasks for parallel execution. This could be the case for small command line applications where it only has a single job which is too small to make sense to parallelize. Concurrent and Parallel Finally, an application can also be both concurrent and parallel in two ways: The first is simple parallel concurrent execution. This is what happens if an application starts up multiple threads which are then executed on multiple CPUs. The second way is that the application both works on multiple tasks concurrently, and also breaks each task down into subtasks for parallel execution. However, some of the benefits of concurrency and parallelism may be lost in this scenario, as the CPUs in the computer are already kept reasonably busy with either concurrency or parallelism alone. Combining it may lead to only a small performance gain or even performance loss. Make sure you analyze and measure before you adopt a concurrent parallel model blindly. I believe this answer to be more correct than the existing answers and editing them would have changed their essence. I have tried to link to various sources or wikipedia pages so others can affirm correctness. Concurrency: the property of a system which enables units of the program, algorithm, or problem to be executed out-of-order or in partial order without affecting the final outcome 1 2. A simple example of this is consecutive additions: 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 Due to the commutative property of addition the order of these can be re-arranged without affecting correctness; the following arrangement will result in the same answer: (1 + 9) + (2 + 8) + (3 + 7) + (4 + 6) + 5 + 0 = 45 Here I have grouped numbers into pairs that will sum to 10, making it easier for me to arrive at the correct answer in my head. Parallel Computing: a type of computation in which many calculations or the execution of processes are carried out simultaneously 3 4. Thus parallel computing leverages the property of concurrency to execute multiple units of the program, algorithm, or problem simultaneously. Continuing with the example of consecutive additions, we can execute different portions of the sum in parallel: Execution unit 1: 0 + 1 + 2 + 3 + 4 = 10 Execution unit 2: 5 + 6 + 7 + 8 + 9 = 35 Then at the end we sum the results from each worker to get 10 + 35 = 45. Again, this parallelism was only possible because consecutive additions have the property of concurrency. Concurrency can be leveraged by more than just parallelism though. Consider pre-emption on a single-core system: over a period of time the system may make progress on multiple running processes without any of them finishing. Indeed, your example of asyncronous I/O is a common example of concurrency that does not require parallelism. Confusion The above is relatively straightforward. I suspect people get confused because the dictionary definitions do not necessarily match what was outlined above: Concurrent: occurring or existing simultaneously or side by side 5. Concurrency: the fact of two or more events or circumstances happening or existing at the same time From searching on google: "define: concurrency". The dictionary defines "concurrency" as a fact of occurrence, whereas the definition in the computing vernacular is a latent property of a program, property, or system. Though related these things are not the same. Personal Recommendations I recommend using the term "parallel" when the simultaneous execution is assured or expected, and to use the term "concurrent" when it is uncertain or irrelevant if simultaneous execution will be employed. I would therefore describe simulating a jet engine on multiple cores as parallel. I would describe Makefiles as an example of concurrency. Makefiles state the dependencies of each target. When targets depend on other targets this creates a partial ordering. When the relationships and recipes are comprehensively and correctly defined this establishes the property of concurrency: there exists a partial order such that order of certain tasks can be re-arranged without affecting the result. Again, this concurrency can be leveraged to build multiple rules simultaneously but the concurrency is a property of the Makefile whether parallelism is employed or not. I believe this answer to be more correct than the existing answers and editing them would have changed their essence. I have tried to link to various sources or wikipedia pages so others can affirm correctness. Concurrency: the property of a system which enables units of the program, algorithm, or problem to be executed out-of-order or in partial order without affecting the final outcome 1 2. A simple example of this is consecutive additions: 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 Due to the commutative property of addition the order of these can be re-arranged without affecting correctness; the following arrangement will result in the same answer: (1 + 9) + (2 + 8) + (3 + 7) + (4 + 6) + 5 + 0 = 45 Here I have grouped numbers into pairs that will sum to 10, making it easier for me to arrive at the correct answer in my head. Parallel Computing: a type of computation in which many calculations or the execution of processes are carried out simultaneously 3 4. Thus parallel computing leverages the property of concurrency to execute multiple units of the program, algorithm, or problem simultaneously. Continuing with the example of consecutive additions, we can execute different portions of the sum in parallel: Execution unit 1: 0 + 1 + 2 + 3 + 4 = 10 Execution unit 2: 5 + 6 + 7 + 8 + 9 = 35 Then at the end we sum the results from each worker to get 10 + 35 = 45. Again, this parallelism was only possible because consecutive additions have the property of concurrency. Concurrency can be leveraged by more than just parallelism though. Consider pre-emption on a single-core system: over a period of time the system may make progress on multiple running processes without any of them finishing. Indeed, your example of asyncronous I/O is a common example of concurrency that does not require parallelism. Confusion The above is relatively straightforward. I suspect people get confused because the dictionary definitions do not necessarily match what was outlined above: Concurrent: occurring or existing simultaneously or side by side 5. Concurrency: the fact of two or more events or circumstances happening or existing at the same time From searching on google: "define: concurrency". The dictionary defines "concurrency" as a fact of occurrence, whereas the definition in the computing vernacular is a latent property of a program, property, or system. Though related these things are not the same. Personal Recommendations I recommend using the term "parallel" when the simultaneous execution is assured or expected, and to use the term "concurrent" when it is uncertain or irrelevant if simultaneous execution will be employed. I would therefore describe simulating a jet engine on multiple cores as parallel. I would describe Makefiles as an example of concurrency. Makefiles state the dependencies of each target. When targets depend on other targets this creates a partial ordering. When the relationships and recipes are comprehensively and correctly defined this establishes the property of concurrency: there exists a partial order such that order of certain tasks can be re-arranged without affecting the result. Again, this concurrency can be leveraged to build multiple rules simultaneously but the concurrency is a property of the Makefile whether parallelism is employed or not. For DevelopersApril 16, 2025Tired of slow apps? It might be a concurrency vs. parallelism issue. Concurrency switches between tasks, while parallelism runs them simultaneously. Learn when to use each.Have you ever wondered why your computer can run multiple programs at once? Or how your app handles so many user requests simultaneously? There are two powerful concepts in play here: concurrency and parallelism.These terms get mixed up all the time, even by experienced developers. And yes, they both aim to make our programs faster and more efficient, but they work in fundamentally different ways.Think of it like this: concurrency is about dealing with lots of things at once, while parallelism is about doing lots of things at once. Subtle difference, right? But understanding this distinction can completely change how you approach problem-solving in your code.What if I told you that sometimes you need concurrency without parallelism? Or that throwing more CPU cores at a problem isn't always the answer?Let's break down these concepts in simple terms, see how they work in the real world, and figure out which approach makes the most sense for your next project. Ready to untangle this confusion once and for all?Ready to build faster, more efficient apps? Join Index.dev and match with top global companies seeking your concurrency and parallelism expertise. Apply now!What Is Concurrency?Concurrency is about structuring your program to handle multiple tasks that overlap in time. But here's the key: these tasks don't necessarily run at the exact same moment. Instead, the system switches between them, giving each task a little attention before moving to the next.Let's look at that web server example:User 1 requests data from the server.User 2 uploads a file.User 3 retrieves images.Without concurrency, each request would have to wait until the previous one is finished. But with concurrency:The server starts processing User 1’s request.While waiting for the database to respond, it moves on to User 2’s file upload.While waiting for the file to upload, it starts fetching User 3’s images.The CPU keeps switching between these tasks until they’re all complete.This makes the system much faster, especially on single-core processors, where only one task can truly run at a time. But concurrency is also useful in multi-core environments.Here’s a simple example using Python’s asyncio module to run multiple tasks concurrently:import asyncio async def fetch_data():   print("Fetching data...")   await asyncio.sleep(2) # Simulate network delay   print("Data retrieved!") async def upload_file():   print("Uploading file...")   await asyncio.sleep(3) # Simulate file upload   print("File uploaded!") async def main():   await asyncio.gather(fetch_data(), upload_file()) # Run tasks concurrently asyncio.run(main())How it works:fetch_data() and upload_file() are asynchronous functions.Instead of waiting for one to finish before starting the other, asyncio.gather() lets them run concurrently.The program switches between tasks while waiting, making better use of time and system resources.This approach makes applications like web servers, chat apps, and data processing pipelines much more efficient.Different Concurrency ModelsAs applications become more complex, developers have evolved different approaches to handling concurrent tasks. Each model has its own strengths and ideal use cases:1. Cooperative MultitaskingThink of this as a friendly system where tasks voluntarily take turns. Each task runs until it reaches a natural stopping point (like waiting for user input or network data). Then, it hands over control to another task. This makes it easy to manage since the tasks themselves decide when to step aside.Where Is It Used?Lightweight embedded systemsEarly versions of Microsoft Windows (Windows 3.x) and Classic Mac OSCoroutines in languages like Python (asyncio) and KotlinExampleIf you’ve used Python’s asyncio, you’ve worked with cooperative multitasking. Here’s a simple coroutine example:import asyncio async def task1():   print("Task 1 started")   await asyncio.sleep(2)   print("Task 1 finished") async def task2():   print("Task 2 started")   await asyncio.sleep(1)   print("Task 2 finished") async def main():   await asyncio.gather(task1(), task2()) asyncio.run(main())Here, task1 and task2 voluntarily pause (await asyncio.sleep()) so other tasks can run, making the program more efficient.2. Preemptive MultitaskingUnlike cooperative multitasking, this model doesn’t rely on tasks playing nice. Instead, the operating system decides when to pause one task and switch to another. This ensures all tasks get a fair share of CPU time, even if some tasks don’t want to give up control.Where Is It Used?Modern operating systems (Windows, macOS, Linux)Web serversJava threads (Thread class) and Python’s threading moduleExampleA web server needs to handle multiple client requests at the same time. Here’s an example using Python’s threading module:import threading import time def handle_request(client_id):   print(f"Handling request from Client {client_id}")   time.sleep(2)   print(f"Finished handling Client {client_id}") threads = [] for i in range(5):   thread = threading.Thread(target=handle_request, args=(i,))   thread.start()   threads.append(thread) for thread in threads: thread.join()Here, multiple client requests are handled concurrently, ensuring no single request blocks the entire server.3. Event-Driven ConcurrencyImagine a restaurant kitchen where the chef only works when there’s an order. Instead of constantly checking if food is ready, the chef responds to new tasks as they arrive. That’s event-driven concurrency. Tasks are triggered by events rather than running continuously.Where Is It Used?JavaScript (Node.js event loop)Python’s asyncioReal-time chat applicationsExampleHere’s a simple event-driven web server (Node.js):const http = require('http'); const server = http.createServer((req, res) => {   res.writeHead(200, {'Content-Type': 'text/plain'});   res.end('Hello, world!'); }); server.listen(3000, () => {   console.log('Server running at); });Node.js uses an event loop to handle multiple requests efficiently without blocking execution.4. Reactive ProgrammingWhat if you could create a stream of data and react to changes automatically? That’s the core idea of reactive programming. Instead of manually checking for updates, you define how the system should respond to changes dynamically.Where Is It Used?RxJava (for Java applications)Reactor (for Spring WebFlux)ReactiveX (used across multiple languages)Real-time data pipelines and dashboardsExampleHere’s a simple example of reactive programming in Python with RxPY:from rx import from_iterable from rx.operators import map, filter data_stream = from_iterable(range(10)) processed_stream = data_stream.pipe(filter(lambda x: x % 2 == 0), # Keep even numbers   map(lambda x: x * 10) # Multiply by 10) processed_stream.subscribe(lambda x: print(f"Received: {x}"))This program processes a stream of numbers reactively, meaning each number is handled as soon as it arrives rather than waiting for all numbers to be processed first.How Models CompareModelStrengthsWeaknessesBest ForCooperativeSimple, low overheadVulnerable to task greedEmbedded systems, async I/OPreemptiveFair resource sharingSynchronization complexityMulti-core CPU workloadsEvent-DrivenHigh I/O scalabilitySingle point of failureWeb servers, real-time appsReactiveClean data flow managementSteep learning curveReal-time UIs, data streamsUse Cases of ConcurrencyConcurrency is the backbone of how modern apps stay responsive and efficient. It shines in situations where your program needs to handle multiple tasks that spend a lot of time waiting, whether for user input, network responses, or disk operations.Let’s explore how it powers everyday tools and systems.1. Web BrowsersA browser juggles multiple tasks at once, like:Rendering pages while fetching images/scripts.Handling user interactions (scrolling, typing, clicking).Background updates (e.g., auto-refreshing tabs).2. Web ServersThink about a busy website, like an online store during a holiday sale. Thousands of users are requesting pages, adding items to carts, and checking out—all at the same time.Web servers like Apache/Nginx use threads or async I/O to process requests without blocking.Rate limits prevent overload (e.g., browsers cap concurrent connections to avoid DoS attacks).3. Chat AppsSlack, WhatsApp, or Discord rely on concurrency to keep conversations flowing:Processing messages while updating the UI.Sending/receiving data without delays.4. Video GamesVideo games are a perfect example of concurrency in action. While you’re playing, multiple things happen at the same time:Graphics render continuously (so you see the game world).User input is processed (e.g., moving a character, firing a weapon).Physics simulations run (so objects interact realistically).Background music & sound effects play seamlessly.What Is Parallelism?Parallelism means actually executing multiple tasks simultaneously by using multiple processing units—typically different CPU cores or processors. Nothing is being faked or switched between; real work is happening in parallel.The key difference from concurrency? Parallelism requires multiple physical processors or cores to truly run things simultaneously. It's about having enough hands on deck to do multiple things at once.Think about that data processing example:When you need to analyze data, generate reports, and run simulations, parallelism lets you assign each task to a different processor. Core 1 crunches the analysis, Core 2 generates the reports, and Core 3 handles the simulations—all happening genuinely at the same time.Parallelism shines when you can break a problem into completely independent pieces. The less these pieces need to talk to each other, the better parallelism works.Here's a simple example using Python's multiprocessing module:import multiprocessing import time def analyze_data(dataset_id):   print(f"Analyzing dataset {dataset_id}...")   time.sleep(3) # Simulates complex analysis   print(f"Analysis of dataset {dataset_id} complete!")   return f"Results from dataset {dataset_id}" def generate_report(report_type):   print(f"Generating {report_type} report...")   time.sleep(2) # Simulates report generation   print(f"{report_type} report complete!")   return f"{report_type} report data" def run_simulation(simulation_params):   print(f"Running simulation with parameters {simulation_params}...")   time.sleep(4) # Simulates complex simulation   print(f"Simulation {simulation_params} complete!")   return f"Simulation results: {simulation_params}" if __name__ == "__main__":   # Create a pool of processes   with multiprocessing.Pool(processes=3) as pool: # Start all tasks in parallel   analysis_result = pool.apply_async(analyze_data, ("customer_behavior",))   report_result = pool.apply_async(generate_report, ("quarterly",))   simulation_result = pool.apply_async(run_simulation, ("market_growth",))   # Get the results (this will wait until tasks complete)   analysis_data = analysis_result.get()   report_data = report_result.get()   simulation_data = simulation_result.get()   print("All tasks completed!")   print(f"Final results: {analysis_data}, {report_data}, {simulation_data}")How it worksEach task runs in a separate process using different CPU cores.All tasks run simultaneously, reducing the total execution time.Unlike concurrency (which switches between tasks), these tasks actually execute in parallel.This method is ideal for CPU-bound tasks where heavy computation is required.Different Parallelism ModelsParallelism is all about doing multiple things at the same time. Unlike concurrency, which is about managing multiple tasks efficiently, parallelism actually runs tasks simultaneously, utilizing multi-core processors or distributed computing resources. Let's explore some key parallelism models and how they are used in code and hardware.1. Data ParallelismEver wondered how large-scale computations, like image processing or deep learning, happen so fast? That’s data parallelism at work. This model splits large datasets into smaller chunks and processes them simultaneously across multiple processors. Each processor performs the same operation on its respective subset of data.When Is It Used?SIMD (Single Instruction, Multiple Data) operationsParallel array processingMapReduce frameworkReal-World ApplicationsImage and signal processingLarge-scale data analysis (Big Data)Scientific simulations2. Task ParallelismTask parallelism takes a different approach. Instead of dividing data, it breaks the overall job into smaller, independent tasks that run in parallel. Each task performs a different operation, allowing multiple processes to work simultaneously without interfering with each other.When Is It Used?Thread-based parallelism in JavaParallel Tasks in .NETPOSIX threadsReal-World ApplicationsA web server handling API requests, database writes, and logging at once.Rendering a game’s physics, audio, and UI in separate threads.Example Here is an example of parallel implementation of an algorithm in Python with multiprocessing:import multiprocessing def compute_square(num):   return num * num if __name__ == "__main__":   numbers = [1, 2, 3, 4, 5]   with multiprocessing.Pool(processes=4) as pool: results = pool.map(compute_square, numbers)   print(results) # Output: [1, 4, 9, 16, 25]This example divides the task of squaring numbers across multiple CPU cores, speeding up execution.3. Pipeline ParallelismImagine a car assembly line: each stage completes a specific task before passing it on. Pipeline parallelism works the same way. Tasks are divided into stages, and each stage processes a portion of the workload concurrently, improving efficiency.When Is It Used?Unix pipeline commands (cat file.txt | grep 'error' | sort)Image processing pipelinesETL (Extract, Transform, Load) pipelines in data engineeringReal-World ApplicationsVideo and audio processingReal-time data streaming applicationsManufacturing and assembly line automation4. GPU ParallelismGPUs aren’t just for gaming! They’re built for massive parallelism, capable of executing thousands of threads simultaneously. This makes them ideal for tasks requiring heavy computations, like AI and deep learning.When Is It Used?CUDA (Compute Unified Device Architecture) by NVIDIAOpenCL (Open Computing Language)TensorFlow and PyTorch for deep learningReal-World ApplicationsMachine learning and deep learningReal-time graphics renderingHigh-performance scientific computingExampleThe following is a code snippet that illustrates GPU parallelism in Deep Learning (using TensorFlow & GPU acceleration):import tensorflow as tf # Check if GPU is available print("GPU Available:", tf.config.list_physical_devices('GPU')) # Sample Neural Network using GPU Acceleration model = tf.keras.models.Sequential([tf.keras.layers.Dense(128, activation='relu'),   tf.keras.layers.Dense(64, activation='relu'),   tf.keras.layers.Dense(10, activation='softmax')]) model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy']) print("Model ready for GPU training!")This snippet ensures TensorFlow uses the available GPU to accelerate deep learning model training.When to Use Which ModelModelStrengthsWeaknessesBest ForData ParallelismSimple for uniform tasksNeeds identical operationsML training, image processingTask ParallelismFlexible for diverse tasksCoordination complexityWeb servers, real-time appsPipelineEfficient for streaming dataBottlenecks stall flowVideo encoding, ETLGPU ParallelismRaw speed for math-heavy workLimited by GPU memoryDeep learning, simulationsUse Cases of ParallelismParallelism is what makes modern computing insanely fast. From AI training to large-scale simulations, parallel processing is everywhere. Here are some real-world examples:1. Machine Learning TrainingWithout parallelism, training deep learning models would take months or even years. Data parallelism splits datasets into batches and trains them across GPUs.TensorFlow’s data parallelism lets researchers train models on distributed clusters.Real-world example: NVIDIA’s M4 chip processes 38 trillion operations/second for AI tasks like image recognition.2. Video RenderingHave you ever wondered why professional animation studios have rooms full of powerful computers? It's all about parallel rendering:Each frame in an animated movie can require hours of computation.By rendering different frames on different machines simultaneously, studios can produce complex animations in reasonable timeframes.A scene that might take weeks to render on a single computer can be completed overnight when distributed across hundreds of machines.Real-world example: Apple’s M4 chip renders 3D graphics in iPad Pro games by distributing tasks across its 10-core GPU.3. Web CrawlersSearch engines need to constantly index billions of web pages. Parallelism makes this enormous task manageable:Search engine crawlers like Googlebot split up the internet into manageable chunks.Different crawler instances process different websites simultaneously.Real-world example: Aptos Labs’ Block-STM validates 160,000+ blockchain transactions/second by parallelizing smart contract execution.4. Data ProcessingWhen companies analyze massive datasets to derive business insights, parallelism is essential:Frameworks like Apache Spark distribute data processing across clusters.Analyzing logs from millions of users becomes feasible.Real-time analytics for fraud detection or ad targeting.Real-world example: IBM’s Summit supercomputer processes mental health data to predict at-risk children, leveraging parallelism for faster insights.5. Scientific SimulationsSome of the most impressive applications of parallelism occur in scientific computing:Weather forecasting models divide the atmosphere into a 3D grid, with different regions calculated on different processors.Pharmaceutical companies simulate molecular interactions in parallel to discover potential new drugs.Physics researchers model particle collisions by distributing calculations across supercomputer clusters.Real-world example: The University of Chicago’s pSIMS project models climate impacts using parallel supercomputers.Learn More: 15 Best AI Tools for Developers To Improve WorkflowConcurrency vs. ParallelismNow that we've explored both concepts in depth, let's compare concurrency and parallelism to better understand when and how to use each approach.Resource UtilizationConcurrency: Runs multiple tasks within a single core by rapidly switching between them. When Task A is waiting for a database query, the CPU jumps to Task B instead of sitting idle.Parallelism: Uses multiple cores or processors to execute tasks at the same time. Imagine multiple cashiers at different registers, each serving a customer simultaneously.FocusConcurrency: Focuses on structure—it's about how we organize and manage multiple tasks to progress simultaneously.Parallelism: Focuses on execution—it's about physically performing multiple computations at the exact same time.Task ExecutionConcurrency: Creates an illusion of simultaneity through rapid task switching. Tasks are actually taking turns using the CPU, but the switching happens so quickly that it appears simultaneous to users.Parallelism: Offers true simultaneity—different tasks actually execute at the same instant in time on different processors.Context SwitchingConcurrency: Involves frequent context switching, where the CPU rapidly jumps between tasks. While this helps keep things moving, excessive switching can slow things down.Parallelism: Avoids this overhead since each task runs uninterrupted on its own processor. The trade-off is that you need multiple physical processing units, which increases hardware costs.When to Use What?Concurrency is great for I/O-bound tasks:Handling multiple network requestsReading and writing filesManaging user input in interactive applicationsParallelism is best for CPU-bound tasks:Complex mathematical computationsImage and video processingMachine learning and data analysisConcurrency vs. Parallelism: Table ComparisonAspectConcurrencyParallelismDefinitionIt's all about juggling multiple tasks, interleaving their execution. Think of it as rapidly switching between tasks, so they seem to run at the same time.It's about simultaneous execution of multiple tasks. They are actually running at the same time on separate resources.ExecutionAchieved with context switching on a single core or thread.Needs multiple cores or processors to execute tasks simultaneously.FocusManaging multiple tasks and making the most of available resources.Splitting one big task into smaller sub-tasks that can be executed at the same time.Use CaseGreat for I/O-bound tasks. Think handling tons of network requests or file operations.Perfect for CPU-bound tasks. Like heavy-duty data processing or training a machine learning model.Resource NeedsCan run on a single core or thread.Needs multiple cores or threads to make it happen.What You GetImproves responsiveness. Tasks are managed smoothly, making things feel faster.Reduces overall execution time by doing tasks at the same time.ExamplesAsynchronous APIs, chat apps, web servers handling lots of requests.Video rendering, machine learning training, scientific simulations.How Can Concurrency and Parallelism Work TogetherMany modern applications use both approaches together:Web servers use concurrency to manage thousands of connections while using parallelism to process requests across multiple coresGame engines use concurrency to handle input, networking, and audio while using parallelism to maximize physics and rendering performanceDatabase systems use concurrency to handle multiple client connections while using parallelism to execute queries fasterSo, how can this all play out in real-world scenarios? Let’s explore a few examples.Financial Data Processing:Scenario: A stock trading app needs real-time data and complex analysis.Concurrency: Fetch stock prices from APIs without blocking.Parallelism: Run Monte Carlo simulations across cores to predict risks.Result: Traders get instant updates and deep insights.Video Processing:Scenario: A TikTok-like app handles uploads and encoding.Concurrency: Let users upload videos while others are processed.Parallelism: Encode videos in parallel using GPU cores.Result: No lag, no crashes—just smooth uploads and fast playback.Data Scraping:Scenario: A marketing tool scrapes websites and analyzes trends.Concurrency: Fetch 100 websites at once (async I/O).Parallelism: Parse data across cores to spot trends.Result: Faster insights, broader coverage.Explore More: Automate Your Daily Workflow with These 19 Useful Python ScriptsConclusionSo there you have it! Concurrency and parallelism might sound similar, but they solve different problems in different ways.Remember: concurrency is about juggling multiple tasks and making progress on all of them, even with limited resources. Parallelism is about throwing more resources at your problem to get things done faster.Your applications probably need both approaches. When your code is waiting for things (like user input or network responses), concurrency keeps things moving. When your code needs raw processing power, parallelism delivers the speed.The next time your app feels sluggish or unresponsive, ask yourself: Is this a concurrency problem or a parallelism problem? The answer will point you toward the right solution.For Developers:Ready to put your concurrency and parallelism skills to work? Join Index.dev's talent network today and get matched with global companies looking for developers who understand these crucial concepts. Build a global remote career!For Clients: Need developers who truly understand concurrency and parallelism? Index.dev connects you with the top 5% of vetted developers in just 48 hours. Start your 30-day free trial today and build your team with experts who can optimize your applications for maximum efficiency. At first, it may seem that concurrency and parallelism may be referring to the same concepts. However, these terms are actually different.This article explains the differences between concurrency vs parallelism. We also use a practical example to explore the concepts even more and show how using concurrency and parallelism can help speed up the web scraping process.For your convenience, you can also watch this tutorial on concurrent vs parallel processing in a video format or keep reading a blog post.This capability of modern CPUs to pause and resume tasks so fast gives an illusion as if more than one task is running in parallel. However, this is not parallel. This is concurrent.Concurrency can be broadly understood as multi-threading. There are usually many ways of creating concurrent applications, and threading is just one of them. Sometimes, other terms like asynchronous tasks are also used. The difference lies in the implementation and details. However, from a broader point of view, they both mean a set of instructions that can be paused and resumed. There is a programming paradigm called Concurrent Computing. This involves writing code so that it seems like more than one process is being performed simultaneously, while they never actually execute at the same time. This is known as concurrent programming. What is a thread?In broad terms, a thread is the smallest set of tasks that can be handled and managed by the operating system without any dependencies on each other. The actual implementation differs in various operating systems. The way of programmatically creating threads also differs in various programming languages.Python provides a powerful threading module for creating and managing threads. Practical exampleTo understand how concurrency works, let’s solve a practical problem. The issue is to process over 200 pages as fast as possible. Here are the details:Step 1. Go to the Wikipedia page with a list of countries by population and get the links of all the 233 countries listed on this page.Step 2. Go to all these 233 pages and save the HTML locally.Let’s create a function to get all the links. At first, we won’t involve concurrency or parallelism here.import requests from bs4 import BeautifulSoup from urllib.parse import urljoin def get_links(): countries_list = ' United_Nations)' all_links = [] response = requests.get(countries_list) soup = BeautifulSoup(response.text, "lxml") countries_el = soup.select('td .flagicon+ a') for link_el in countries_el: link = link_el.get("href") link = urljoin(countries_list, link) all_links.append(link) return all_linksThis function gets the response from the link and uses BeautifulSoup to extract all the links. Links that we retrieve are relative links. They are converted to absolute links using urljoin. Now let’s write a function that doesn’t use any threading, but sequentially downloads the HTML from all those 233 links.First, let’s create a function to fetch and save a link.def fetch(link): response = requests.get(link) with open(link.split("/")[-1]+".html", "wb") as f: f.write(response.content)This function is simply getting the response of the parameter link and saving it as an HTML file.Finally, let’s call this function in a loop:import time if __name__ == '__main__': links = get_links() print(f"Total pages: {len(links)}") start_time = time.time() # This for loop will be optimized for link in links: fetch(link) duration = time.time() - start_time print(f"Downloaded {len(links)} links in {duration} seconds")With our computer, this took 137.37 seconds. Our objective is to bring this time down.Using concurrency to speed up processesAlthough we can create threads manually, we’ll have to start them manually and call the join method on each thread so that the main program waits for all these threads to complete.The better approach is to use the ThreadPoolExecutor class. This class is part of the concurrent.futures module. The benefit of using this class is that it allows us an easy interface for creating and executing threads. Let’s see how it can be used.First, we need to import ThreadPoolExecutor:from concurrent.futures import ThreadPoolExecutorNow, the for loop written above can be changed to the followingwith ThreadPoolExecutor(max_workers=16) as executor: executor.map(fetch, links)Here, the executor applies the function fetch to every item of links and yields the results. The maximum number of threads is controlled by max_workers argument.The final result is astonishing! All these 233 links were downloaded in 11.23 seconds. It’s a better result than the synchronous version which took around 138 seconds.It’s important to find the sweet spot for the max_worker. On our computer, if the max_worker parameter is changed to 32, the time comes down to 4.6 seconds. Increasing this number further doesn’t improve things much. This sweet spot will differ for every processor for the same code.Note: the print() function isn’t thread-safe. The reason is that print works with a reference to the standard output. The standard output is shared globally. We’ll use print() with a new line character explicitly and an empty end parameter.What is parallelism?In the previous section, we looked at a single processor. However, most processors have more than one core. In some cases, a machine can have more than one processor.One example is parallel computing. This is a type of computation in which multiple processors carry out many processes at the same time. To achieve this parallel processing, specialized programming is needed. This is known as parallel programming, where the code is written to utilize multiple CPU Cores. In this case, more than one process is actually executed in parallel. The following image should help understand parallelism and how it helps in executing multiple tasks.Let’s go back to the previously mentioned practical issue of downloading the HTML from all those 233 links.In Python, parallelism can be achieved by using multitasking. It allows us for a simultaneous execution (download several links at the same time) by using several processors.To write an effective code that can be run on any machine, you would need to know the number of processors available on that machine. Python provides a very useful method, cpu_count(), to get the count of the processor on a machine. This is very helpful to find the exact number of tasks that can be processed in parallel. Note that in the case of a multi-core CPU, each core works as a different CPU.Let’s start with importing the required module:from multiprocessing import Pool, cpu_countNow we can replace the for loop in the synchronous code with this code:with Pool(cpu_count()) as p: p.map(fetch, links)This will create a multiprocessing pool that is equal to the count of the CPU. It means that the limit of multiple tasks being carried out would be determined when the code is actually running.This fetches all 233 links in 18.10 seconds. It’s also noticeably faster than the synchronous version which took around 138 seconds.Difference between concurrency and parallelismHere’s a list of parallelism vs concurrency differences:Concurrency is when multiple tasks can run in overlapping periods. It’s an illusion of multiple tasks running in parallel because of a very fast switching by the CPU. Two tasks can’t run at the same time in a single-core CPU. Parallelism is when tasks actually run in parallel in multiple CPUs.Concurrency is about managing multiple different tasks at the same time, while a parallel program is running multiple instruction sequences at the same time.In Python, concurrency is achieved by using threading, while parallelism is achieved by using multitasking.Concurrency needs only one separate CPU Core, while to achieve parallelism it needs more than one. Concurrent programs are about interruptions, and parallelism is about isolation.Combination of concurrent and parallel programmingThis is often known as Parallel Concurrent execution. The following image can help to understand the combination of parallelism and concurrency.As it’s evident, if multiple CPUs are running separate threads, the process is parallel and concurrent at the same time.Suitable solution for web scrapingIt can be assumed that both parallel and concurrent programming make the web scraping process much faster. However, this should be taken with a pinch of salt. Every project is unique, and the complexity of every project is different. Starting with concurrency first and then looking at parallelism would be a great idea. Combining concurrency and parallelism may help in some cases, but it can make code complex and introduce challenging trace bugs. Once again, you should choose the one that suits your web scraping project best.In addition to optimizing your code, implementing proxy servers is another powerful way to scale your web scraping efforts efficiently. Proxies help to distribute requests across different IP addresses, avoiding rate limits and IP bans that can prevent your scraper from running smoothly – even when your system is fully concurrent or parallelized. Residential, rotating, or datacenter proxies can each play a role depending on your target website and business needs. Additionally, if you’re looking to test proxies before committing to a full subscription, you can always take advantage of high-quality free proxies coming from a reputable provider.You can always forget about complex web scraping processes and choose an advanced public data collection solution – Web Scraper API. Try it for free to decide if it's a suitable option for your case. ConclusionIn this article, we explored concurrency vs parallelism and described what is the difference between concurrency and parallelism. Concurrency can be easily understood as multiple threads or units of work that can be paused and resumed. Parallelism is simply multiple tasks running on multiple cores. Concurrency and parallelism can be achieved using Python threading and multitasking libraries, respectively. Using either concurrent vs parallel execution will improve the performance of the web scraping process significantly.See more comparison articles, such as Go vs Python, by exploring our blog. I believe this answer to be more correct than the existing answers and editing them would have changed their essence. I have tried to link to various sources or wikipedia pages so others can affirm correctness. Concurrency: the property of a system which enables units of the program, algorithm, or problem to be executed out-of-order or in partial order without affecting the final outcome 1 2. A simple example of this is consecutive additions: 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45 Due to the commutative property of addition the order of these can be re-arranged without affecting correctness; the following arrangement will result in the same answer: (1 + 9) + (2 + 8) + (3 + 7) + (4 + 6) + 5 + 0 = 45 Here I have grouped numbers into pairs that will sum to 10, making it easier for me to arrive at the correct answer in my head. Parallel Computing: a type of computation in which many calculations or the execution of processes are carried out simultaneously 3 4. Thus parallel computing leverages the property of concurrency to execute multiple units of the program, algorithm, or problem simultaneously. Continuing with the example of consecutive additions, we can execute different portions of the sum in parallel: Execution unit 1: 0 + 1 + 2 + 3 + 4 = 10 Execution unit 2: 5 + 6 + 7 + 8 + 9 = 35 Then at the end we sum the results from each worker to get 10 + 35 = 45. Again, this parallelism was only possible because consecutive additions have the property of concurrency. Concurrency can be leveraged by more than just parallelism though. Consider pre-emption on a single-core system: over a period of time the system may make progress on multiple running processes without any of them finishing. Indeed, your example of asyncronous I/O is a common example of concurrency that does not require parallelism. Confusion The above is relatively straightforward. I suspect people get confused because the dictionary definitions do not necessarily match what was outlined above: Concurrent: occurring or existing simultaneously or side by side 5. Concurrency: the fact of two or more events or circumstances happening or existing at the same time From searching on google: "define: concurrency". The dictionary defines "concurrency" as a fact of occurrence, whereas the definition in the computing vernacular is a latent property of a program, property, or system. Though related these things are not the same. Personal Recommendations I recommend using the term "parallel" when the simultaneous execution is assured or expected, and to use the term "concurrent" when it is uncertain or irrelevant if simultaneous execution will be employed. I would therefore describe simulating a jet engine on multiple cores as parallel. I would describe Makefiles as an example of concurrency. Makefiles state the dependencies of each target. When targets depend on other targets this creates a partial ordering. When the relationships and recipes are comprehensively and correctly defined this establishes the property of concurrency: there exists a partial order such that order of certain tasks can be re-arranged without affecting the result. Again, this concurrency can be leveraged to build multiple rules simultaneously but the concurrency is a property of the Makefile whether parallelism is employed or not.