HOCHSCHULE
HAMM-LIPPSTADT

Parallel Computing

Prof. Dr. René Krenz-Baath

Name : Pinak Ganatra
Email : pinak.ganatra@stud.hshl.de
Matrikel Nr. : 1210339

Datum : 07.07.2024

1. Introduction

Parallel computing has become increasingly important due to the physical limitations of transistor scaling and the need for improved performance in various computationally intensive applications. With the advent of multi-core processors, parallel programming has become crucial for improving application performance and responsiveness. By utilising multiple cores or processors simultaneously, computationally intensive tasks can be executed more efficiently, leading to significant performance gains. Additionally, parallel programming allows for better utilisation of available hardware resources and enhances the responsiveness of applications by offloading tasks to separate threads, preventing the main UI thread from freezing.

Motivation and Amdahl's Law
Moore's Law, which states that the number of transistors on a chip doubles approximately every 18 months, has been a driving force behind the development of faster processors. However, as we approach the physical limits of transistor scaling, parallel computing becomes crucial for achieving further performance gains.

Amdahl's Law is a fundamental concept that highlights the maximum speedup achievable through parallelization. It states that the sequential portion of a program ultimately limits the maximum speedup, as given by the formula: $S = 1 / (b + (1 - b) / n)$, where 'b' is the sequential fraction, and 'n' is the number of processors.

1.1 Parallel Architectures and Classifications

Parallel computer architectures can be classified in several ways:

I. By control structure:

A. SISD (Single Instruction, Single Data): A single processor executing a single instruction stream on a single data stream.
B. SIMD (Single Instruction, Multiple Data): A single instruction stream executed on multiple data streams simultaneously, such as vector processors, VLIW processors, and array computers.
C. MIMD (Multiple Instruction, Multiple Data): Multiple processors, each with its own instruction stream and data stream. It can be further divided into synchronous MIMD (with a central clock and program memory) and asynchronous MIMD (with local clocks and program memories).
D. SPMD (Single Program, Multiple Data): Running the same program on each processor but with different data.

II. By memory organisation:

A. Shared memory systems: Multiple processors share a common memory space, allowing easy communication and data sharing. This includes UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access) systems.

B. Distributed memory systems: Each processor has its own private memory, and data must be explicitly communicated between processors (e.g., multicomputers, clusters).

III. By interconnection topology:
Interconnection networks play a crucial role in parallel computing architectures by enabling communication and data transfer between processors and memory modules. They can be characterised by various topologies (e.g., fat-tree, mesh, hypercube) and evaluated based on graph-theoretical properties such as node degree, diameter, connectivity, bandwidth, and throughput.

IV. By degree of synchronous execution:
Synchronous systems (all processors operate in lockstep with a single clock) and asynchronous systems (processors operate independently with their own clocks).

1.2 Challenges and Complexities

However, parallel programming introduces several challenges and complexities that must be addressed. Parallel code is inherently more complex than sequential code due to the need for synchronisation, communication between threads, and handling race conditions. Debugging parallel code is also more difficult because of the non-deterministic nature of thread execution and potential for race conditions. Furthermore, achieving linear speedup with an increasing number of cores or processors is not always possible due to factors like communication overhead and sequential portions of the code, as described by Amdahl's Law.

1.3 Parallel Programming in C# and the Task Parallel Library (TPL)

In the context of C#, parallel programming is facilitated by the Task Parallel Library (TPL), introduced in .NET 4.0. The TPL provides a higher-level abstraction for parallelizing code, making it easier to write and maintain parallel programs. Key concepts and components of the TPL include:

I. Task: A lightweight unit of execution that can run asynchronously.
II. Parallel.For and Parallel.ForEach: Constructs for parallelizing loop iterations.
III. Parallel LINQ (PLINQ): Enables parallelization of LINQ queries.
IV. Task.WhenAll and Task.WhenAny: Methods for coordinating multiple tasks.
V. Cancellation: Mechanisms for cancelling long-running tasks.

1.4 Thread Safety and Synchronisation

Parallel programming also necessitates an understanding of thread safety and synchronisation primitives to manage shared resources effectively. These include locks, semaphores, and monitors, which are essential for preventing race conditions and ensuring data integrity in concurrent operations.

2. Design of Parallel Programs

Overview of parallel programming models:

- **A simple generic model:** tasks and channels [Foster]: This refers to a basic model proposed by Foster, where parallel programs are composed of tasks and communication channels between them.
- **Message passing:** This is a parallel programming model where processes communicate by explicitly sending and receiving messages to exchange data and coordinate their activities.
- **Shared memory programming:** In this model, multiple processes or threads share a common memory space, allowing them to communicate and coordinate by reading and writing to shared variables.
- **Dataparallel programming:** This programming model is suitable for problems that can be decomposed into data elements that can be processed independently and in parallel.

2.1 Foster's Model

In his book "Designing and Building Parallel Programs," Foster proposes a model based on tasks that interact with each other through communication channels. The key components of Foster's model are:

2.1.1 Notion Of Task:

A task is a fundamental unit of computation and execution in parallel programming.

I. A parallel computation consists of tasks: A parallel computation or program is composed of multiple tasks that can be executed concurrently or in parallel.
II. Tasks execute concurrently: Tasks are designed to execute concurrently or simultaneously, rather than sequentially, to leverage the parallel processing capabilities of the system.
III. A task encapsulates a sequential program and local memory: Each task encapsulates a sequential program or a set of instructions, as well as its own local memory space. This local memory is private to the task and cannot be directly accessed by other tasks.
IV. A task can perform 4 basic actions: In addition to performing local operations and accessing its local memory, a task can perform four basic actions:
   A. send a message: A task can send a message to another task for communication and coordination.
   B. receive a message: A task can receive messages sent by other tasks.
   C. create a new task: A task can create or spawn new tasks, enabling dynamic parallelism.
   D. terminate: A task can terminate its own execution.
V. Tasks can be mapped to physical processors: Tasks are logical units of computation that can be mapped or assigned to physical processors or cores for execution.
VI. Tasks define a notion of locality: Tasks introduce the concept of locality in parallel programming. Data items stored in a task's local memory are considered "close" or

locally accessible, while data stored in other tasks' memories or shared memory spaces are considered "remote" and may require communication or data transfers to access.

2.1.2 Channel:

A channel is a communication link that connects a task's outport to another task's inport. Channels are used to transfer data and coordinate the execution of tasks.

2.1.3 Communication Semantics:

I. Channels are buffered, meaning that they can store messages until the receiving task is ready to consume them. This allows for asynchronous communication between tasks.
II. Sending a message through a channel is an asynchronous operation, meaning that the sending task can continue execution without waiting for the message to be received.
III. Receiving a message from a channel is a synchronous operation, meaning that the receiving task is blocked until the expected message arrives.

2.1.4 PCAM Details

I. **Partitioning**:

In the partitioning stage, the computation and data are decomposed into small, independent tasks. The goal is to identify opportunities for parallel execution without considering the practical limitations of the target system, such as the number of processors. This stage focuses on breaking down the problem into smaller, manageable units that can be executed concurrently. This fine-grained decomposition ensures both computation and data are divided into manageable pieces.There are two main approaches to partitioning: domain decomposition and functional decomposition. Let's discuss each of them in detail and provide examples.

Domain Decomposition: Domain decomposition, also known as data decomposition, involves partitioning the data associated with the problem into smaller subsets that can be processed independently by different tasks. Each task operates on a portion of the data, and the results are combined to solve the overall problem. Domain decomposition is particularly useful when the problem involves large datasets or when the computation can be naturally divided based on the data.

Example: Consider a problem of applying an image filter to a large image. Using domain decomposition, the image can be partitioned into smaller sub-images or tiles. Each task is assigned a specific sub-image to process independently. The tasks apply the image filter to their respective sub-images concurrently. Once all the tasks complete their processing, the filtered sub-images are combined to form the final filtered image.

In this example, the data (image) is decomposed into smaller subsets (sub-images), and each task operates on its assigned subset independently. This allows for parallel processing of the image, reducing the overall execution time.

Functional Decomposition: Functional decomposition involves partitioning the problem based on the different functions or operations that need to be performed. Each task is responsible for executing a specific function or a set of related functions. The tasks collaborate and communicate with each other to solve the overall problem. Functional decomposition is suitable when the problem can be divided into distinct functional units or when there are dependencies between different operations.

Example: Consider a problem of processing customer orders in an e-commerce system. Using functional decomposition, the problem can be partitioned into different functional tasks such as:

A. Order Validation: This task validates the customer's order details, checks inventory availability, and ensures the correctness of the order.
B. Payment Processing: This task handles the payment processing for the order, including capturing payment details, verifying payment authorization, and updating the payment status.
C. Order Fulfillment: This task manages the fulfillment process, including generating shipping labels, updating inventory, and coordinating with the warehouse for order dispatch.
D. Notification: This task sends notifications to the customer regarding the order status, shipping updates, and any relevant information.

Each functional task can be assigned to a separate processor or executed concurrently. The tasks communicate and coordinate with each other to ensure the smooth processing of customer orders. For example, the Order Validation task may pass the validated order details to the Payment Processing task, which in turn notifies the Order Fulfilment task once the payment is successfully processed.

II. **Communication**:

The communication stage determines the communication patterns and mechanisms required to coordinate the execution of tasks. It involves defining appropriate communication structures, such as channels or shared memory, and algorithms for data exchange and synchronisation between tasks. The communication patterns should ensure that tasks can exchange necessary data and coordinate their execution effectively. Let's explore each aspect of communication in Foster's model in detail.

Channels: In Foster's model, communication between tasks is conceptualised as channels. A channel is a logical link that connects two tasks, allowing one task (producer) to send messages and the other task (consumer) to receive those messages. Channels provide a structured way to represent the flow of data between tasks.

Channel Structure: The channel structure connects tasks that require data (consumers) with tasks that possess that data (producers). It defines the communication patterns and dependencies between tasks. The goal is to establish channels that enable efficient data transfer and coordination between producers and consumers.

Cost of Communication: Introducing channels and communication operations incurs both intellectual and physical costs. The intellectual cost refers to the effort required to define and manage the channels, while the physical cost represents the overhead associated with actually sending messages. Foster's model emphasises the need to avoid unnecessary channels and communication operations to minimize these costs.

Distribution of Communication: Foster's model suggests distributing communication operations over many tasks. This means that instead of having a few tasks handling all the communication, it is better to spread the communication load across multiple tasks. This distribution helps to balance the communication overhead and improve overall performance.

Concurrent Execution of Communication: The model encourages organising communication operations in a way that allows for concurrent execution. By structuring communication patterns appropriately, tasks can communicate and exchange data in parallel, rather than sequentially. This enables better utilisation of parallel resources and can lead to improved performance.

Communication Axes: Foster's model describes four axes that characterise communication patterns:

A. Local ↔ Global:

- Local communication involves a task communicating with only a small set of other tasks. It is characterised by a low density of communication partners.
- Global communication requires each task to communicate with many tasks. It involves a high density of communication partners.

B. Structured ↔ Unstructured:

- Structured communication follows a regular pattern, such as a tree or grid. It exhibits a well-defined and predictable communication structure.
- Unstructured communication may form arbitrary graphs without a clear regular structure. It is more complex and less predictable.

C. Static ↔ Dynamic:

- Static communication patterns do not change during the execution of the parallel program. They are determined at compile-time or initialization.

- Dynamic communication patterns may be determined by data computed at runtime and can vary significantly. They adapt to the changing needs of the program during execution.

D. Synchronous ↔ Asynchronous:

- Synchronous communication requires coordinated execution between producers and consumers. The producer and consumer must cooperate in the data transfer, ensuring that the consumer is ready to receive when the producer sends.
- Asynchronous communication allows a consumer to receive data without the explicit cooperation of the producer. The producer can send data without waiting for the consumer to be ready, and the consumer can retrieve the data later.

Foster's model emphasises the importance of carefully designing communication structures to minimise costs, distribute communication load, enable concurrent execution, and match the specific needs of the parallel program. By following these principles, developers can create efficient and scalable parallel programs that effectively leverage communication to achieve high performance.

III. **Agglomeration**:

In the agglomeration stage, the tasks and communication structures defined in the previous stages are evaluated in terms of performance and implementation costs. If the granularity of tasks is too fine, resulting in high communication overhead or inefficient utilisation of resources, tasks can be combined into larger tasks. Agglomeration aims to find the right balance between parallelism and performance by adjusting the granularity of tasks.

Agglomeration involves answering two main questions:

A. Is it useful to combine, or agglomerate, tasks to reduce the number of tasks?

- Combining tasks can help reduce communication overhead and task creation costs.
- Agglomeration can increase the granularity of tasks, making them more suitable for parallel execution.

B. Is it worthwhile to replicate data and/or computation?

- Replicating data or computation can sometimes be more efficient than communicating data between tasks.
- Replication can help reduce communication costs and improve performance.

Relation to Mapping: The agglomeration phase may still result in a larger number of tasks than the available processors. The final assignment of tasks to processors is deferred to the mapping phase. Agglomeration focuses on optimising the granularity and structure of tasks, while mapping handles the actual allocation of tasks to processors.

Goals of Agglomeration: Three sometimes-conflicting goals guide the decisions made during agglomeration:

A. Reducing communication costs:

- Increasing the granularity of computation and communication can help reduce communication overhead.
- Larger tasks tend to have fewer communication requirements relative to their computational workload.

B. Retaining flexibility:

- Agglomeration should strive to maintain flexibility in terms of scalability and mapping decisions.
- Overly aggressive agglomeration may limit the ability to scale the parallel program or adapt to different parallel architectures.

C. Reducing software engineering costs:

- Agglomeration should consider the impact on software engineering costs.
- Combining tasks or replicating data and computation may simplify the overall parallel program structure and reduce development complexity.

Granularity Considerations: A large number of fine-grained tasks does not necessarily result in an efficient parallel algorithm. Fine-grained tasks often incur significant communication costs and task creation overhead. Increasing the granularity of tasks through agglomeration can help reduce these overheads and improve overall performance.

Computation vs. Communication Trade-off: Agglomeration involves considering the trade-off between computation and communication. In some cases, it may be more efficient for a task to compute a needed quantity locally rather than receiving it from another task where it has already been computed. This decision depends on factors such as the cost of communication, the complexity of the computation, and the available resources.

Overlapping Communication and Computation: Agglomeration can also explore opportunities to overlap communication and computation. By carefully structuring tasks and communication patterns, it may be possible to reduce the number of communication cycles required to distribute computed data. Overlapping communication and computation can help hide communication latency and improve overall performance.

Agglomeration for Non-Concurrent Tasks: Agglomeration is particularly beneficial when analysis of communication requirements reveals that a set of tasks cannot execute concurrently. In such cases, combining the non-concurrent tasks into a single larger task can reduce communication overhead and improve efficiency without sacrificing parallelism.

IV. Mapping:

The mapping stage assigns each task to a processor in the target system. The goal is to maximise processor utilisation and minimise communication costs. Mapping can be specified statically, where tasks are assigned to processors at compile-time, or determined dynamically using load-balancing algorithms that distribute tasks among processors at runtime. Efficient mapping takes into account factors such as task dependencies, communication patterns, and processor capabilities.

A. **Applicability of Mapping:** The mapping problem arises in parallel systems where tasks are not automatically scheduled, such as distributed memory systems or scalable parallel computers. In uniprocessors or shared memory computers with automatic task scheduling, the mapping problem is typically handled by the system itself.

B. **Challenges in Mapping:** Developing general-purpose mapping mechanisms for scalable parallel computers is an ongoing research challenge. The mapping problem is known to be NP-complete, meaning that finding an optimal mapping is computationally intractable for large-scale problems.

C. **Goals of Mapping:** The primary goal of mapping is to minimise the total execution time of the parallel program. Two strategies are employed to achieve this goal:

   Enhancing concurrency:

   a. Tasks that can execute concurrently should be placed on different processors to maximise parallelism.
   b. This allows multiple tasks to be executed simultaneously, reducing overall execution time.

   Increasing locality:

   c. Tasks that communicate frequently should be placed on the same processor to minimise communication overhead.
   d. Placing communicating tasks on the same processor reduces the need for inter-processor communication, improving performance.

D. **Task Scheduling:** Task scheduling algorithms can be used when a functional decomposition yields many tasks with weak locality requirements. In this approach, a centralised or distributed task pool is maintained, where new tasks are placed, and workers (processors) take tasks from the pool for execution.

E. **Managers and Workers:** A common mapping strategy is the manager-worker model:

Manager:

a. A central manager task is responsible for problem allocation and task distribution.
b. The manager maintains a pool of tasks and assigns them to workers based on their availability and load.

Workers:

c. Each worker repeatedly requests a task from the manager, executes it, and then requests another task.
d. Workers can also generate new tasks during execution and send them back to the manager for allocation to other workers. c. Interaction:
e. The manager and workers communicate to coordinate task allocation and execution.
f. Workers request tasks from the manager, and the manager assigns tasks to workers based on load balancing and other criteria.

The manager-worker model allows for dynamic load balancing, as the manager can assign tasks to workers based on their current workload and availability. This helps to ensure that all processors are utilised efficiently and can adapt to varying computational demands.

F. **Mapping Strategies:** There are various mapping strategies that can be employed, depending on the characteristics of the parallel program and the target architecture:

Static mapping:

a. Tasks are assigned to processors at compile-time or before program execution.
b. Static mapping is suitable when the workload and communication patterns are known in advance and do not change during runtime.

Dynamic mapping:

c. Tasks are assigned to processors at runtime based on the current system state and load.
d. Dynamic mapping allows for adaptability and load balancing in response to changing workload and resource availability.

Hybrid mapping:

e. A combination of static and dynamic mapping techniques can be used to balance the benefits of both approaches.
f. Certain tasks may be statically mapped, while others are dynamically assigned based on runtime conditions.

2.2 Model Programming: Message Passing

In the first variant of the message passing programming model communication between tasks or processes occurs through named channels and ports.

I. Communication via named channels and ports: In this variant of the message passing model, communication between tasks or processes happens through named channels and ports. Each task or process has designated input and output ports for sending and receiving messages.
II. Messages sent via channels: Messages are sent through channels, which act as message queues or mailboxes. These channels provide a way to buffer and store messages until they are received by the intended recipient.
III. Outport/inport pairs connected by channels: Each task or process has a set of outports (for sending messages) and inports (for receiving messages). These outport/inport pairs are connected by channels, forming the communication pathways between tasks or processes.
IV. Send is non-blocking, receive is blocking: In this model, the send operation is typically non-blocking, meaning that a task or process can send a message and continue executing without waiting for the message to be received. However, the receive operation is blocking, which means that a task or process will wait (block) until a message is available in the corresponding input channel or port.

2.2.1 Point-to-Point Communication:

I. Blocking vs. Non-blocking Communication:
Blocking send: The sender waits until the receiver has received the message.
Non-blocking send: The sender continues execution immediately after initiating the send operation.
Blocking receive: The receiver waits until a message has been received.
Non-blocking receive: The receiver checks for an available message and returns immediately if none is available.

II. Synchronous vs. Asynchronous Communication:
Synchronous communication: Both the sender and receiver must be available simultaneously for the communication to occur.
Asynchronous communication: The sender can initiate the send operation without the receiver being ready, and the message is buffered until the receiver is ready to receive it.

III. Buffered vs. Unbuffered Communication:
Buffered communication: Messages are stored in a buffer until they can be received or sent.
Unbuffered communication: Messages are directly transferred between the sender and receiver, without intermediate buffering.

2.2.2 Collective Communication:

groups of processes or tasks communicate together.The collective communication operations are as follows:

I. Broadcast: One process sends the same data to all other processes in the group.
II. Gather: Each process sends its data to a single designated process, which collects all the data.
III. Scatter: A single process distributes different data to all other processes in the group.
IV. Reduce: All processes perform an operation (e.g., sum, max, min) on their data, and the result is combined and stored in a single designated process.
V. Barrier: All processes synchronise at a barrier point, ensuring that no process proceeds until all processes have reached the barrier.

2.2.3 Message Passing Interface (MPI):

MPI is a standardised and portable library for message passing parallel programming. MPI provides a set of routines and functions for point-to-point and collective communication operations, enabling the development of parallel applications for distributed memory systems.

I. MPI Communicators: Groups of processes that can communicate with each other.
II. MPI Datatypes: Defining and handling different data types for communication.
III. MPI Collective Operations: Implementations of the collective communication operations mentioned earlier (broadcast, gather, scatter, reduce, barrier).
IV. MPI Topologies: Defining logical process topologies (e.g., cartesian, graph) for efficient communication patterns.
V. MPI Profiling: Tools and techniques for profiling and optimising MPI-based parallel applications.

2.3 Shared Memory Programming Model

The shared memory programming model is fundamental for parallel programming in systems with multiple processors or cores sharing a common memory space. Key aspects include:

I. **Common Address Space**:

   All tasks (threads or processes) have access to a shared memory space, allowing them to read and write to any memory location, regardless of the processor or core.

II. **Asynchronous Read/Write**:

Tasks perform asynchronous read and write operations on the shared memory, enabling concurrent data access and modification. This can lead to race conditions if not properly managed.

III. **Communication via Shared Data Structures**:

Tasks communicate by accessing and modifying shared data structures within the common address space, eliminating the need for explicit message passing.

IV. **Synchronisation**:

Synchronisation primitives like locks (e.g., mutexes) and semaphores are used to control access to shared data, ensuring data consistency and preventing race conditions.

V. **No Data Distribution or Ownership**:

There is no explicit data distribution or ownership; all tasks have equal access to the entire shared memory, simplifying programming by avoiding the need for explicit data communication..

Advantages

A. Easier program development due to direct access and modification of shared data.
B. No need to specify data communication, as data is implicitly shared.

Disadvantages

A. Hard to manage data locality, which can lead to performance issues, especially in NUMA systems.
B. Performance tuning is more complex than in message passing models due to the need for careful management of synchronisation, cache coherence, and memory access patterns.

2.4 Data-Parallel Programming Models

Concurrency through Data Parallelism: Data-parallel programming achieves concurrency by applying the same operation simultaneously to multiple elements of a data structure. Examples include:

I. Adding 2 to all elements of an array.
II. Increasing the salary of all employees with 5 years of service.

These examples show how data parallelism performs computations concurrently on different data elements.

Fine Granularity: Data-parallel models often operate at a fine granularity, treating each operation on a single data element as a separate task. This allows for high concurrency, particularly effective for computations with a high degree of data independence.

Concept of Locality: Locality in data-parallel models is not as explicit as in message-passing or shared memory models. Instead, the compiler or programmer specifies data partitioning and mapping across processing elements.

Advantages of Data-Parallel Programming:

I. Extracts massive parallelism for large data sets, useful in matrix computations and image processing.
II. Simplified code with a single control flow thread, expressing parallelism through data operations.
III. Compiler-automated communication, easing the programming compared to message-passing models.
IV. Provides a shared address space abstraction, hiding memory distribution complexities.
V. Executable on both SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction, Multiple Data) architectures, offering hardware flexibility.

3. Parallel Computing in C# using PLINQ

PLINQ (Parallel LINQ) is a parallel implementation of LINQ (Language Integrated Query) in C#. LINQ is a set of language extensions that allows developers to write expressive and declarative queries over data sources, such as collections or databases. PLINQ extends LINQ by providing a simple and efficient way to parallelize these queries and take advantage of multi-core processors.

With PLINQ, developers can write parallel queries using familiar LINQ syntax and let the framework handle the complexities of parallel execution, such as partitioning the data, distributing the workload across multiple threads, and merging the results. PLINQ automatically analyses the query and determines the optimal degree of parallelism based on factors like the number of available cores and the size of the data.

3.1 PLINQ Essentials for optimal performance

Partitioning: PLINQ automatically partitions data into smaller chunks for parallel processing, but default partitioning may not be optimal. Developers can use the `Partitioner` class to customise partitioning based on data and query characteristics:

I. **Chunk Partitioning:** Best for arrays and lists, ensuring balanced workload distribution.
II. **Range Partitioning:** Ideal for indexed data sources like arrays or lists, assigning specific index ranges to partitions.

III. **Hash Partitioning:** Suitable for unordered data, distributing elements based on their hash codes.
IV. **Custom Partitioning:** Allows developers to define partitioning logic based on domain-specific knowledge.

Preserving Ordering: PLINQ does not guarantee result order by default. Use the `AsOrdered()` method to maintain original ordering, but be aware of the performance overhead.

Limiting Parallelism: PLINQ determines the degree of parallelism based on system resources, but developers can use `WithDegreeOfParallelism()` to limit concurrent tasks. This is useful for I/O-bound operations or when parallelization overhead is high.

Handling Exceptions: Exception handling in parallel code is crucial. Use the `WithCancellation()` method for graceful query cancellation and `WithMergeOptions()` to control exception aggregation and propagation.

Measuring Performance: To optimize PLINQ performance, measure execution time with the `Stopwatch` class and use profiling tools like Visual Studio's CPU Usage tool to identify and address performance bottlenecks.

3.2 Efficiently Parallelize I/O- Intensive FNs with PLINQ

When parallelizing I/O-intensive functions with PLINQ in parallel computing, it is crucial to consider the characteristics of I/O operations and apply strategies for maximum efficiency:

I. **Asynchronous I/O**: Use asynchronous I/O operations with the `async` and `await` keywords in C# to prevent threads from blocking while waiting for I/O completion. PLINQ integrates well with asynchronous programming, enhancing the parallelization of I/O-bound tasks.

II. **Limiting Concurrency**: Control the degree of concurrency to avoid overloading the I/O subsystem using PLINQ's `WithDegreeOfParallelism()` method. This prevents resource contention and ensures optimal performance based on available I/O bandwidth and the nature of I/O operations.

III. **Batching I/O Operations**: Improve performance by batching I/O operations to reduce the overhead of individual requests. Group elements into batches using PLINQ's `Batch()` method, which minimises the number of I/O requests and enhances throughput.

IV. **Caching and Buffering**:Implement caching and buffering to reduce the impact of I/O latency. Use in-memory data structures or distributed caching systems for caching frequently accessed data. Buffering involves accumulating I/O requests and processing them in batches to optimise performance.

V. **Overlapping I/O and Computation**: Enhance performance by overlapping I/O operations with computational tasks. PLINQ can execute other parallel tasks while waiting for I/O completion, utilizing system resources more effectively and reducing I/O latency impact.

3.3 Functional Purity in PLINQ

Functional purity is a key principle in functional programming, crucial for safe and efficient parallel execution, especially in the context of PLINQ (Parallel Language Integrated Query). Functional purity ensures that functions always produce the same output for the same input without any side effects, thus avoiding race conditions, inconsistent results, and unexpected behaviour during parallel processing.

3.3.1 Key Aspects of Functional Purity in PLINQ:

I. **Immutability**:
   A. Functions should not modify shared state or mutable data structures.
   B. They should operate on immutable data or create new immutable results.
   C. Ensures safe access and processing by multiple threads without interference or data corruption.

II. **Side-Effect-Free**:
   A. Functions should not perform actions like modifying global variables, doing I/O operations, or interacting with external systems.
   B. Absence of side effects allows for safe and efficient parallel execution.

III. **Deterministic**:
   A. Functions should always yield the same output for the same input, regardless of execution order or timing.
   B. Guarantees consistent and predictable results in parallel processing.
   C. Allows PLINQ to reorder and parallelize execution without affecting the final outcome.

IV. **Thread-Safe**:
   A. Functions should be executable concurrently by multiple threads without needing explicit synchronisation.
   B. Thread safety is inherent in pure functions since they don't access or modify shared state.
   C. Enables efficient parallel execution without synchronisation overhead.

3.4 Effective Cancellation Techniques for PLINQ Queries

Effective cancellation is essential in managing PLINQ queries, particularly for long-running or resource-intensive operations. It allows for the graceful termination of a PLINQ query before completion, freeing up system resources and preventing unnecessary computations.

CancellationTokenSource and CancellationToken:
In C#, cancellation is implemented using the `CancellationTokenSource` and `CancellationToken` classes. The `CancellationTokenSource` acts as a controller, while the `CancellationToken` represents the cancellation request. To enable cancellation in a PLINQ query, create a `CancellationTokenSource` and pass its associated `CancellationToken` to the query.

WithCancellation Extension Method:
PLINQ provides the `WithCancellation` extension method to associate a CancellationToken with a PLINQ query. By calling `WithCancellation` and passing a CancellationToken, PLINQ monitors the token and aborts the query if cancellation is requested. Example:

```
CancellationTokenSource cts = new CancellationTokenSource();
var result = data.AsParallel().WithCancellation(cts.Token).Select(...).Where(...);
```

Cancellation Propagation:
When a PLINQ query is cancelled, the request propagates through the query pipeline, attempting to stop execution as soon as possible and discarding remaining work. Cancellation is cooperative; individual operations within the query need to support it.

Cancellable Operations:
Ensure cancellation works correctly by using cancellable operations within the PLINQ query. Many PLINQ operators, such as Select, Where, and `ForAll`, support cancellation out of the box by periodically checking the cancellation token. For custom operations or long-running computations, manually check the cancellation token and respond using the `CancellationToken.ThrowIfCancellationRequested` method. Example:

```
var result = data.AsParallel().WithCancellation(cts.Token).Select(item =>
{
   // Long-running computation
   for (int i = 0; i < 1000000; i++)
   {
      cts.Token.ThrowIfCancellationRequested();
      // Perform computation
   }
   return item; });
```

Cancellation Handling:
When a PLINQ query is cancelled, an `OperationCanceledException` is thrown. Handle this exception appropriately to prevent unhandled exceptions and perform necessary cleanup or resource deallocation. Use a `try-catch` block to catch the `OperationCanceledException` and handle cancellation gracefully, performing any required cleanup operations or informing the user of the cancellation.

3.5 Parallel Aggregation with PLINQ

Parallel aggregation with PLINQ allows you to efficiently perform aggregation operations on large datasets by leveraging parallel processing. PLINQ provides a set of extension methods, such as `Sum`, `Count`, `Average`, `Max`, `Min`, and `Aggregate`, that make it easy to perform parallel aggregation.

PLINQ uses a parallel aggregation algorithm that partitions the data, performs local aggregation on each partition, and combines the local aggregates to produce the final result. This process is handled automatically by PLINQ, abstracting away the complexities of thread management and synchronisation.

While parallel aggregation can provide significant performance benefits, it's important to consider factors such as dataset size, aggregation complexity, and system resources. PLINQ automatically adjusts the degree of parallelism, but you can use the `WithDegreeOfParallelism` method for explicit control.

By leveraging parallel aggregation with PLINQ, you can efficiently process large datasets and achieve faster aggregation results in your parallel applications. Example:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = numbers.AsParallel().Sum();
double average = numbers.AsParallel().Average();
int max = numbers.AsParallel().Max();
```

4. Parallel Class in C#

Demystifying the Parallel class is crucial for achieving optimal performance in parallel programming using C#. The Parallel class, introduced in .NET Framework 4.0, provides a simplified way to write parallel code and take advantage of multi-core processors. Here's an explanation and brief summary of how to effectively utilise the Parallel class for optimal performance:

Parallel.Invoke:

The `Parallel.Invoke` method allows you to execute multiple actions concurrently. It takes an array of `Action` delegates as parameters and executes them in parallel. The method waits for all actions to complete before continuing. Example:

```
Parallel.Invoke(
        () => DoWork1(),
        () => DoWork2(),
        () => DoWork3()
);
```

Parallel.For and Parallel.ForEach:

The `Parallel.For` and `Parallel.ForEach` methods provide a way to parallelize loops. They automatically partition the loop iterations and distribute them among available threads for parallel execution.

- `Parallel.For` is used for iterating over a range of integer values.
- `Parallel.ForEach` is used for iterating over a collection or an array.

Example:

```
Parallel.For(0, 100, i =>
{
        // Parallel loop body
        DoWork(i);
});
List<string> items = new List<string> { "A", "B", "C" };
Parallel.ForEach(items, item =>
{
        // Parallel loop body
        ProcessItem(item);
});
```

Cancellation and Stopping Parallel Loops:
The Parallel class supports cancellation and stopping parallel loops using a `CancellationToken`. By passing a cancellation token to the `Parallel.For` or `Parallel.ForEach` methods, you can gracefully cancel the parallel operation. Example:

```
CancellationTokenSource cts = new CancellationTokenSource();
ParallelOptions options = new ParallelOptions { CancellationToken = cts.Token };
try
{
        Parallel.For(0, 100, options, i =>
        {
                // Parallel loop body
                DoWork(i);
        });
}
catch (OperationCanceledException)
{
         // Handle cancellation
}
```

Outer vs Inner Loops:
Outer Loop: The outer loop refers to the top-level loop in a nested loop structure. It controls the overall iteration and is responsible for coordinating the parallel execution of inner loops.
Inner Loop: The inner loop is nested within the outer loop and represents the parallel workload. Each iteration of the inner loop can be executed independently and concurrently.

When using the Parallel class, it's generally recommended to parallelize the outer loop while keeping the inner loop sequential. This approach allows for better load balancing and avoids the overhead of parallelizing small iterations.

Indexed Parallel.ForEach:
The Parallel class provides an indexed version of the Parallel.ForEach method, which allows you to access the index of each element being processed. This can be useful when you need to perform operations that depend on the index or when you want to preserve the order of results. The indexed Parallel.ForEach method takes an additional parameter of type ParallelLoopState, which provides information about the current iteration and allows for controlling the loop execution.

Harnessing Local Values in Parallel Processing
Local values allow you to maintain thread-specific data within parallel loops, avoiding race conditions and ensuring thread safety. Here's an explanation and brief summary of how to effectively utilise local values in parallel processing:

I. **Thread-Local Storage (TLS):** (TLS) is a mechanism that allows each thread to have its own separate storage for data. In C#, you can use the `ThreadLocal<T>` class to create thread-local variables. Each thread has its own instance of the variable, and modifications made by one thread do not affect the value seen by other threads.
II. **Local Accumulators:** are variables used to accumulate results within parallel loops. Each thread maintains its own local accumulator, avoiding the need for synchronisation and reducing contention. After the loop, the local accumulators from all threads are combined to produce the final result.
III. **Partitioning and Local Results:** When working with large datasets, partitioning the data and processing each partition locally can improve performance and reduce memory usage. Each thread operates on its assigned partition and produces local results, which are then combined to obtain the final result.

5. Task Parallelism In C# with PFX

Task Parallelism is a powerful concept in C# that allows you to write concurrent and asynchronous code using the Task Parallel Library (TPL) and the async/await keywords. TPL provides a high-level abstraction over low-level threading mechanisms, making it easier to write parallel and asynchronous code.

Task Class:

The `Task` class represents an asynchronous operation in C#. It encapsulates a unit of work that can be executed concurrently with other tasks. Tasks can be created using the `Task.Run` method or by using the `Task` constructor.

Example:

```
Task task = Task.Run(() =>
{
        // Task code
        DoWork();
});
```

Here, a new task is created using `Task.Run`, and the task code is specified as a lambda expression. The task is executed asynchronously on a separate thread from the thread pool.

Async and Await Keywords:

The `async` and `await` keywords are used to write asynchronous code in C#. The `async` keyword is used to mark a method as asynchronous, and the `await` keyword is used to wait for the completion of an asynchronous operation without blocking the calling thread.

Example:

```
async Task DoWorkAsync()
{
	// Asynchronous code
	await Task.Delay(1000);
	Console.WriteLine("Work completed.");
}
```

The `DoWorkAsync` method is marked as `async`, indicating that it contains asynchronous code. The `await` keyword is used to wait for the completion of the `Task.Delay` operation, which simulates an asynchronous delay of 1 second.

Task Continuation:
Task continuation allows you to specify additional tasks to be executed after the completion of a task. You can use the `ContinueWith` method to chain tasks together and define the order of execution. Example:

```
Task<int> task1 = Task.Run(() => GetResult());
Task<string> task2 = task1.ContinueWith(t => ProcessResult(t.Result));
```

`Here,` `task1` is created to retrieve a result asynchronously. `task2` is defined as a continuation of `task1` using the `ContinueWith` method. It takes the result of `task1` and processes it further.

Task Composition:
Task composition allows you to combine multiple tasks and execute them in parallel or in a specific order. You can use methods like `Task.WhenAll` to wait for the completion of multiple tasks and `Task.WhenAny` to wait for the completion of any one of the specified tasks. Example:

```
Task<int> task1 = Task.Run(() => DoWork1());
Task<string> task2 = Task.Run(() => DoWork2());
Task<bool> task3 = Task.Run(() => DoWork3());

Task[] tasks = { task1, task2, task3 };
await Task.WhenAll(tasks);
```

Here, three tasks (`task1`, `task2`, and `task3`) are created to perform different operations concurrently. The `Task.WhenAll` method is used to wait for the completion of all three tasks before proceeding further.

5.1 Task Execution with TaskCreationOptions

TaskCreationOptions in C# provide a way to optimise task execution by specifying various options when creating tasks. These options allow you to fine-tune the behaviour of tasks and improve performance in different scenarios.

I. **TaskCreationOptions Enum:** TaskCreationOptions is an enumeration that defines various flags you can use when creating tasks. These options can be combined using bitwise OR operations.

II. **Key TaskCreationOptions:**
   a. LongRunning: Use this option for tasks that are expected to run for a long time. It hints the Task Scheduler to create a new thread instead of using the thread pool.
   b. PreferFairness: This option suggests that the Task Scheduler should try to execute tasks in the order they were created. It can be useful for maintaining a sense of fairness in task execution.
   c. AttachedToParent: This option creates a child task that is attached to its parent task. The parent task will not complete until all its child tasks have completed.
   d. DenyChildAttach: This option prevents other tasks from attaching to this task as child tasks.
   e. HideScheduler: This option hides the current Task Scheduler in the created task, which can be useful for avoiding the current synchronisation context.

III. **Combining Options:** You can combine multiple TaskCreationOptions using bitwise OR operations. Example:

```
Task combinedTask = Task.Factory.StartNew(() =>
{
        // Task operation
}, TaskCreationOptions.LongRunning | TaskCreationOptions.PreferFairness);
```

IV. Performance Considerations:

A. Use LongRunning for tasks that are expected to run for more than a few seconds to avoid blocking thread pool threads.
B. PreferFairness can impact performance by potentially delaying task execution, so use it judiciously.
C. AttachedToParent is useful for creating task hierarchies but can complicate task management.
D. DenyChildAttach and HideScheduler can help in specific scenarios but should be used with caution.

5.2 Efficient Task Waiting Techniques

Efficient task waiting techniques are crucial for optimising performance and resource utilisation in parallel programming with C#.

I. **Task.Wait():** The simplest way to wait for a task to complete is using the Wait() method. However, it blocks the calling thread, which can lead to inefficient resource usage.

II. **Task.WaitAll() and Task.WaitAny():** These methods allow waiting for multiple tasks to complete.
    a. WaitAll(): Waits for all specified tasks to complete.
    b. WaitAny(): Waits for any one of the specified tasks to complete.

III. **async/await Pattern:** The async/await pattern provides a non-blocking way to wait for tasks, allowing the calling thread to remain responsive.

IV. **Task.WhenAll() and Task.WhenAny():** These methods return a Task that completes when all or any of the specified tasks complete, allowing for more flexible composition of asynchronous operations.

V. **ContinueWith():** This method allows you to specify a continuation task that will run after the original task completes, without blocking the calling thread.

VI. **Task.Delay() for Timeouts**: You can use Task.Delay() in combination with Task.WhenAny() to implement timeouts without blocking threads.

VII. **Polling with Task.IsCompleted:** For scenarios where you need to periodically check task status without blocking, you can use the IsCompleted property.

5.3 Exception-Handling in Tasks

Exception handling in tasks is a critical aspect of writing robust parallel and asynchronous code in C#.

I. **Basic Exception Handling**: When a task throws an exception, it is captured and stored within the task object. The exception is not immediately propagated to the calling thread.

II. **AggregateException:** When using methods like Task.Wait() or Task.WaitAll(), exceptions are wrapped in an AggregateException, which can contain multiple inner exceptions if multiple tasks failed.

III. **Exception Handling with async/await:** When using async/await, exceptions are automatically unwrapped, and you can catch specific exception types directly.

IV. **Task.Exception Property:** You can check the Exception property of a completed task to access any unhandled exceptions. Example:

```
Task task = Task.Run(() =>
{
        throw new InvalidOperationException("Task failed");
});
task.ContinueWith(t =>
{
        if (t.IsFaulted)
        {
                Console.WriteLine($"Task failed with
        exception:{t.Exception.InnerException.Message}");
        }
});
```

V. **Exception Handling in Task Continuations:** You can specify different continuations for successful completion, faulted state, and cancelled state.

VI. **Observing Unhandled Exceptions:** You can use TaskScheduler.UnobservedTaskException event to handle exceptions that were never observed (e.g., fire-and-forget tasks).

Example:

```
TaskScheduler.UnobservedTaskException += (sender, e) =>
{               Console.WriteLine($"Unobserved task exception:
{e.Exception.InnerException.Message}");
e.SetObserved(); // Prevent the exception from crashing the process
};
```

5.4 Task Continuations

Task Continuations in C# provide a powerful mechanism for specifying additional work to be performed after a task completes. They allow you to chain tasks together and create complex workflows.

I. **Basic Task Continuation:** You can use the ContinueWith method to specify a continuation task that will run after the original task completes. Example:

```
Task<int> task = Task.Run(() => PerformCalculation());
Task continuation = task.ContinueWith(t =>
{
    int result = t.Result;
    Console.WriteLine($"Calculation result: {result}");
});
```

II. **Continuation Options:** The ContinueWith method accepts TaskContinuationOptions to specify when the continuation should run. Example:

```
Task task = Task.Run(() => PerformOperation());
task.ContinueWith(t => Console.WriteLine("Task completed
successfully"),
TaskContinuationOptions.OnlyOnRanToCompletion);
task.ContinueWith(t => Console.WriteLine("Task faulted"),
                  TaskContinuationOptions.OnlyOnFaulted);
task.ContinueWith(t => Console.WriteLine("Task was canceled"),
                  TaskContinuationOptions.OnlyOnCanceled);
```

III. **Multiple Continuations:** A task can have multiple continuations, which can run concurrently or in a specific order. Example:

```
Task<int> task = Task.Run(() => PerformCalculation());
Task     continuation1     =     task.ContinueWith(t     =>
Console.WriteLine($"Result: {t.Result}"));
Task     continuation2     =     task.ContinueWith(t     =>
SaveResultToDatabase(t.Result));
```

IV. **Continuation Chains:** You can create chains of continuations, where each continuation task becomes the antecedent for the next continuation. Example:

```
 Task<int> task = Task.Run(() => PerformCalculation())
.ContinueWith(t => ProcessResult(t.Result))
.ContinueWith(t => SaveResultToDatabase(t.Result));
```

V. **Async Continuations:** You can use async lambdas in continuations to perform asynchronous operations. Example:

```
Task<int> task = Task.Run(() => PerformCalculation());
Task continuation = task.ContinueWith(async t =>
{
    int result = t.Result;
    await SaveResultToDatabaseAsync(result);
});
```

VI. **Handling Exceptions in Continuations:** Continuations can handle exceptions from their antecedent tasks. Example:

```
Task task = Task.Run(() =>
{
    throw new InvalidOperationException("Operation failed");
});
task.ContinueWith(t =>
{
    if (t.IsFaulted)
    {
                             Console.WriteLine($"Task    failed:
{t.Exception.InnerException.Message}");
    }
});
```

VII. **Conditional Continuations:** You can use methods like ContinueWhen to create continuations that depend on multiple tasks. Example:

```
Task<int> task1 = Task.Run(() => PerformCalculation1());
Task<int> task2 = Task.Run(() => PerformCalculation2());
Task continuation = Task.Factory.ContinueWhenAll(new[] { task1,
task2 },
    tasks =>
    {
        int result1 = tasks[0].Result;
        int result2 = tasks[1].Result;
                Console.WriteLine($"Combined result: {result1 +
result2}");
    });
```

5.5 Task Scheduling and Safe UI Updates

Task Scheduling and Safe UI Updates are crucial concepts in C# for maintaining responsiveness and thread safety in applications, especially those with graphical user interfaces.

Explanation:

I. **Task Scheduling:** Task scheduling refers to the process of managing and executing tasks efficiently across available threads or processors. In C#, the Task Parallel Library (TPL) handles most of the scheduling automatically. a. Default TaskScheduler: By default, tasks are scheduled on the ThreadPool using the default TaskScheduler. Example:

```
Task.Run(() => PerformBackgroundWork());
```

b. Custom TaskScheduler: You can create custom TaskSchedulers for specific scheduling needs. Example:

```
public class CustomTaskScheduler : TaskScheduler
{
        protected override void QueueTask(Task task) { /*
Implementation */ }
    protected override bool TryExecuteTaskInline(Task task, bool
taskWasPreviouslyQueued) { /* Implementation */ }
     protected override IEnumerable<Task> GetScheduledTasks() {
/* Implementation */ }

}
```

II. **Safe UI Updates:** UI elements in most frameworks (like WinForms or WPF) are not thread-safe and should only be accessed from the UI thread. a. Using Dispatcher (WPF):

```
await Task.Run(() =>
{
    // Perform background work
    var result = PerformCalculation();

    // Update UI
    Application.Current.Dispatcher.Invoke(() =>
    {
        ResultTextBlock.Text = result.ToString();
    });
});
```

b. Using SynchronizationContext (WinForms):

```
private SynchronizationContext _uiContext;
public Form1()
{
    InitializeComponent();
    _uiContext = SynchronizationContext.Current;
}

private async Task UpdateUIAsync()
{
    await Task.Run(() =>
    {
        // Perform background work
        var result = PerformCalculation();

        // Update UI
        _uiContext.Post(_ =>
        {
            resultLabel.Text = result.ToString();
        }, null);
    });

}
```

III. **ConfigureAwait(false):** When you don't need to return to the original context (e.g., for non-UI operations), you can use ConfigureAwait(false) to improve performance. Example:

```
await SomeAsyncMethod().ConfigureAwait(false);
```

IV. **Task.Run for CPU-bound work:** Use Task.Run to offload CPU-intensive work to a background thread. Example:

```
private async void CalculateButton_Click(object sender,
EventArgs e)
```

```
{
                int     result     =     await     Task.Run(()     =>
PerformHeavyCalculation());
    ResultLabel.Text = result.ToString();

}
```

V. **Progress Reporting:** Use IProgress<T> and Progress<T> for reporting progress from background tasks to the UI. Example:

```
private async void StartButton_Click(object sender, EventArgs e)
{
    var progress = new Progress<int>(value =>
    {
        progressBar.Value = value;
    });

    await Task.Run(() => PerformLongRunningOperation(progress));
}

private      void      PerformLongRunningOperation(IProgress<int>
progress)
{
    for (int i = 0; i <= 100; i++)
    {
        // Do work
        progress.Report(i);
        Thread.Sleep(100);
    }

}
```

5.6 TaskCompletionSource

TaskCompletionSource is a powerful class in C# that allows you to create and control the completion of a Task manually. It's particularly useful for wrapping asynchronous operations that don't naturally return a Task, or for creating custom asynchronous operations.

I. **Basic Usage:** TaskCompletionSource<TResult> allows you to create a Task<TResult> that you can complete, fail, or cancel at will. Example:

```
TaskCompletionSource<int> tcs = new TaskCompletionSource<int>();
Task<int> task = tcs.Task;

// Complete the task
tcs.SetResult(42);

// Use the task
```

```
int result = await task;
Console.WriteLine(result); // Outputs: 42
```

II. **Error Handling:** You can set an exception to indicate that the task failed. Example:

```
TaskCompletionSource<int> tcs = new TaskCompletionSource<int>();
Task<int> task = tcs.Task;

// Fail the task
tcs.SetException(new         InvalidOperationException("Operation
failed"));

try
{
    int result = await task;
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.Message); // Outputs: Operation failed
}
```

III. **Cancellation:** You can cancel the task using a CancellationToken. Example:

```
CancellationTokenSource cts = new CancellationTokenSource();
TaskCompletionSource<int> tcs = new TaskCompletionSource<int>();
Task<int> task = tcs.Task;

// Cancel the task
cts.Cancel();
tcs.SetCanceled(cts.Token);

try
{
    int result = await task;
}
catch (OperationCanceledException)
{
    Console.WriteLine("Task was canceled");
}
```

IV. **Wrapping Asynchronous Operations:** TaskCompletionSource is useful for wrapping callback-based asynchronous operations. Example:

```
public Task<string> ReadFileAsync(string path)
{
    var tcs = new TaskCompletionSource<string>();
        FileStream   fs   =   new   FileStream(path,   FileMode.Open,
FileAccess.Read, FileShare.Read, 4096, true);
```

```
    byte[] bytes = new byte[fs.Length];
    fs.BeginRead(bytes, 0, (int)fs.Length, ar =>
    {
        try
        {
            int bytesRead = fs.EndRead(ar);
            fs.Close();
              string content = Encoding.UTF8.GetString(bytes, 0,
bytesRead);
            tcs.SetResult(content);
        }
        catch (Exception ex)
        {
            tcs.SetException(ex);
        }
    }, null);
    return tcs.Task;
}
```

V. **Timeout Handling:** You can implement custom timeout logic using TaskCompletionSource. Example:

```
public async Task<T> WithTimeout<T>(Task<T> task, TimeSpan
timeout)
{
    var tcs = new TaskCompletionSource<T>();
    using (var cts = new CancellationTokenSource())
    {
        var delayTask = Task.Delay(timeout, cts.Token);
        var resultTask = await Task.WhenAny(task, delayTask);
        if (resultTask == delayTask)
        {
            tcs.SetException(new TimeoutException());
        }
        else
        {
            cts.Cancel();
            tcs.SetResult(await task);
        }
    }
    return await tcs.Task;
}
```

VI. **Combining Multiple Asynchronous Operations:** TaskCompletionSource can be used to create complex asynchronous workflows. Example:

```
public Task<int> CombineResultsAsync(Task<int> task1, Task<int>
task2)
{
    var tcs = new TaskCompletionSource<int>();
    Task.WhenAll(task1, task2).ContinueWith(t =>
    {
        if (t.IsFaulted)
            tcs.SetException(t.Exception.InnerExceptions);
        else if (t.IsCanceled)
            tcs.SetCanceled();
        else
            tcs.SetResult(t.Result[0] + t.Result[1]);
    });
    return tcs.Task;
}
```

6. Conclusion

Parallel computing in C# has proven to be a powerful paradigm for harnessing the capabilities of modern multi-core processors and distributed systems. Throughout this report, we've explored the fundamental concepts of task parallelism and PLINQ, as well as advanced techniques for optimising performance and handling complex scenarios. The Task Parallel Library (TPL) and PLINQ, combined with C#'s async/await pattern, provide developers with high-level abstractions that simplify the creation of efficient, scalable, and responsive applications. We've seen how proper task management, efficient waiting techniques, and careful exception handling are crucial for building robust parallel systems.

As software systems continue to grow in complexity and the demand for processing power increases, the ability to write effective parallel code becomes increasingly valuable. C#'s rich ecosystem of parallel programming tools and techniques positions it as a strong language choice for developers tackling these challenges. While parallel programming introduces new complexities, the benefits in terms of performance and resource utilisation are substantial. By applying the principles and techniques discussed in this report, developers can harness the power of parallel computing to create faster, more efficient, and more scalable applications in C#, meeting the demands of an increasingly parallel computing landscape.

7. Bibliography

- *Mastering Parallel Programming in C# series Part (1-4.9)*. (n.d.). [Video]. YouTube. Retrieved May 26, 2024, from https://www.youtube.com/watch?v=x6HgmdaAUWs&list=PL_talnLgOSYUXGsyaOh96yeL55iEXsgmo&index=2
- *Threading in C# - Part 5 - Parallel Programming*. (n.d.). https://www.albahari.com/threading/part5.aspx#:~:text=PFX%20Concepts,-There%20are%20two&text=This%20is%20called%20data%20parallelism,thread%20perform%20a%20different%20task.
-

AI Tools Used	**What were AI Tools used for?**
Quillbot	Paraphrase, writing assistance
Claude AI	Research Assistance, information extraction from pdfs and images etc. Assist to find code examples.