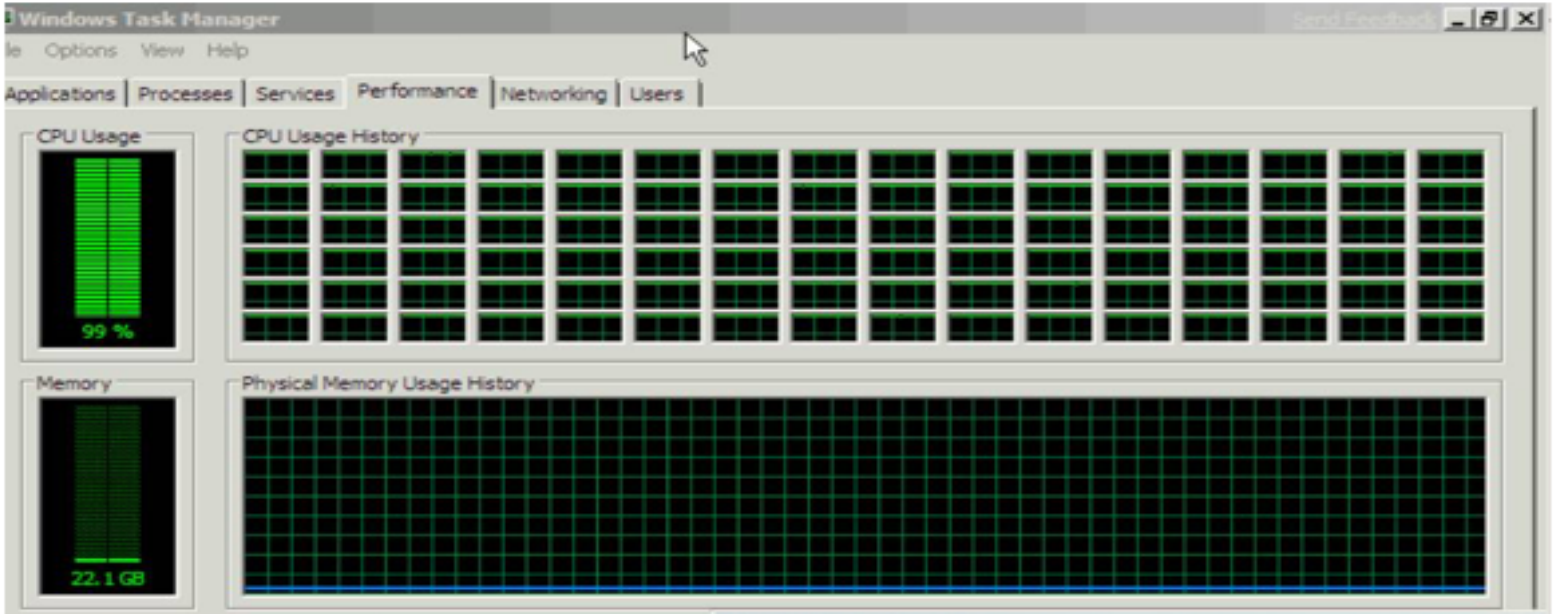
# Why concurrency

# Objectives

- What is the problem with traditional imperative concurrent programming
- Define terminologies
- Immutability and Isolation are your best friends to write concurrent application

# Moore's law - The Concurrency challenge

*Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than it used to be!*



# The need parallelism


MULTITHREADING
THEORY


PRACTICE

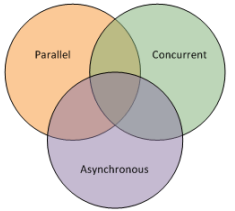
# The issue is Shared of Memory

- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Very difficult to parallelize
- Difficult to maintain and test
- Locking, blocking, call-back hell

# Immutability

- *Purely functional languages do not use mutable data or mutable states, and thus are inherently thread safe – the output value of a function depends entirely on its arguments.*
  - This frees up the programmer from having to manage complex thread synchronization / locks

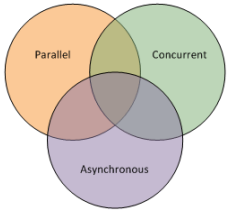
# Immutability




```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }
    public ImmutablePerson UpdateAge(int age)
    {
            return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```

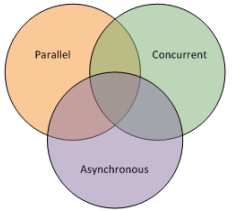
```
type Person = {FirstName:string; LastName:string; Age:int}
```

# Compare And Swap - CAS




```
void DoWithCAS<T>(ref T location, Func<T, T> generator)
                                                    where T : class
    {
        T temp, replace;
        do
        {
            temp = location;
            replace = generator(temp);
        } while (Interlocked.CompareExchange(ref location,
                                                replace, temp) != temp);
    }
```

# Compare And Swap - CAS




```
public class LockFreeStack<T> {
    private class Node {
        public T Data;
        public Node Next;
    }
    private Node head;

    public void Push(T element) {
        Node node = new Node {Data = element};
        DoWithCAS(ref head, h =>
        {
            node.Next = h;
            return node;
        });
    }
```

# The Solution is Immutability and Isolation

IMMUTABILITY +
ISOLATION =

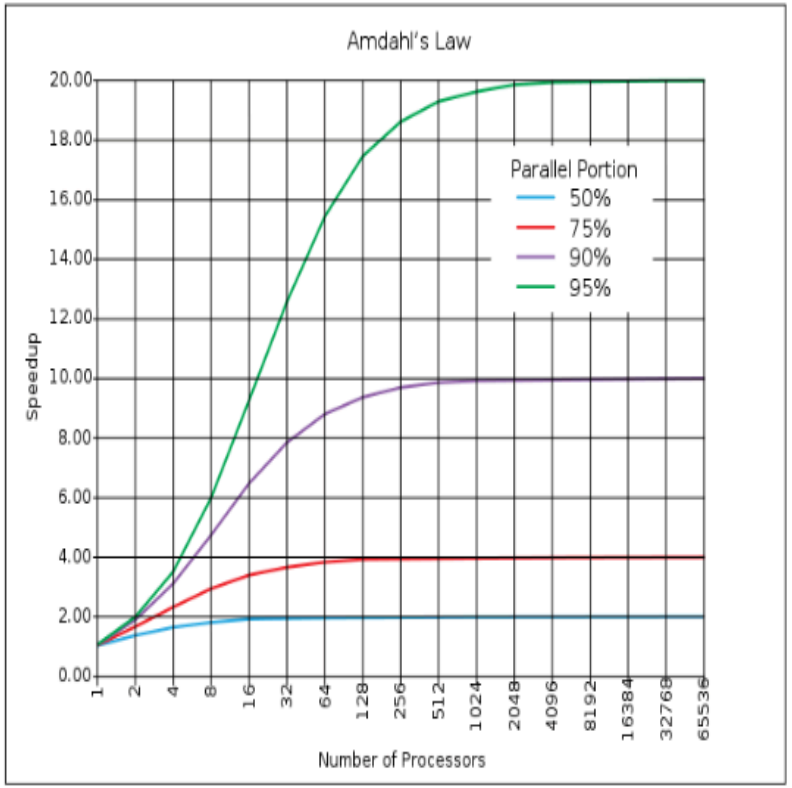
**BEST CONCURRENT PROGRAMMING MODEL**

**Message Passing Concurrency**

# The new reality : Amdahl's law

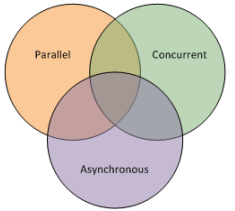
**The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.**

*For example if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors are used*



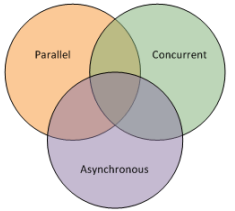
# Concurrent Terminologies

# Definition




- Concurrent
- Multithreaded
- Parallel
- Asynchronous

# Definition

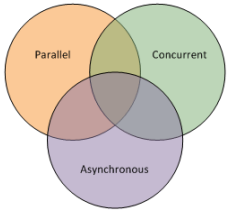



- Concurrent
- Multithreaded
- Parallel
- Asynchronous

Several things happening at once

TPL – Async – Agent - RX

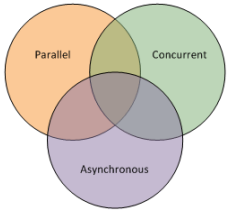
# Definition




- Concurrent
- Multithreaded
- Parallel
- Asynchronous

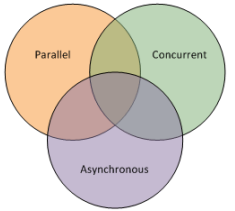
- Several things happening at once
  TPL – Async – Agent - RX
- Multiple execution context

# Definition




| | |
|---|---|
| Concurrent | Several things happening at once<br>TPL – Async – Agent - RX |
| Multithreaded | Multiple execution context |
| Parallel | Multiple simultaneous computations<br>TPL – Async – Agent – RX - GPU |
| Asynchronous | |

# Definition




| | |
|---|---|
| Concurrent | Several things happening at once<br>TPL – Async – Agent - RX |
| Multithreaded | Multiple execution context |
| Parallel | Multiple simultaneous computations<br>TPL – Async – Agent – RX - GPU |
| Asynchronous | Asynchronous<br>TAP – Async-Workflow – Agent |

# Parallel and Concurrent programming

- **Asynchronous** programming
- **Parallel** programming
- **Concurrent** programming
- **Reactive** programming

# Parallel and Concurrent programming

**Asynchronous** programming

**Parallel** programming

**Concurrent** programming

**Reactive** programming

Avoid blocking of threads for I/O operations.
Programs don't complete immediately and can perform meaningful work in the meantime

***Asynchronous workflows***

# Parallel and Concurrent programming

- **Asynchronous** programming
- **Parallel** programming
- **Concurrent** programming
- **Reactive** programming

Run CPU-intensive tasks on multi-core faster.
Multiple threads executing simultaneously in order to speed up the program execution

***Task Parallel Library (TPL)***

# Parallel and Concurrent programming

- **Asynchronous** programming
- **Parallel** programming
- **Concurrent** programming
- **Reactive** programming

Make interactive programs more scalable and responsive. Programs with multiple threads of execution, each typically executing different code

***F# Agents & TPL Dataflo & Reactive Extensions***

# Parallel and Concurrent programming

**Asynchronous** programming

**Parallel** programming

**Concurrent** programming

**Reactive** programming

User-interfaces with lots of event handling.

Reactive programming is about writing applications in such a way that they respond to events as they occur in real time

***Asynchronous workflows***

***Reactive Extensions***


*The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.*

*-- Edsger Dijkstra*