

## Table of Contents

1. [Speedup und Effizienz](#)
2. [Amdahl und Gustavson](#)
3. [Mutual Exclusion](#)
4. [Mutex und Semaphore](#)
5. [Concurrency und Parallelism](#)
6. [Future, Promise, Continuation](#)
7. [Asynchore Programmierung, Future, Promise, Continuation](#)

## Beschreibung von Speedup und Effizienz

### Speedup

**T1..** Zeit mit einer CPU

**TP..** Zeit mit mehreren CPUs

$$S_p = \frac{T_1}{TP}$$

Wie schnell ist ein paralleler Algorithmus im Vergleich zu einem sequentiellen.

### Effizienz

**SP..** Speedup

**TP..** Anzahl der Prozessoren

$$E_P = \frac{S_P}{P}$$

Wie gut werden die verfügbaren Prozessoren genutzt.

## Speedup und Effizienz Amdahl und Gustavson

### Amdahl's Law

Amdahls Law berechnet den theoretischen Speedup der bei der Ausführung einer fixen Arbeitsbelastung erwartet werden kann, wenn die Anzahl der Prozessoren erhöht wird.

- Annahme: Ein Teil der von einem seriellen Programm ausgeführt wird, ist parallelisierbar
- Annahme: Unser Programm lässt sich teilweise parallel implementieren
- Sei  $\sigma$  der Anteil von  $T_1$ , der dem sequentiellen Code entspricht.

### Formeln:

$$S_P \leq \frac{T_1}{T_P} = \frac{1}{\sigma + \frac{(1 - \sigma)}{P}}$$

$$E_P \leq \frac{S_P}{P} = \frac{1}{\sigma(P - 1) + 1}$$

$$\lim_{P \rightarrow \infty} S_P \leq \frac{1}{\sigma}$$

$$\lim_{P \rightarrow \infty} E_P = 0$$

## Beispiel

### Example

Ten people want to paint ten rooms. One room is twice as big. What is the speedup?

We have ten processors ( $P = 10$ ) and 11 units of work. One eleventh of the program is serial.

$$S_P \leq \frac{1}{\sigma + \frac{1-\sigma}{P}} = \frac{1}{\frac{1}{11} + \frac{1-\frac{1}{11}}{10}} = \frac{11}{2} = 5.5$$

3. A program consists of 50% serial code. What is the speedup using Gustafson's Law for  $P = 4$  processors? How about Amdahl's Law?

**Solution:** According to Gustafson's Law, we have:

$$S = s + (1 - s) \cdot P = 0.5 + 0.5 \cdot 4 = 2.5$$

How about Amdahl's Law? In this case we would have

$$S \leq \frac{1}{0.5 + \frac{0.5}{4}} = \frac{1}{0.625} = 1.6$$

5. We are given two variants of the same program:

- In *Program A*, 80% of the code can be parallelized
- In *Program B*, 90% of the code can be parallelized

Assume *Program A* runs on a six-core machine ( $P = 6$ ) while *Program B* runs on a four-core machine ( $P = 4$ ). What is the speedup achieved in each case? Which factor has the bigger impact: increasing the proportion of code that can be parallelized or increasing the number of processors?

**Solution:**

According to Amdahl's Law we get:

$$S_P \leq \frac{1}{\sigma + \frac{1-\sigma}{P}} = \frac{1}{0.2 + \frac{0.8}{6}} = 3.00 \quad (\text{Program A, } P = 6)$$

$$S_P \leq \frac{1}{\sigma + \frac{1-\sigma}{P}} = \frac{1}{0.1 + \frac{0.9}{4}} = 3.08 \quad (\text{Program B, } P = 4)$$

According to Gustafson's Law, we get:

$$S_P = s + q \cdot P = 0.2 + 0.8 \cdot 6 = 5.0 \quad (\text{Program A})$$

$$S_P = s + q \cdot P = 0.1 + 0.9 \cdot 4 = 3.7 \quad (\text{Program B})$$

## Gustafson's Law

Gustafson's Law berechnet ebenfalls den theoretischen Speedup, aber mit einem mehr optimistischen (realistischen) Model.

1. Es gibt Programme die mehr Speedup erreichen wie Amdahl vorhersagt
2. Die Problemgröße wächst und Computer werden immer besser
3. Der Arbeit für den parallelen Teil wächst meist schneller als der serielle Teil.

**Zusammengefasst:** Wenn die Problemgröße wächst, der serielle Anteil schrumpft, wird der Speedup größer. Gustafson's Law hält, solange Punkt 3. zutrifft.

**Formeln:**

**s..** Anteil der seriellen Arbeit

**q..** Anteil der parallelen Arbeit

**P..** Anzahl der Prozessoren

$$S_P = s + q * P = 1 + (P - 1) * q$$

7. In Gustafson's Law, what happens if  $P \rightarrow \infty$ ?

**Solution:** The speedup will also tend to go to infinity  $S_P \rightarrow \infty$ .

**Beispiel:**

3. A program consists of 50% serial code. What is the speedup using Gustafson's Law for  $P = 4$  processors? How about Amdahl's Law?

**Solution:** According to Gustafson's Law, we have:

$$S = s + (1 - s) \cdot P = 0.5 + 0.5 \cdot 4 = 2.5$$

How about Amdahl's Law? In this case we would have

$$S \leq \frac{1}{0.5 + \frac{0.5}{4}} = \frac{1}{0.625} = 1.6$$

5. We are given two variants of the same program:

- In *Program A*, 80% of the code can be parallelized
- In *Program B*, 90% of the code can be parallelized

Assume *Program A* runs on a six-core machine ( $P = 6$ ) while *Program B* runs on a four-core machine ( $P = 4$ ). What is the speedup achieved in each case? Which factor has the bigger impact: increasing the proportion of code that can be parallelized or increasing the number of processors?

**Solution:**

According to Amdahl's Law we get:

$$S_P \leq \frac{1}{s + \frac{1-s}{P}} = \frac{1}{0.2 + \frac{0.8}{6}} = 3.00 \quad (\text{Program A, } P = 6)$$

$$S_P \leq \frac{1}{s + \frac{1-s}{P}} = \frac{1}{0.1 + \frac{0.9}{4}} = 3.08 \quad (\text{Program B, } P = 4)$$

According to Gustafson's Law, we get:

$$S_P = s + q \cdot P = 0.2 + 0.8 \cdot 6 = 5.0 \quad (\text{Program A})$$

$$S_P = s + q \cdot P = 0.1 + 0.9 \cdot 4 = 3.7 \quad (\text{Program B})$$

## Mutual Exclusion

### = (Wechselseitiger Ausschluss)

Mindestens eine Ressource muss in einem nicht gemeinsam nutzbaren Modus gehalten werden, das heißt, nur ein Prozess kann die Ressource zu einem bestimmten Zeitpunkt nutzen.

- Höchstens ein Prozess der Code eines Kritischen Abschnitts ausführt, in dem er eine Sperre (Lock) erwirbt
- Egal wie der Prozess sie erworben hat, er wird eventuell terminieren
- Die Lösung des Mutual Exclusion Problems entspricht der Implementierung eines lock objects (Speerobjektes)

## Mutex und Sempaphore

### Mutex

Mutex = Synchronisationsmechanismus, der gegenseitigen Ausschluss beim Zugriff auf kritische Abschnitte gewährleistet.

### Eigenschaften:

- Mutex hat zwei Hauptoperationen: acquire() (Sperren), release() (Freigeben)
- Kann nur von dem Thread freigegeben werden der ihn auch gesperrt hat.
- Mutex kann gesperrt(locked) oder entsperrt(unlocked) sein
- Lock Counter gibt an wie oft der Mutex Rekursiv gesperrt wurde
- **Lock Rekursion vermeiden!!!!!!**

**Verwendung:** Schutz vor Kritischen Abschnitten, wo nur ein Thread gleichzeitig Zugriff haben darf

### Semaphore

Semaphore = Komplexerer Synchronisationsmechanismus, der auf einem Zähler basiert.

### Eigenschaften:

- Startet mit einer initialen Anzahl von Tokens (Zählerstand)
- hat ebenfalls zwei Hauptoperationen
- acquire() verbraucht einen Token (Zähler --1)
- release() stellt wieder einen Token zu Verfügung (Zähler ++1)
- Wenn keine Tokens mehr zur Verfügung stehen, müssen die Threads warten.
- Kann von jedem Thread freigegeben werden,

**Verwendung** Typisch für die Begrenzung der Anzahl von gleichzeitigen Zugriffen auf eine Ressource

### Binäre Semaphore

- Spezialfall
- Zähler kann nur Wert 0 und 1 annehmen
- Ähnlich wie Mutex, kann aber von jedem Thread freigegeben werden
- Ist die einfachste Form einer Semaphore

## Concurrency and Parallelism

### Concurrency (Nebenläufigkeit)

Beschäftigt sich damit mehrere Dinge gleichzeitig zu behandeln (Struktur)

- Beschreibt die Fähigkeit, mehrere Aufgaben gleichzeitig zu behandeln (virtuell parallel)
- Ausführung verläuft verschachtelt, es ist immer nur ein Prozess aktiv
- Alles wird auf einer einzigen CPU ausgeführt
- Fokussiert sich auf die Struktur und Organisation des Programmes

**Ziel:** Programm in unabhängig voneinander ausführbare Teile zu zerlegen

### Parallelism (Parallelität)

Beschäftigt sich damit mehrere Dinge gleichzeitig auszuführen (Ausführung)

- Spezialfall der Concurrency
- Aufgaben werden tatsächlich gleichzeitig ausgeführt

- Benötigt mehrere CPUs
- Fokussiert sich auf die gleichzeitige Ausführung
- Physische parallel Ausführung von Prozessen

## Asynchrone Programmierung Future Promise Continuation

Das asynchrone Programmiermodell ermöglicht es, dass lang andauernde oder blockierende Aufgaben (wie z. B. Netzanfragen) parallel ausgeführt werden, ohne die Ausführung des Hauptprogramms zu unterbrechen. Dies geschieht durch die Trennung von Initiierung und Abschluss einer Aufgabe.

Eine asynchrone Funktion startet die Arbeit in einem anderen Thread und gibt sofort ein Objekt zurück, das die asynchrone Operation repräsentiert - das sogenannte Future/Promise.

### Future/Promise

Ein Future/Promise ist ein Platzhalter für ein Ergebnis, das noch nicht bekannt ist, meist weil seine Berechnung noch nicht abgeschlossen ist. Es repräsentiert das Ergebnis eines asynchronen Aufrufs.

### Future/Promise in C#

In C# wird dies durch `async`, `await` und `Task` implementiert. Der Platzhalter Promise/Future, der das zukünftige Ergebnis darstellt, wird durch einen `Task` oder `Task<T>` repräsentiert.

```
public async Task<string> GetDataAsync() {
    // long execution time
}
```

### Continuation

Eine Asynchrone aufgabe welche von einer vorangehenden Aufgabe aufgerufen wird, wenn diese fertig ist.

```
//waitforfivesecondsthenprintthe number 42
Task.Delay(5000).GetAwaiter().OnCompleted(()=> Console.WriteLine(42));

//or
Task.Delay(5000).ContinueWith(ant =>Console.WriteLine (42));
```

```
primeNumberTask.ContinueWith(antecedent =>
{
    int result=antecedent.Result;
    Console.WriteLine(result);
});
```

**Die wichtigsten Konzepte sind:**

- `async` markiert eine Methode als asynchron
- `Task` (oder `Task<T>` für Methoden, die einen Wert zurückgeben) repräsentiert die asynchrone Operation und fungiert als Platzhalterobjekt (Future)
- `await` wird verwendet, um auf das Ergebnis einer asynchronen Methode zu warten, ohne den aktuellen Thread zu blockieren

## Verwendung von await in C#

Das `await`-Schlüsselwort hat folgende Eigenschaften:

- Es wird innerhalb einer asynchronen Methode verwendet, um auf den Abschluss einer anderen asynchronen Methode zu warten
- Wenn eine asynchrone Methode einen Await-Ausdruck oder eine Anweisung erreicht, wird die Methode angehalten, und die Kontrolle wird an den Aufrufer zurückgegeben
- Die Methode wird fortgesetzt, nachdem die erwartete asynchrone Methode abgeschlossen ist
- Wenn eine asynchrone Methode keinen Await-Ausdruck enthält, wird sie synchron ausgeführt (blockiert)

Hier ein Beispiel:

```
// Während CallDependencyAsync() awaited wird,
// kann der Aufrufer von DoSomethingAsync() andere Aufgaben ausführen,
// da er die Kontrolle zurückerhält
public async Task<int> DoSomethingAsync() {
    var result = await CallDependencyAsync();
    return result + 1;
}
```

## Thread Safe Unbounded Queue

```
class ProducerConsumerQueue :IDisposable {
    SemaphoreSlim sem = new SemaphoreSlim(0, int.MaxValue); //signal worker thread
    Thread worker;
    object locker = new object(); //explicit sync object
    Queue<object> jobs = new Queue<object>();
    bool terminate = false; //signal termination

    public ProducerConsumerQueue() {
        worker = new Thread(Work);
        worker.IsBackground = true;
        worker.Start();
    }

    public void EnqueueJob(object job){
        lock (locker){ // Thread-safe job addition
            jobs.Enqueue(job); //add new job to the job queue
        }
        sem.Release(); //signal that new work is available -> Wake up Worker if
```

```

blocked
}

public void Dispose(){
    terminate = true;
    signal.Release();
    worker.Join(); //terminate work
}
void Work() {
    while (!terminate) {
        object job = null;
        lock(locker) { // Thread-safe job retrieval
            if(jobs.Count > 0){
                job = jobs.Dequeue(); //get next job
            }
        }

        if(job != null){ //execute job or wait
            Console.WriteLine("Job"+job);
        }
        else {
            sem.Wait(); // No jobs available, wait for signal from EnqueueJob ->
Blocked
        }
    }
}
}
}

```