Amdahl's Gesetz ist zu pessimistisch

- weil es von einem Problem fixer Größe ausgeht
- Mit mehr Prozessoren können größere Probleme gelöst werden können
- σ ist nicht konstant bei steigender Prozessorzahl

Gustafson's Law

Gustafson's Gesetz besagt, dass bei zunehmender Problemgröße Speedup und Effizienz gesteigert werden können, vorausgesetzt der sequentiell abzuarbeitende Programmteil wächst langsamer als der parallelisierbare Teil.

Der entscheidende Unterschied zu Amdahl ist, dass der parallele Anteil mit der Anzahl der Prozessoren wächst. Der sequentielle Teil wirkt hier nicht beschränkend, da er mit zunehmendem N unbedeutender wird. Geht N gegen unendlich, so wächst der SpeedUp linear mit der Anzahl der Prozessoren N.

Superlinear Speedup

Ein Speedup von $S_n > n$ ist möglich

- Caching Effekte
- Ineffiziente sequentielle Algorithmen
- Unnatürliche sequentielle Algorithmen

Concurrency ≠ Parallelism

Concurrency processes are only virtually parallel, execution is teethed, just one process is executed at a time, can be done with just one CPU

Parallelism processes are physically parallel, multiple processes are executed at a time, multiple cores or CPUs necessary

Multithreading

Risiko

„*Deadlocks, Starvation, Race Conditions*“

- **Safety** - shared data might be corrupted
- **Liveness** - threads might "starve" if not properly coordinated
- **Non-determinism** - the same program run twice may lead to different results
- **Run-time Overhead** - thread creation, context switching, synchronization takes time

Lösungen

- **Critical Sections** - define "dangerous" areas, code fragments that use shared resources, ensure that critical sections are atomic
- **Mutual Exclusion** - only one process at a time may

enter a critical section, all other processes must wait, requires locks, semaphores, monitors, …

C# Threading

Warum kein „*Abort*“?

Exception, unbekannter Abbruchspunkt, Ressourcen möglicherweise nicht freigegeben

Lock

- protecting critical sections, locking particular objects, Locking is very fast
- not known beyond process boundaries

Task

Unterscheid Run vs Factory.StartNew

Run Task.Factory.StartNew(A, CancellationToken.None, TaskCreationOptions.DenyChildAttach, TaskScheduler.Default);

StartNew Task.Factory.StartNew(A, CancellationToken.None, TaskCreationOptions.None, TaskScheduler.Current);

Parallel

TPL Reduce

```
static double IntegrateTPL(int n, double a, double b) {
 object locker = new object();
 double sum = 0.0;
 double w = (b − a) / n;
 var rangePartitioner = Partitioner.Create(0, n);
 Parallel.ForEach(rangePartitioner,
  () => 0.0,
  (range, state, partialResult) => {
   for (int i = range.Item1; i < range.Item2; i++) {
    partialResult += w * F(a + w * (i + 0.5));
   }
   return partialResult;
  },
  partialResult => {
   lock(locker) sum += partialResult;
  });
 return sum;
}
```

OMP Reduce

```
double integrateOMP(int n, double a, double b) {
  double sum = 0.0;
  double w = (b − a) / n;
  #pragma omp parallel for reduction(+ : sum)
  for (int i = 0; i < n; ++i) {
    sum += w * f(a + w * (i + 0.5));
  }
  return sum;
}
```

OMP

OpenMP Scheduling

```
#pragma omp parallel for [schedule(static)] [num_threads(THREADS)]
for (i = 0; i < N; i++)
```

size –optionaler Parameter

Static Hier werden den Threads Iterationen zugeteilt

bevor sie die Schleifendurchläufe ausführen. Die Iterationen werden standardmäßig zwischen den Threads aufgeteilt, wenn nicht der Parameter „chunk“ angegeben wird.

Dynamic Hier werden den Threads eine bestimmte Anzahl an Iterationen(chunk) zugeteilt (default = 1). Hat ein Thread eine Iteration erledigt, holt es sich eine neue Iteration von den noch über gebliebenen Iterationen.

Guided Ähnlich wie bei Dynamic, nur wird hier anfangs eine große Anzahl an Iterationen zugeteilt. Diese Anzahl verringert sich exponential bis sie eine definierte Size erreicht hat.

Runtime Schedule Schema wird durch die Umgebungsvariable OMP_SCHEDULE festgelegt.

TPL Queues

Globale Queue FIFO, globale Queue in die alle Threads außer den Worker-Threads des Thread-Pools Tasks stellen

Worker Queue LIFO, beinhaltet von Worker erzeugte Tasks, wenn leer „work stealing“ zuerst global task queue, dann local von anderen workern, „work stealing → FIFO)

Max Worker Beispiel

```
class MaxWorkerThread {
 private readonly int threads;
 public MaxWorkerThread(int threads) {
  this.threads = threads;
 }
 public int getMax(int[] values) {
  int max = int.MinValue;
  var partitioner = Partitioner.Create(0,
   values.Length,
   Math.Ceiling(values.Length / threads));
  var ts =
   partitioner
   .GetDynamicPartitions()
   .Select(s => new Thread(() => {
    for (int i = s.Item1; i < s.Item2; i++)
     if (values[i] > max)
      Interlocked.Exchange(ref max, values[i]);
   }))
   .ToList();
  ts.ForEach(t => t.Start());
```

OMP ohne Reduce

```
int scalar(int argc, _TCHAR* argv[]) {
 int result = 0;
 #pragma omp parallel default(shared)
 {
  #pragma omp for
   for (int i = 0; i < 5; i++) {
    #pragma omp critical
     { result += v1[i] * v2[i]; }
   }
  return result;
 }
```



```
 ts.ForEach(t => t.Join());
 return max;
}
```

Autoreset Event

```
AutoResetEvent ar = new AutoResetEvent(false);
public void Run() {
 Task task = Task.Factory.StartNew(() => {
  //process
   ar.Set(); });
 ar.WaitOne();
}
```

Queue Beispiel

```
BlockingCollection<object> bc = new BlockingCollection<object>();
public WorkerAsync8C() {
 Task.Factory.StartNew(() => {
  while (!bc.IsCompleted)
   if (bc.TryTake(out object item))
    Console.WriteLine("ran_job_" + item.ToString());
 });
}
public void ProcessJobAsync(object job) => bc.Add(job);
```

Semaphore

```
private int _counter = 0;
private Semaphore _semaphore = new Semaphore(3, 3);
public void DoSomething() {
 _semaphore.WaitOne();
 _counter++; //max 3 weil max 3
 _semaphore.Release();
}
```


Performance

Speedup $S_n = \frac{T_1}{T_n}$

Efficiency $E_n = \frac{S_n}{n}$

Amdahl's Law

$$E_n = \leq \frac{\frac{S_n}{n}}{\frac{\sigma+\frac{(1-\sigma)}{n}}{n}} = \frac{1}{\sigma(n-1)+1}$$

Grundaussage *Die Leistung eines Systems bei der Hinzunahme von n Prozessoren ist geringer als die n-fache Leistung eines Einzelprozessors. Bei einer Zahl von n Prozessoren bringt der $(n-1)$-te hinzugenommene Prozessor noch einen stärkeren Leistungszuwachs, als der n-te Prozessor.*

Die möglichen Erhöhung der Prozessorzahl eines Systems führt nicht immer zu einer deutlichen Erhöhung der Leistung eines System. Mindestens die wesentlichen Teile eines Programmes (die am häufigsten genutzten Teile) müssen parallelisiert werden. Mit der Prozessorzahl muß auch die Größe des Problems ansteigen, damit eine steigende parallelisierung dann auch mit steigender absoluter Leistung korrespondiert. Wesentlich für eine hohe Effizienz ist eine gleichmäßige Verteilung der Aufgaben auf die Prozessoren.