kend, 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, hard to perform more complex signalling

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 num_threads(THREADS)
for (i = 0; i < N; i++)...
```



```
#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)

OMP ohne Reduce

```
int _tmain(int argc, _TCHAR* argv[]) {
int v1[] = {1, 2, 3, 4, 5};
int v2[] = {6, 7, 8, 9, 10};
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]; }
  }
}
cout << "result:␣" << result << endl;
cin.get();
return 0;
}
```

Worker Beispiel

```
public class Woker {
private readonly int numThreads;
private readonly object lockObj = new object();
public Woker(int numThreads) {
this.numThreads = numThreads;
}
public int GetMax(int[] values) {
var threads = new Thread[numThreads];
int max = Int32.MinValue;
int index = -1;
```


Queue Beispiel

WIP

```
for (int i = 0; i < numThreads; i++) {
threads[i] = new Thread(() => {
int cur = interlocked.increment(ref index);
while (cur < values.length) {
if (values[cur] > max) {
lock(lockObj) {
if (values[cur] > max)
max = values[cur];
}
}
cur = interlocked.increment(ref index);
}
});
threads[i].Start();
}
foreach(Thread t in threads) {
t.Join();
}
return max;
}
}
```

Performance

$T_1 \ldots$ runtime of a singleprocessor program

$T_n \ldots$ runtime of a multiprocessor program

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

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

Amdahl's Law

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

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 beschrän-