Paralleles Rechnen

(Architektur verteilter Systeme)

von
Thomas Offermann
Philipp Tommek
Dominik Pich

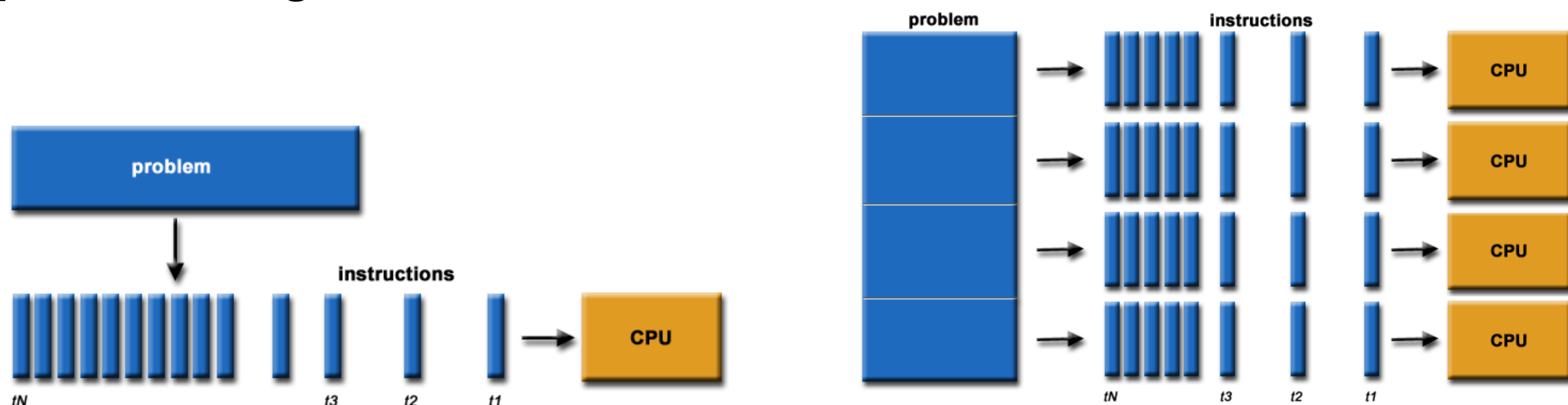
Gliederung

- Motivation
  - Anwendungsgebiete
  - Warum paralleles Rechnen
- Flynn's Klassifikation
- Theorie: Parallel Programmieren
  - Inkl. Grenzen & Kosten
- Arten der parallelen Programmierung

Motivation

Was ist paralleles Rechnen?

- beim seriellen Rechnen wird ein Problem in diskrete Aufgaben zerteilt, welche anschließend mit einer CPU nacheinander abgearbeitet werden.
- Beim parallelen Rechnen wird ein Problem in diskrete Aufgaben zerlegt, welche anschließend mit mehreren CPU's parallel abgearbeitet werden.



Motivation II

- Parallelität kann realisiert werden durch:
  - Einen Computer mit mehreren CPU's
  - Mehreren Computern welche durch ein Netzwerk miteinander in Verbindung stehen
  - Ein Mix aus beiden (siehe Bsp. Praktikum)

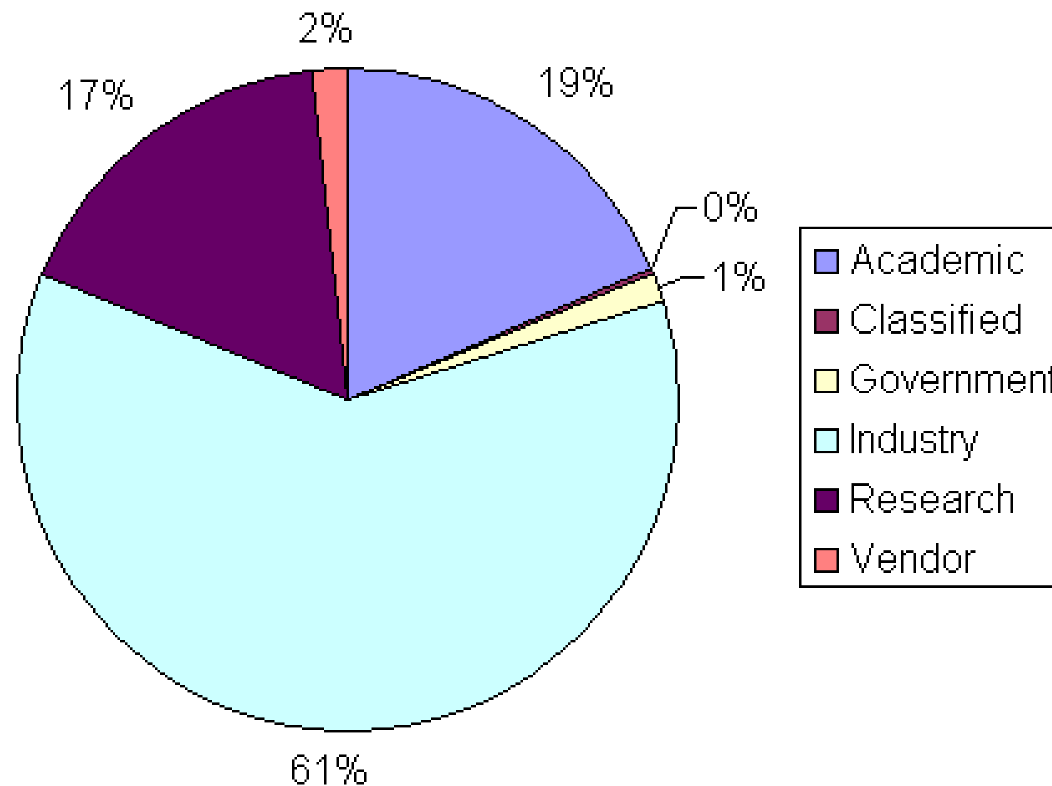
- Parallelität wird verwendet wenn:
  - das Problem so zerlegt werden kann, dass die Unterteile parallel verarbeitet werden können
    - Ergebnisse in Echtzeit vorliegen müssen
    - durch eine parallele Verarbeitung eine signifikante Zeitersparnis bringt

Anwendungsgebiete

Das Universum ist parallel!

- Alles was im Universum passiert, verläuft parallel
  - Galaxie (Planeten, Asteroiden)
  - Geographie (tektonischen Platten)
  - Wetter (Stürme, Fluten)
  - Industrie (Erzeigen Fahrzeugen oder Maschinen)
- Es wird bereits angewendet bei:
  - Datenbanken (z.B. Data mining)
  - Websuchmaschinen (crowler)
  - der Wissenschaft (z.B. Mathematik, Ingenieur, Medizin)
  - ...

Überblick derzeitige Anwendungsgebiete



Quelle: Lawrence Livermore National Laboratory
(https://computing.llnl.gov/tutorials/parallel_comp)

Warum paralleles Rechnen

- Zeit- u. Kostenersparnis
  - Je mehr Ressourcen verwendet werden, um so schneller wird ein Ergebnis (unter Vorbehalt) erzielt
- Komplexe Problemstellung
  - Sehr komplexe Problemstellungen können nur parallel bearbeitet werden (z.B. Windkanal)
- Parallelität wird verlangt
  - Lange Berechnungen laufen im Hintergrund während die Kommunikation aufrecht erhalten werden muss
- Dezentrale Rechenressourcen
  - Rechenleistung über das Netzwerk teilen, wenn nicht benötigt
- Grenzen von seriellen Rechnern
  - Geschwindigkeit (Grenze: Lichtgeschwindigkeit)
  - Große (Grenze: Atomare Prozessoren (Theorie))

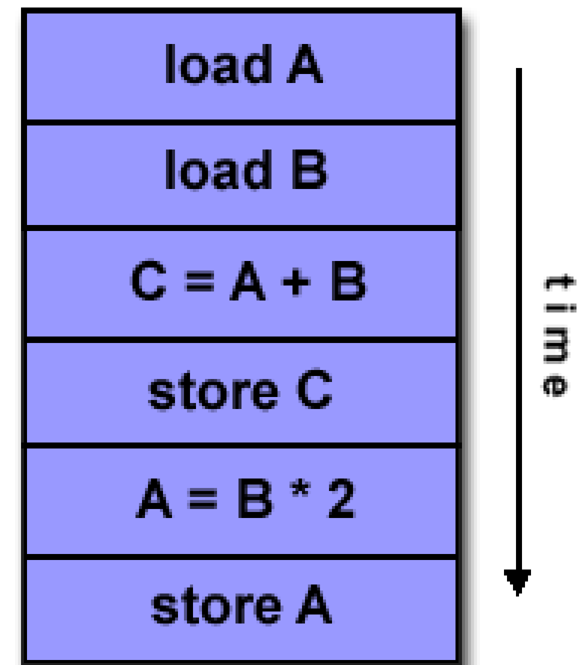
Flynn's Klassifikation

- Michael J. Flynn publizierte 1972 eine Unterteilung von Rechnerarchitekturen
- Dabei werden die Architekturen nach der Anzahl der Befehls- und Datenströme unterteilt

S I S D	S I M D
Single Instruction, Single Data	Single Instruction, Multiple Data
M I S D	**M I M D**
Multiple Instruction, Single Data	Multiple Instruction, Multiple Data

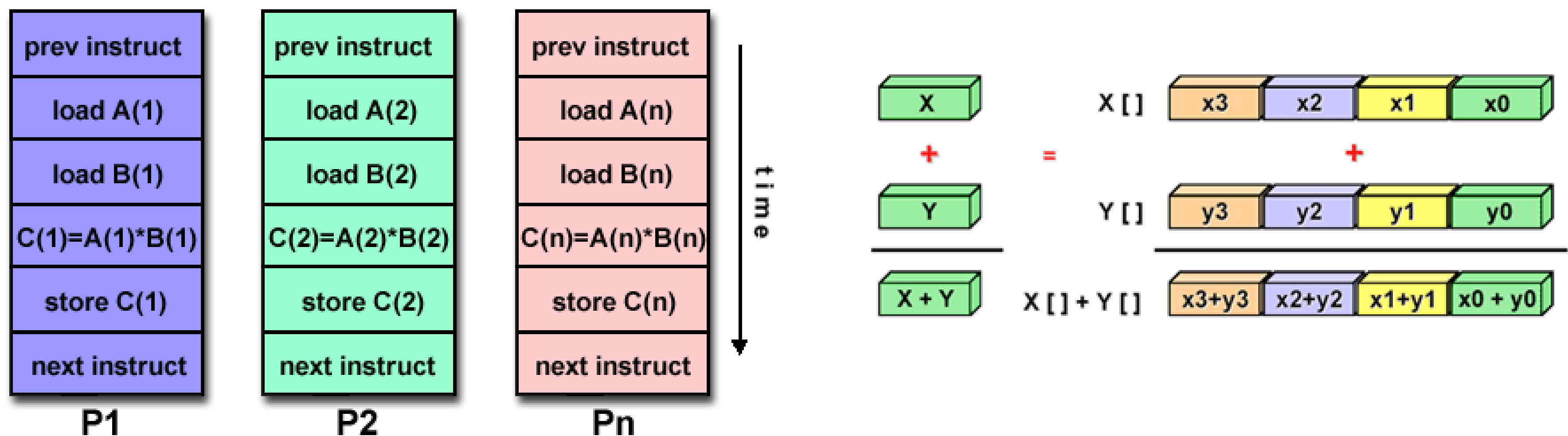
Single Instruction Single Data (SISD)

- Ein serieller (nicht paralleler) Rechner (von Neumann Architektur)
- Der Prozessor kann nur eine Anweisung gleichzeitig verarbeiten
- Der Prozessor kann immer nur einen Wert verändern
- Anwendung muss deterministisch sein
- Älteste und am weiten verbreitete Rechnerarchitektur



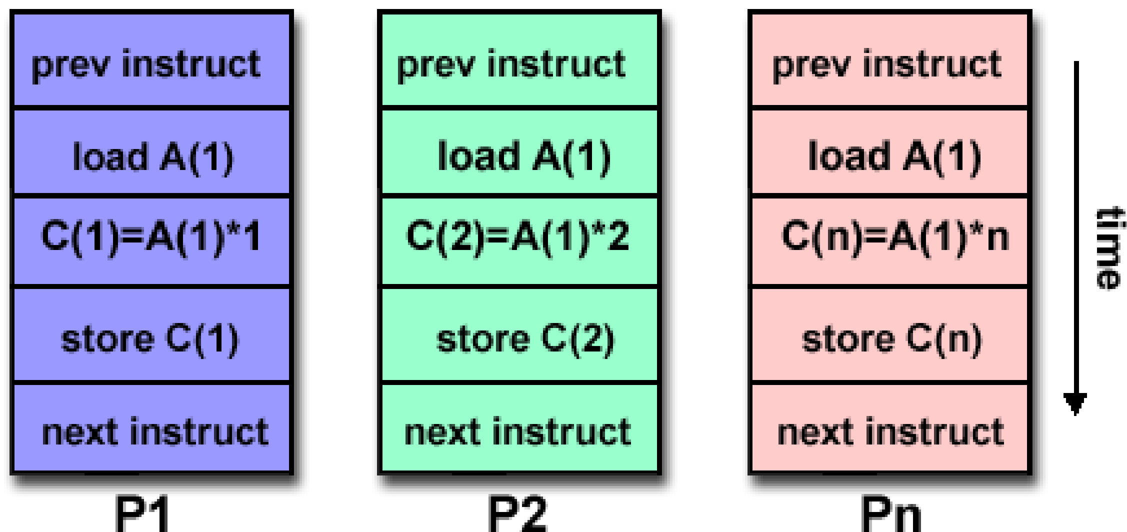
Single Instruction Multiple Data (SIMD)

- Alle Prozessoren bekommen die selben Anweisungen zur selben Zeit
- Jeder Prozessor kann mit anderen Daten arbeiten
- z.B. moderne Computer mit Grafikprozessoren (GPU's)



Multiple Instruction Single Data (MISD)

- Ein Datenstrom für alle Prozessoren
- Prozessor arbeitet mit eigenen Anweisungen
- Jeder Prozessor arbeitet mit den selben Daten im eigenen unabhängigen Speicherbereich

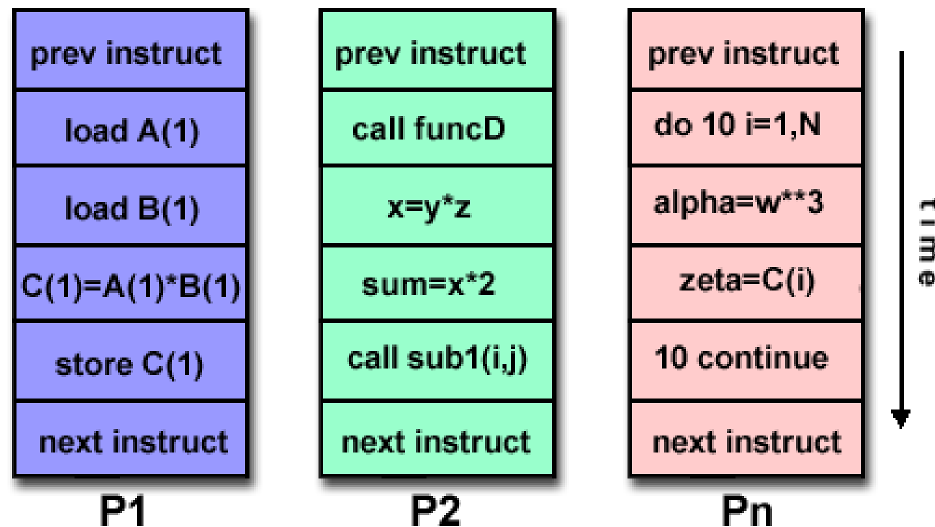


z.B. Bei der Kryptographie!

Multiple Instruction Multiple Data (MIMD)

- Eigener Datenstrom für jeden Prozessor
- Eigene Anweisungen für jeden Prozessor
- Die Verarbeitung kann
  - Deterministisch/Nicht-Deterministisch
  - Synchron/Asynchron

  sein



Parallel Programmieren

- Automatisch
  - Hierbei durchforstet der Compiler den Quellcode um die Anwendung zu parallelisieren
  - ...häufig werden nur Schleifen parallelisiert
- Manuell
  - Durch den Programmierer werden Flags gesetzt, welche dem Compiler zeigen was parallelisiert werden kann
  - … kann in Verbindung mit der automatischer Variante verwendet werden

Parallel Programmieren

Das Problem muss verstanden werden!

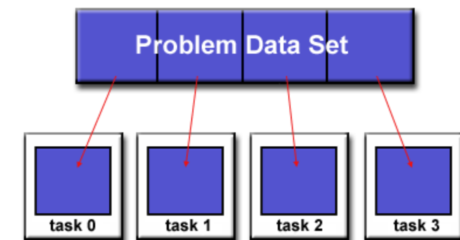
- es kann nicht alles parallelisiert werden
  - z.B. Fibonaccizahlen mit folgender Formel berechnen

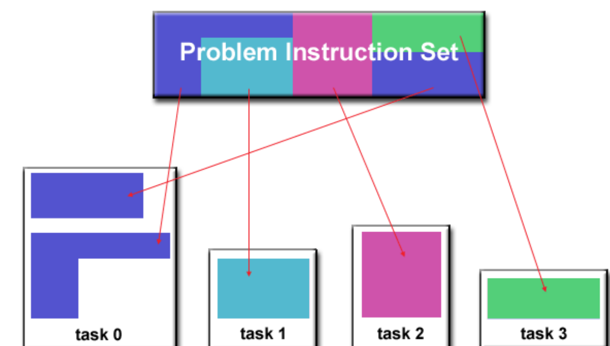
$$F(k + 2) = F(k + 1) + F(k)$$

- nur diskrete unabhängige Probleme können parallelisiert werden

Aufteilung

- Das Problem muss in diskrete „chunks“ untergliedert werden, welche auf mehreren Tasks verarbeitet werden können
  - das wird ***decomposition*** oder ***partitioning*** genannt
  - Es gibt zwei Arten der Aufteilung
    - 1. Domain Dekomposition
      - (Jeder Task arbeitet auf seinen Daten)
    - 2. funktionale Dekomposition
      - (Jeder Task wird abhängig von seiner Aufgabe zerlegt) z.B.
        - Ökosysteme
        - Audiosignalfilter





Kommunikation

Ob und wie oft kommuniziert werden muss ist von dem Problem abhängig

- Bei unabhängigen Daten und unabhängigen Aufgaben, bedarf es keiner Kommunikation
  - z.B. aus einem Bild ein Negativ errechnen (alle Schwarzen Pixel werden weiß und umgekehrt)
- Bei abhängigen Daten bzw. abhängigen Aufgaben bedarf es Kommunikation
  - z.B. Windkanalberechnung (Kommunikation an den Schnittstellen der Sektoren)

Faktoren der Kommunikation

Faktoren die beachtet werden sollten:

- Kosten der Kommunikation
  - Zwei Prozessoren die mit einander kommunizieren, arbeiten nicht
  - Ein Prozessor, der auf eine Nachricht wartet arbeitet nicht
- Latenz und Bandbreite
  - Latenz ist die Zeit welche für das Versenden benötigt wird (Ping)
  - Datendurchsatz (häufig ist es sinnvoller große Datenpakete zu verschicken)
- Sichtbarkeit der Kommunikation
  - Bei dem Message Passing Modell ist die Kommunikation leicht zu verfolgen
  - Bei dem Data Parallel Modell insbesondere bei der dezentralen Speicherverwaltung ist es dem Programmierer u.U. Nicht möglich die Kommunikation der Tasks zu verfolgen

Faktoren der Kommunikation II

- Synchrone vs. Asynchrone Kommunikation
  - Synchrone Kommunikation bedarf Vorbereitung (Empfänger muss auf den Empfang eingestellt sein) => kann von Programmierer oder auf einer tieferen Eben vom System realisiert werden
  - Asynchrone Kommunikation erlaubt das verschicken von Nachrichten ohne das Warten auf andere.

    (Task1 sendet Ergebnis an Task2 und setzt seine Arbeit direkt fort. Unabhängig ob Task2 schon die Daten für die weitere Berechnung hat oder nicht)
- Art der Kommunikation
  - Punkt zu Punkt Kommunikation
  - Gemeinsame Kommunikation
- Overhead und Komplexität
  - Durch die Kommunikation beanspruchte Rechenkapazität / Zeit

Synchronisation

- Barrier (Barriere) Blocken
  - Alle Tasks arbeiten, bis zu einer gewissen Stelle, dann kommunizieren Sie untereinander
- Lock / semaphore (blockierendes Signal)
  - Arbeiten auf dem gleichen Datensatz
  - Nur ein Task kann die Daten bearbeiten
  - Der erste Task, der die Daten blockt, kann Sie bearbeiten
  - Die anderen Tasks können die Daten markieren und wenn der erste Task diese frei gibt bearbeiten.
- Synchroner Nachrichtenversand
  - Betrifft alle Tasks die kommunizieren wollen
  - Bedarf Voraussetzungen der Kommunikation (erwartet den Empfang)

Weitere Aspekte der parallelen Programmierung

- Data Dependency
  - Ergebnisse aus einen Task werden von einem anderen für die weitere Berechnung benötigt
- Load Balancing
  - Alle Prozessoren sollen im Idealfall permanent ausgelastet sein
- Granularity
  - Zeitspanne bis zur nächsten Kommunikation
    - Fine-grain Parallism (kleine Blöcke => Overhead)
    - Coarse-grain Parallism (größe Blöcke => Wartezeit)

Grenzen und Kosten der parallelen Programmierung

- **Amdahls Gesetz:**
  - Die Geschwindigkeitssteigerung steht in Abhängigkeit zur Anzahl der parallelisierbaren Tasks (P)

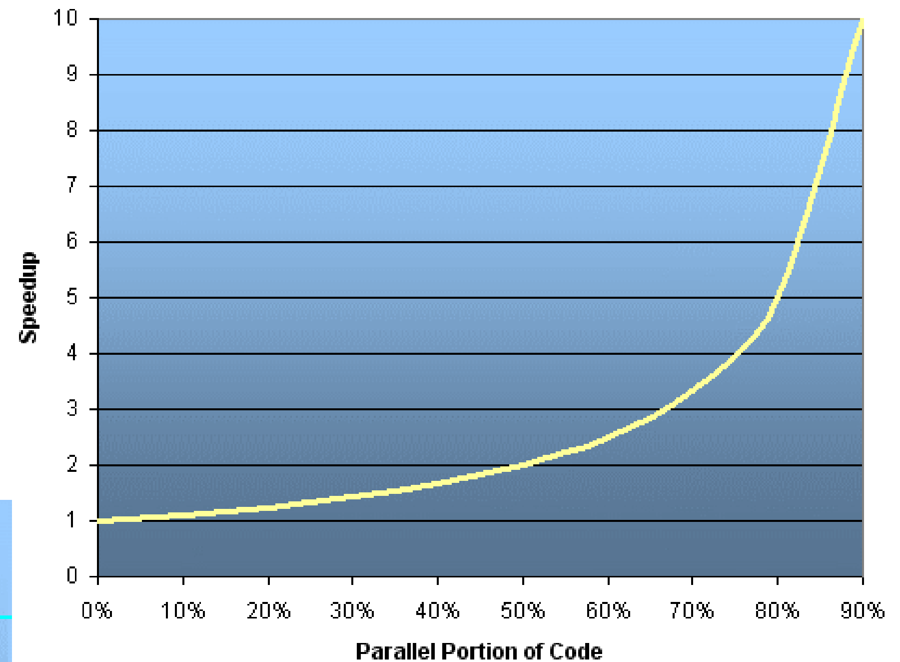
    [Speedup = 1 / (1-P)]

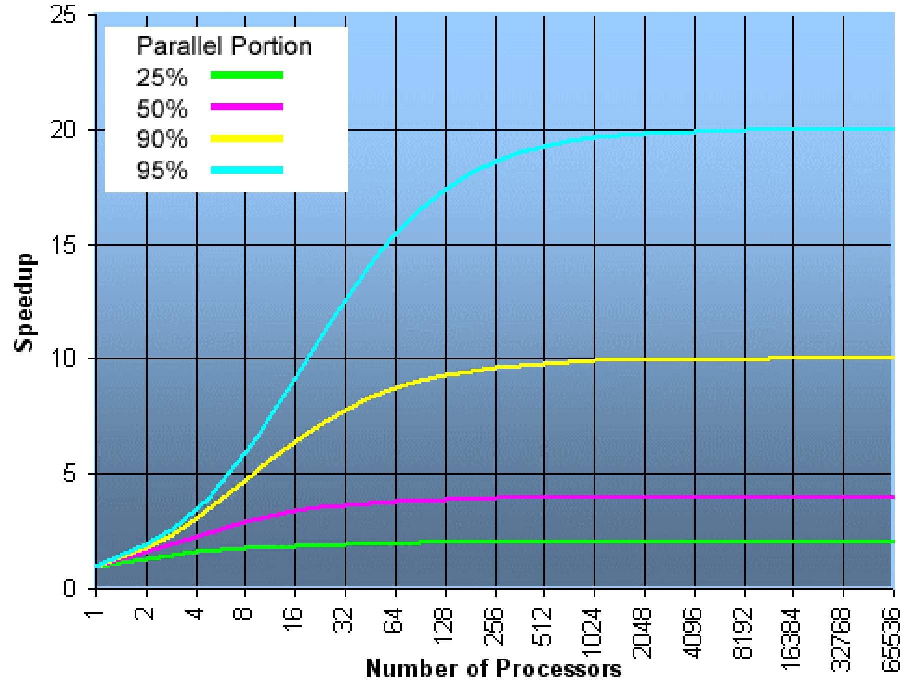
  - Erweiterung: Durch die Anzahl der Prozessoren (N) und der nicht parallelisierbaren Prozesse (S)

    [Speedup = 1/(P/N+S)]

Bsp. Performancesteigerung

Performancesteigerung durch die Anzahl an parallelisierbaren Prozessen





Performancesteigerung durch die prozentuale Anzahl an parallelisierbaren Prozessen unter Berücksichtigen der Anzahl der verfügbaren Prozessoren

Grenzen und Kosten der parallelen Programmierung

- Komplexität
  - Erhöht die Entwicklungszeit der Software
  - Kosten und Arbeitssteigerung in allen Phasen der Entwicklung
- Portierbarkeit
  - Eingeschränkt möglich dank vorhandenen APIs
  - Hardware spielt eine entscheidende Rolle
- Ressourcen
  - Alle Prozessoren werden verwendet, die kumulierte Gesamtlaufzeit muss reduziert werden
  - Kleine Programme ggf. seriell schneller, da kürzere Entwicklung

Arten der parallelen Programmierung

- Shared Memory
- Threads
- Data Parallel
- Message Passing

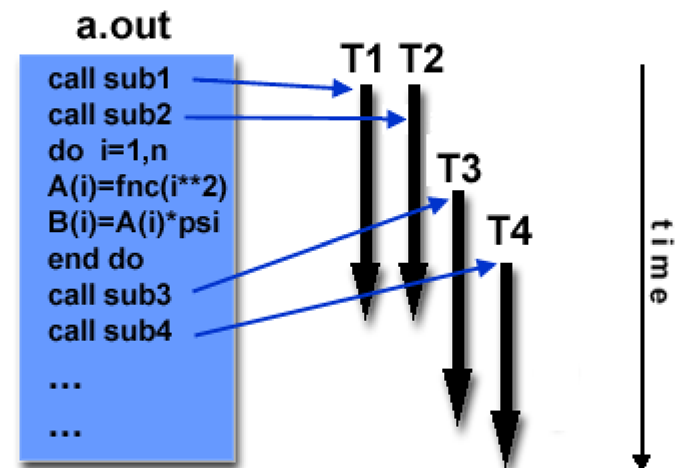
- Die Modelle sind nicht an die Hardware gebunden
  - Theoretisch kann jedes Modell auf jeder Hardware angewendet werden
- Die Wahl des richtigen Modells ist abhängig von den eigenen Präferenzen und der vorhandenen Infrastruktur
- Es gibt kein „Bestes Modell“

Shared Memory Model

- Tasks teilen sich den Adressraum
- lesen und schreiben verläuft asynchron
- Speicherkontrolle durch Kontrollmechanismen z.B. (locks/semaphores)
- Vorteil aus Sicht der Programmierer
  - die Daten werden als Eigentum angesehen, es bedarf keiner Kommunikation der Tasks untereinander
- Nachteil
  - Es ist schwer zu verstehen wie die Daten lokal gehalten werden mit denen der Prozessor arbeitet
  - Cache refreshes und bus traffic steigen, wenn viele Prozessoren gleichzeitig auf die Daten zugreifen müssen

Threads Modell

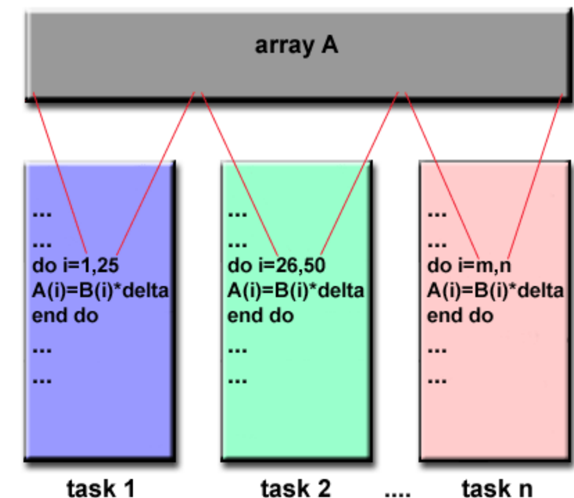
- Ein Prozess kann gleichzeitig viele Programmpfade (execution Paths) haben
  - Hauptprogramm wird in Threads geteilt
- Threads die dem selben Prozess zugeordnet sind verwenden den selben Adressraum
  - Kommunikation untereinander möglich
  - Jeder Thread ist für die Ausführung einer bestimmten (Teil-)Aufgabe verantwortlich
  - Zustände von Threads:
    - Inaktiv
    - rechnend (engl. running)
    - rechenbereit (engl. ready)
    - blockiert (waiting)



Data Parallel Modell

- Daten sind normalerweise in einer gewissen Struktur (z.B. Array)
- Tasks arbeiten gemeinsam auf der selben Datenstruktur, jeder Task allerdings in seinem eigenen Bereich
    - Tasks haben häufig die selben Aufgaben

      (z.B. Alle Elemente des Arrays um 4 erhöhen)



- Shared Mamory:

  alle arbeiten auf dem selben Datenspeicher

- Distributed Memory:

  die Daten werden aufgeteilt und alle arbeiten mit ihren eigenen „Chunks“ im lokalen Speicher

Message Passing Model

Da das Message Passing bei der anschließenden Übung eine zentrale Rolle spielt, wird dies nun detailliert durch

Philipp Tommek

beschrieben!