Verteilte und Parallele Programmierung

Mit Virtuellen Maschinen

Prof. Dr. Stefan Bosse

Universität Koblenz - FB Informatik - FG Praktiksche Informatik

Einführung in die verteilte und parallele Datenverarbeitung

Was unterscheidet verteilte und parallele Datenverarbeitung?

Welche Eigenschaften besitzen verteilte gegenüber parallelen Systemen?

Performanz

- Wir werden noch Metriken und Maßzahlen für die Bewertung von verteilten und parallelen Systemen und Programmen kennen lernen
- Startpunkt: Sequenzielle Performanz
  - Rechenzeit
  - Speicherbedarf
- Messung der sequenziellen Performanz sollte vergleichbar (also normiert) sein bezüglich:
  - Rechnerarchitektur
  - Speicherarchitektur
  - **Programmiersprache** und ggfs. **VM**

Benchmarks

- Es gibt eine Vielzahl von Benchmarks um die Rechenleistung von Rechnern zu bestimmen:
  - Häufig elementare Maschineninstruktionen / Zeiteinheit (GIPS/FIPS)

Aber sind solche Masszahlen für uns hilfreich?

- Besser Vergleich verschiedener Programmiersprachen und deren VM Leistung
  - Whetstone
  - **Dhrystone**

Dhrystone

- Berücksichtigt eine große Menge von Operationen die typischerweise in Programmen vorkommen:

  - Berechnung und Zuweisungsanweisung
  - Verschiedene Datentypen: Skalare, Arrays, Rekords, Objekte (mit Methoden)
  - Statische und dynamische Speicherallokation (in VMs i.a. immer dynamisch!)
  - Funktionen und Funktionsaufrufe
  - Kontrollflusskonstrukte (Schleifen, bedingte Verzweigungen)

https://dl.acm.org/doi/pdf/10.1145/358274.358283 Weicker, R. P. (1984). Dhrystone: a synthetic systems programming benchmark. Communications of the ACM, 27(10), 1013-1030

[Weicker, R. P. (1984)]

TABLE I. Static Statements

Source Language	Knuth 71 [17] FORTRAN	Alex. & Wortman 75 [1] XPL	Elshoff 77 [12] PL/1	Tanen-baum 78 [26] SAL	Grune 79 [14] ALGOL 68	Shimasaki et al. 80 [24] Pascal	Cook & Lee 82 [6] Pascal	Brookes et al. 82 [2] Pascal	De Prycker 82 [8] Pascal	Zeigler & Weicker 82 [28] Ada	Dobbs 83 [10] Ada
Assignment	51.0	54.0	45.7	46.5	49.7	33.8	44.0	42.0	49.3	37.1	33.7
Call	12.0	16.9	15.4	24.9	31.0	40.3	31.8	34.3	29.4	26.8	23.8
user procedures	5.0	?	11.1	24.6	27.7	36.7	16.6	17.9	9.0	26.8	23.8
standard procedures (e.g., I/O)	7.0	?	4.3	0.3	3.3	3.6	15.2	16.4	20.4	–	–
Return	4.0	4.4	0.1	4.2	–	–	–	–	–	6.9	8.2
If	10.0	16.8	21.1	17.2	8.8	18.0	14.8	14.3	9.2	9.8	10.6
with else	–	10.4	7.7	?	?	?	7.4	7.4	?	?	?
without else	10.0	6.4	13.4	?	?	?	7.4	6.9	?	?	?
Loop with Condition	–	2.2	?	2.1	3.2	2.2	3.3	2.2	4.7	1.3	6.4
while	–	2.2		1.6	3.2	1.2	2.6	1.5	3.7	.09	
repeat	–	–		0.5	–	1.0	0.7	0.7	1.0	?	
Loop with "for"	9.0	3.4		3.4	5.1	0.9	2.8	2.1	6.7	0.9	
With	–	–	–	–	–	3.7	2.1	3.8	?	?	?
Case	–	0.8	–	0.3	1.7	0.7	0.9	0.8	0.0	0.4	1.5
Exit Loop	–	–	–	1.4	0.0	–	–	–	–	1.4	2.5
Goto	9.0	1.4	3.8	–	0.0	0.3	0.3	0.5	0.3	0.0	1.6
Other	5.0	–	–	–	–	–	0.1	–	–	15.4	11.6

Abb. 1. Analyse der prozentualen Verteilung von verschiedenen Anweisungen in verschiedenen Programmiersprachen

Führe den Dhrystone Benchmark Test auf verschiedenen Rechnern und verschiedenen VM (und native C Version) aus. VMs: JavaScript, Python, Lua

Ergebnisse:

Rechenzeiten

Die Laufzeit eines Programms setzt sich aus verschiedenen Anteilen zusammen, die je nach Rechnerarchitektur, Betriebssystem und Softwarearchitektur variieren können.

1. Startzeit der Laufzeitumgebung: native SOL/DLL, Betriebssystem, z.B. Prozess, und ggfs. VM
2. Startzeit des Programms: Laden des Codes, Speichermanagement, Initialisierung von Daten
3. Kommunikationszeit Dateneingabe
4. Ausführung und Berechnung
5. Bei Parallelisierung: Kommunikation
6. Kommunikationszeit Datenausgabe
7. Zeit für Programmterminierung

Terminologie

Prozessor

Eine physische Datenverarbeitungseinheit als Teil einer Maschine die i.A. sequenziell einen Strom aus Anweisungen schrittweise verarbeitet.

Virtuelle Maschine

Ein Programm dass einen Prozessor emuliert.

Programm

Eine Menge aus Anweisungen, die von einem Prozessor oder einer Virtuellen Maschine verarbeitet werden kann. Ein Programm enthält neben Anweisungen auch Daten.

Prozess

Ein Programm in Ausführung mit einem zeitlich veränderlichen Daten- und Kontrollzustand

Verteilte vs. Parallele Systeme

Verteiltes System

Ein verteiltes System ist eine System aus **lose gekoppelten** Prozessoren oder Computern, die über ein Kommunikationsnetzwerk miteinander verbunden sind (**Multicomputer**).

- **Speichermodell**: Verteilter Speicher → Jeder Prozessor verfügt über privaten Speicher
- **Kommunikation**: Nachrichtenbasiert über Netzwerke
- **Ressourcen**: Nicht direkt geteilt

Verteilte vs. Parallele Systeme

Paralleles System

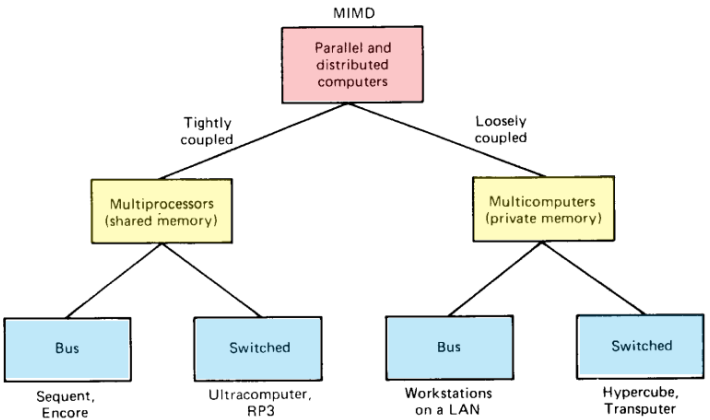
Ein paralleles System ist ein Zusammenführung von **stark gekoppelten** Prozessoren (**Multiprozessoren**)

- **Speichermodell**: Gemeinsamer geteilter Speicher (Shared Memory)
- **Kommunikation**: Prozessoren greifen auf Speicher direkt über elektrische Signale zu → Switched Network (Kreuzschiene) | Bus → Punkt-zu-Punkt | Punkt-zu-N-Netzwerke
- **Ressourcen**: Gemeinsam genutzt (Bus, Speicher, IO)

Man unterscheidet: **Mehrkern** und **Mehrprozessor** Rechner

Verteilte vs. Parallele Systeme

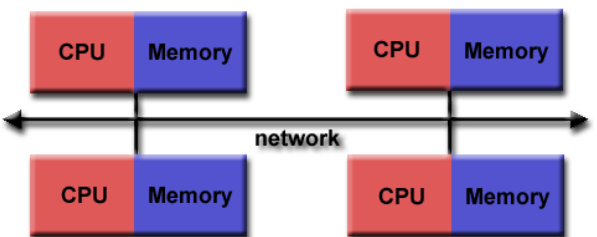
[1]



Taxonomie von verteilten und parallelen Systemen

Verteilter Speicher

- Jeder Prozessor hat eigenen Speicher
- Netzwerk aus Prozessoren/Maschinen
- Zugriff auf Speicher erfordert **nachrichtenbasierte** Netzwerkkommunikation
- Vorteil: Speicher ist *skalierbar mit Anzahl der Prozessoren*
- Nachteil: Langsamer Speicherzugriff zwischen Prozessen

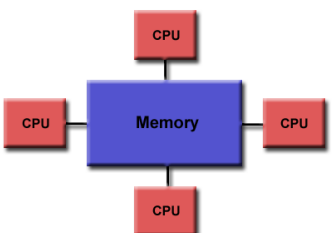


[computing.llnl.gov]

Geteilter Speicher

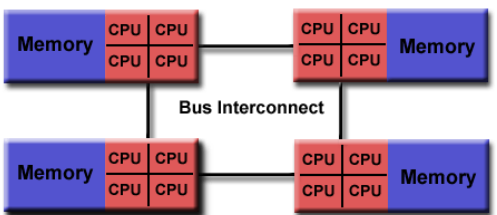
Unified Memory Architecture

- Symmetrisches Multiprocessing (SMP)
- Vorteil: Konstante Zugriffszeit auf Speicher
- Vorteil: Schneller Speicherzugriff zwischen Prozessen



Non Unified Memory Architecture

- Vorteil: Clustering von SMPs
- Nachteil: Ungleiche Zugriffszeiten auf Speicher



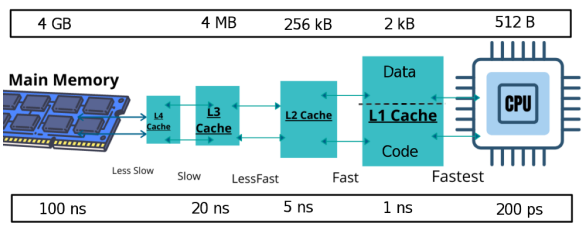
Speichermodell und Speicherarchitekturen

> Das Speichermodell und die Speicherarchitektur einer Rechneranlage bestimmen wesentlich über die Performanz und Skalierbarkeit von paralleler und verteilter Datenverarbeitung!

Speicherhierarchie

- Speichersysteme sind in modernen Rechneranlagen mehrstufig aufgebaut.
- Speicher enthält: 1. Daten 2. Anweisungen (Code)
- Das Speichersystem S eines Rechners besteht aus einer Pipeline von unterschiedlichen Speichermodulen s_{i} mit unterschiedlichen Speichergrößen m_{i}:

$$S(M) = s_1(m_1) \rightarrow s_2(m_2) \rightarrow .. \rightarrow s_k(m_k)$$



Warum gibt es ein hierarchisches Speichersystem?

Verteilte Systeme

Entwurfskriterien und Eigenschaften

Namensgebung

Wie können wir ein Objekt benennen, das weit entfernt an einem unbekanntem Ort ist?

Robustheit

Was passiert, wenn eine Maschine oder ein Netzwerk ausfällt?

Sicherheit

Wie können wir unser System vor Versagen, Betrug, Eindringen, Diebstahl von Daten, ... schützen?

Performance

Langsamer als je zuvor?

Konsistenz

Ich mache eine Banktransaktion, die Bestätigung der Transaktion ging verloren, und die Transaktion wurde wiederholt. Mein Konto wurde zweimal belastet? Was passiert bei gleichzeitigen Zugriff?

Skalierbarkeit

Was passiert mit diesen Kriterien, wenn wir die Anzahl der Prozessoren/Maschinen um das Zehnfache erhöhen?

Parallele Systeme

Definition

- Zerlegung (Partitionierung) eines sequenziellen Algorithmus oder eines Programms in **parallele Tasks** (Prozesse) → **Parallele Komposition**
- Ausführung der Prozesse parallel (nebenläufig und ggfs. konkurrierend) auf mehreren Verarbeitungseinheiten (u. A. generische programmgesteuerte Prozessoren)

Motivation für parallele Datenverarbeitung

- Verkleinerung der Berechnungslatenz

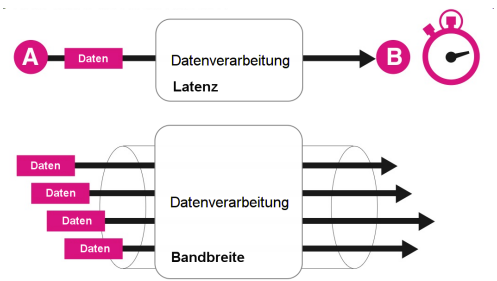
> Def. **Latenz**: Gesamte oder Teilbearbeitungszeit eines Datensatzes

- Erhöhung des Datendurchsatzes

> Def. **Datendurchsatz**: Anzahl der verarbeiteten Datensätze pro Zeiteinheit

Parallele Systeme

- Latenz und Bandbreite sind zunächst unabhängig!
- Pipelining kann die Bandbreite erhöhen (nur sinnvoll bei Datenströmen), aber nicht die Latenz!
- Parallele Tasks können die Latenz verringern



Unterschied von Latenz (oben, einzelner Datensatz) zu Bandbreite (unten, Datensatzstrom)

Parallele Systeme

Parallelität und Nebenläufigkeit

- Parallelität und Nebenläufigkeit ist ein zeitliches Ablaufmodell
- Beschreibt eine zeitliche Überlappung oder Gleichzeitigkeit bei der Ausführung von parallelen Prozessen
- Nebenläufigkeit kann ohne Synchronisation auskommen!
- Häufig mit Konkurrenz verwechselt (English aber richtig: Concurrency==Parallelität, Gelichzeitigkeit)

Konkurrenz

- Concurrency → übereinstimmend!
- Konkurrenz beschreibt den Wettbewerb um geteilte Ressourcen!
- Wettbewerb bedeutet Konflikt welcher aufgelöst werden muss!
- Synchronisation zw. Prozessen!
- Konsens Programmiermodell!

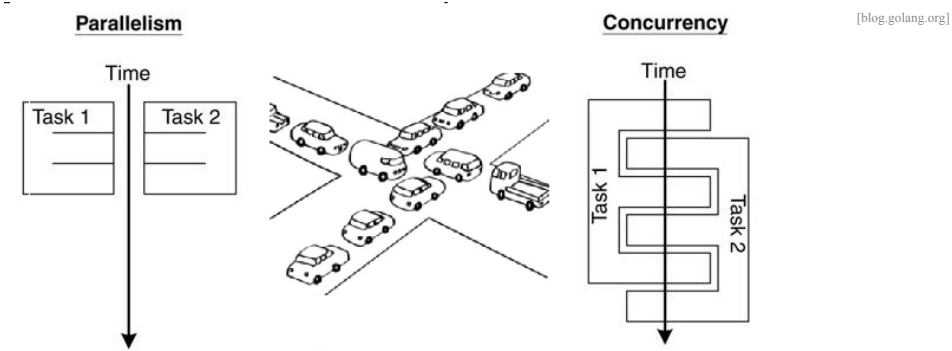


Abb. 2. (Links) Zeitlich überlappende parallele Ausführung von Datenverarbeitung (Rechts) Zeitlich versetzte und synchroniserte Datenverabeitung mit geteilten Ressourcen und Konkurrenz

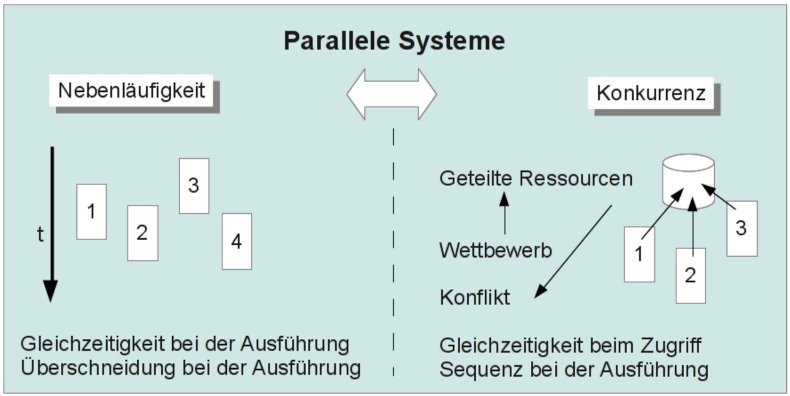


Abb. 3. (Links) Zeitlich überlappende parallele Ausführung von Datenverarbeitung (Rechts) Zeitlich versetzte und synchroniserte Datenverabeitung mit geteilten Ressourcen und Wettbewerb (Konkurrenz)

Konkurrenz (Wettbewerb)

- Es gibt eine oder mehrere geteilte Ressourcen $\mathbb{R}=\{R_1,R_2,..\}$, die nicht atomar sind und daher immer nur einzeln verändert werden dürfen (nicht parallel) → **Datenkonsistenz**
- Gleichzeitiger **Zugriff** wird bei paralleler Ausführung irgendwann zu einem Zeitpunkt $t \geq 0$ auftreten
- Die **Ausführung** des gleichzeitigen Zugriffs wird dann aber sequenziell durchgeführt!

- Parallelität/Nebenläufigkeit wird durch die Ausführungsplattform bereitgestellt!
- Wettbewerb/Konkurrenz muss i.A. durch Programmierung gelöst werden!

Sequenzielle Programmierung mit Lua

Lua ist eine einfach zu erlernende sequenzielle Programmiersprache für Klasse 1/2/3 Interpreter und Skriptprogrammierung

Lua bringt aber auch wichtige Konzepte für die parallele Programmierung mit: Anonyme Funktionen, Funktionale Ausdrücke, Programmflussblockierung

Lua :: Daten und Variablen

- Variablen werden mit dem Schlüsselwort `local` definiert → Erzeugung eines Datencontainers!
- Es gibt keine Typendeklaration in Lua! Dynamische Typisierung zur Laufzeit. **Kerndatentypen**: $\mathbf{T}_{\text{core}}$=`{number, boolean, table, string, function, nil}`
- Alle Variablen sind **polymorph** und können alle Werttypen aufnehmen (auch dynamisch wechselnd zur Laufzeit).
- Bei der Variabledefinition kann ein Ausdruckswert zugewiesen werden

```
local v = ε;  v = ε; u = ε(v);
```

Def. 1. Definition von lokalen Variablen und Verwendung in Anweisungen und Ausdrücken

Beispiele Variablen und Ausdrücke

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT var assign

Lua :: Funktionen

- Funktionen können mit einem Namen oder anonym definiert werden
- Funktionen sind Werte 1. Ordnung → Funktionen können Variablen oder Funktionsargumenten zugewiesen werden
- Eine Funktion kann einen Wert mit der `return` Anweisung zurückgeben. Ohne explizite Wertrückgabe → `undefined`
- Es wird nur Call-by-value Aufruf unterstützt - jedoch werden Objekte, Funktionen und Arrays als Referenz übergeben; Parameter p_i sind an Funktionsblock gebunden

```
function foo (p1,p2,..)
  local p;
  statements;
  return ε
end
⇒
foo(ε1,ε2,..)
```

Def. 2. Funktionsdefinition und Anwendung

Lua :: Funktionen

- Da in Lua Funktionen Werte erster Ordnung sind können
  - Funktionen an Funktionen übergeben werden und
  - Funktionen neue Funktionen zurückgeben (als Ergebnis mit `return`)
- Und Funktionsaufrufe können geschachtelt und rekursiv sein:

```
function f(x) return ε(x) end
function g(x) return f(ε(x)) end
function fac(n) if n>1 then return n*fac(n-1)
                else return n end
x=g(f(ε))
```

- Es können daher **anonyme** Funktionen `function (..) .. end` definiert werden die entweder einer Variablen als Wert oder als Funktionsargument übergeben werden:

```
local foo = function (pi) return ε(pi) end
a = T{1,2,3}
b = a:map(function(elem,index) return ε(elem,index) end)
```

Beispiele Funktionen

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT fun1 fun2

Lua :: Datenstrukturen

> In Lua sind Objekte universelle Datenstrukturen (sowohl Datenstrukturen als auch Objekte) die mit Hashtabellen implementiert werden. Arrays werden in Lua ebenfalls als Hashtabelle implementiert!. D.h. Objekte == Datenstrukturen == Arrays == Hashtabellen.

- Es gibt *kein* nutzerdefinierbares Typensystem in Lua.
- Eine Datenstruktur kann jederzeit definiert und verändert werden (d.h. Attribute hinzugefügt werden)

```
local dataobject = {
  a=ε,
  b=ε, ..
  f=function () .. ed
}
..
dataobject.c = ε
```

Lua :: Datenstrukturen

- Auswahl eines Arrayelements: `array[index]`
- Auswahl eines Datenobjektelements (Attribut): `dataobject.attribute`
- Dadurch dass Objekte und Arrays mit Hashtabellen implementiert (d.h. Elemente werden durch eine Textzeichenkette referenziert) werden gibt es verschiedene Möglichkeiten auf Datenstrukturen und Objektattribute zuzugreifen:

```
dataobject.attribute ↔ dataobject["attribute"] ↔
array[index] ↔ array["attribute"]
```

Def. 3. Lua Äquivalenz Array und Datenstruktur

```
local data = { 1,2,3,4 }
local complex = { r=1, i=-1 }
```

Bsp. 1. Lua Datenobjekte

Lua :: Objekte

- Objekte zeichnen sich in der objektorientierten Programmierung durch Methoden aus mit der ein Zugriff auf die privaten Daten (Variablen) eines Objekts möglich wird.
- In Lua kann auf Variablen eines Objekts (die Attribute) immer direkt zugegriffen werden.
- Attribute können Funktionen sein - jedoch können die Funktionen nicht wie Methoden direkt auf die Daten des Objektes zugreifen.
- Daher definiert man Methoden über Prototypenerweiterung in Lua.
- Die Methoden können über die `self` Variable direkt auf das zugehörige Objekt zugreifen (also auch auf die Variablen/Attribute)
- Es gibt eine Konstruktionsfunktion (Muster über `class` Funktion erzeugbar) für solche Objekte mit Prototypendefinition der Methoden
- Objekte werden mit dem `new` Operation durch die Konstruktionsfunktion erzeugt.

Lua :: Objekte

```
constructor=Class() -- Alt: class()
function constructor:init (pi)
  self.x=ε
  ..
end
function constructor:method (..) {
  self.x=ε;
  ..
}
...

local obj =  constructor:new(..);
obj:method(..)
```

Def. 4. Lua Objektklassen mit Definition einer Konstruktorfunktion, Objektinstanziierung, und Methodenanwendung

Lua :: Objekte

Parallel LuaJit Virtual Machine (LVM)

CLEAR GET RESET RESTART EXIT objects1 objects2 objects3

Zusammenfassung

- Verteilte und parallele Systeme unterscheiden sich vor allem durch die Kopplungsstärke der Verarbeitungselemente (Prozessoren) und dem Speichermodell:
  - Geteilter Speicher
  - Verteilter Speicher
- Es wird unterschieden zwischen paralleler == nebenläufiger Ausführung und Wettbewerb um geteilte Ressourcen
- Eine sequenzielle Programmiersprache wie Lua bietet direkt keine Sprachkonstrukte für die Parallelisierung von Programmen, aber:
  - Funktionen als Werte 1. Ordnung erlauben die funktionale Parallelisierung, die noch behandelt wird.