Verteilte Systeme

...für C++ Programmierer

Parallele Programmierung

by

Dr. Günter Kolousek

Einführung

- Aufteilung der Problemstellung in Teilprobleme
- nebenläufige Abarbeitung der Teilprobleme
- Heute meist
  - hohe Datenparallelität, d.h. viele kleine Teilprobleme $\rightarrow$ *Beschleunigung* einer Aufgabe
  - große Anzahl von Anwendern, d.h. Anzahl der Anfragen auf viele Rechner verteilen, die alle dieselbe SW abarbeiteten $\rightarrow$ unabhängige Aufgaben *gleichzeitig* bearbeiten

Anwendungen

- Baustatik, Maschinenbau, Medizin, Chemie, Biologie, Militär, Physik
- Crashtestsimulationen im Fahrzeugbau, Strömungssimulationen in der Luftfahrttechnik, Wettervorhersage, Rendern in der Computergraphik, Suche in Bildinhalten (Personen, Gegenstände,…), Suchen in großen Problemräumen und in großen Datenbeständen (Brute-force in Kryptographie, Graphensuche wie in Logistik, Schach,… Google, social media,…), DNA-Sequenzanalyse, Vorhersage von Erdbeben und Vulkanausbrüchen, Generierung von Animationsfilmen, Erkennung und Verarbeitung menschlicher Sprache

Situation

- Mooresches Gesetz:

  Die Anzahl der Transistoren pro Chip verdoppelt sich etwa alle zwei Jahre.

  - letzten 50 Jahre: ok
  - nächsten 10-20 Jahre: wahrscheinlich ok
- Wirth'sches Gesetz sagt aus, dass Software schneller langsamer wird, als Hardware schneller.
  - Variante von Bill Gates

    The speed of software halves every 18 months.

$\rightarrow$ Anforderungen werden immer größer!

Situation – 2

- Taktfrequenzen
  - verdoppelten sich in den 1990er-Jahre alle 18 bis 20 Monate
  - Seit 2000-2005 nicht mehr!
  - Maximal 4GHz (im Desktop und Serverbereich)
  - ”Frequency Wall”
    - höhere Frequenz $\rightarrow$ Spannung höher $\rightarrow$ Verlustleistung höher
  - ”Power Wall”
    - Verlustleistung $\rightarrow$ Wärme kann nicht mehr abgeführt werden
  - Existierende, nicht parallelisierte SW profitiert nicht mehr automatisch von der Leistungsteigerung der HW (d.h. durch Steigerungen der Taktfrequenz)

the free lunch is over (Herb Sutter, 2005)

Lösungsansätze

- Problemraum vereinfachen $\rightarrow$ Algorithmus anpassen

Lösungsansätze

- Problemraum vereinfachen $\rightarrow$ Algorithmus anpassen
- Algorithmen optimieren

Lösungsansätze

- Problemraum vereinfachen $\rightarrow$ Algorithmus anpassen
- Algorithmen optimieren
- Implementierung verbessern
  - Facebook
    - `<string>` meist inkludierter Header
    - 18% der CPU-Zeit in `std`
    - Optimierung von `std::string` $\rightarrow$ `fbstring`
      Methode `size()`:

`g++ string`	`fbstring`
1.6ns	0.9ns

Lösungsansätze

- Problemraum vereinfachen $\rightarrow$ Algorithmus anpassen
- Algorithmen optimieren
- Implementierung verbessern
  - Facebook
    - `<string>` meist inkludierter Header
    - 18% der CPU-Zeit in `std`
    - Optimierung von `std::string` $\rightarrow$ `fbstring`
      Methode `size()`:

`g++ string`	`fbstring`
1.6ns	0.9ns

      $\rightarrow$ Gewinn: 1% Performance!!!

Lösungsansätze – 2

- Schnellere Hardware $\rightarrow$ Kosten

Lösungsansätze – 2

- Schnellere Hardware $\rightarrow$ Kosten
- Spezielle Hardware, z.B.
  - Angepasste Schaltungen
    - FPGA (field programmable gate array): kleine Stückzahlen, niedrige Entwicklungskosten, schnelle Anpassung
    - ASIC (application-specific integrated circuit): Kosten ab mittleren Stückzahlen geringer
  - DSP-Prozessoren
  - Graphikprozessoren (GPU)

Lösungsansätze – 2

- Schnellere Hardware $\rightarrow$ Kosten
- Spezielle Hardware, z.B.
  - Angepasste Schaltungen
    - FPGA (field programmable gate array): kleine Stückzahlen, niedrige Entwicklungskosten, schnelle Anpassung
    - ASIC (application-specific integrated circuit): Kosten ab mittleren Stückzahlen geringer
  - DSP-Prozessoren
  - Graphikprozessoren (GPU)
- Aufteilen in Teilprobleme $\rightarrow$ parallele Abarbeitung

Parallelisierung?

- Welche Teilaufgaben lassen sich überhaupt abspalten?
- Lassen sich nicht zerlegbare Algorithmen umformulieren, sodass eine Zerlegung möglich ist?
- Wie groß ist der Anteil der zerlegbaren Teilaufgaben der Gesamtaufgabe?
- Welche Zeiteinsparung ist erreichbar?
- Ist der Nutzer bereit die Kosten zu tragen?
  - Hardware
  - Software
    - Die Entwicklungskosten sind *viel* höher!
    - Man muss die HW gut kennen und die SW daraufhin anpassen.

Möglichkeiten der Parallelisierung

- Zerlegung der Gesamtaufgabe in Teilaufgaben, sodass mehrere Prozessoren die Teilaufgaben parallel abarbeiten können.
- Zerlegung in Teilaufgaben, die hintereinanader ausgeführt werden
  - Gesamtzeit der Lösung einer Gesamtaufgabe wird nicht kürzer
  - Durchsatz bei der Lösungen vieler Aufgaben höher
- Zerlegung in Teilaufgaben, die hintereinander ausgeführt werden, aber die mit spezieller HW (meist parallel) gelöst werden

Parallelität in der HW

- Prozessorarchitektur
  - Pipelining
  - Superskalarität
  - HW-seitiges Multithreading
  - Vektoreinheiten
  - Coprozessoren
- Rechnerarchitektur
  - Multicore-Prozessoren (!)
  - Multiprozessorsysteme
  - Cluster

Pipelining

1. Befehl aus Arbeitsspeicher (engl. fetch)
2. Befehl dekodieren (engl. decode) und ggf. Daten aus Registern oder dem Arbeitsspeicher laden
3. Befehl ausführen (engl. execute)
4. Ergebnis in Register oder Arbeitsspeicher schreiben (engl. write back)

Pipelining – 2

Befehl 1	fetch	decode	execute	write	
Befehl 2		fetch	decode	execute	write
Befehl 3			fetch	decode	execute
…					

Abhängigkeiten zwischen Befehlen $\rightarrow$ Wartezyklen bis Ergebnis

- Datenabhängigkeit
- Abhängigkeiten im Kontrollfluss (z.B. bedingte Sprunganweisungen)

Pipelining – Optimierungen

- Umordnungen
  - durch Compiler
  - durch Prozessor
  - Probleme durch bedingte Verzweigungen
    - u.U. Rücknahme von Instruktionen
- Prefetching: Laden von Daten aus dem Hauptspeicher weit vor der Benutzung ($\rightarrow$ Out-of-Order … OoO)
  - U.U. Verwerfen der Ergebnisse und Rücksetzen der Register
  - $\rightarrow$ Angriffsvektor: Fehlspekulationen haben Nebeneffekte
    - z.B. Spectre-1: Längenüberprüfung von Feldern, Daten werden im vorhinein gelesen, dann liegt Ergebnis der Längenprüfung vor, aber: Daten bleiben im Cache!

Superskalarität

superskalare Prozessoren enthalten mehrere gleichartige Funktionseinheiten

- Rechenwerke für Ganzzahl- und Gleitkommaarithmetik
- Lade- und Speichereinheiten

Damit können mehrere Befehle parallel ausgeführt werden (wenn keine Abhängigkeiten)

HW-seitiges Multithreading

- mehrere Threads $\rightarrow$ Wartezeiten, z.B. bei Hauptspeicherzugriffen
- daher in der Zwischenzeit Befehle eines anderen Threads ausführen
- dazu: mehrere Registersätze!
- wird auch Hyper-Threading genannt
- HW-Threads erscheinen dem Benutzer wie echte Kerne
- Performancegewinn ca. 10-20%

CPU-Info in Linux

```
$ lscpu
Architecture:            i686
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  4
On-line CPU(s) list:     0-3
Thread(s) per core:      2
Core(s) per socket:      2
Socket(s):               1
...
```

Vektoreinheiten

- ein Befehl verarbeitet mehrere Daten gleichzeitig
  - z.B. Vektoraddition
  - z.B. Verarbeitung mehrerer Pixel eines Bildes
- Intel MMX (1997)
  - 64-Bit-Register: 8 Bytes oder 4 16-Bit-Wörter oder 2 32-Bit-Wörter
- Intel SSE (SSE2, SSE3,…)
  - 128 bzw. 256-Bits, d.h. auch Gleitkommazahlen

Vektoreinheiten – 2

- Flynnsche Klassifikation
  - SISD … single instruction, single data
    - klassische Von-Neumann Architektur
  - SIMD … single instruction, multiple data
    - Vektorprozessoren
  - MISD … multiple instructions, single data
    - theoretischer Natur
  - MIMD … multiple instructions, multiple data
    - Multicore- und Multiprozessorsysteme

Coprozessoren

- FPU (floating point unit), heute in der CPU
- GPU (graphics processing unit)
  - werden zunehmend für numerische Berechnungen verwendet $\rightarrow$ GPGPU (general purpose computation on graphics processing units)
- Spezielle Coprozessoren
  - Dekodieren von Videos
  - Ver- und Entschlüsseln

Rechnerarchitektur

- Einteilung bzgl. Aufbau
  - homogen: alle Rechner/Prozessoren/Kerne gleich
  - heterogen: verschiedenartige Rechner/Prozessoren/Kerne, z.B. Graphikkern in CPU
- Speicherarchitekturen
  - UMA (uniform memory access): alle Prozessoren/Kerne: Zugriff auf gleichen Hauptspeicher
  - NUMA: jeder Prozessor: eigener Speicher, Zugriff auf fremden Speicher: Verbindungsnetzwerk (Faktor 2!)
- Rechnerarchitektur
  - Multicore vs. Multiprozessor
  - Cluster-Architektur: heterogen/homogene Rechner verbunden über Netzwerk
- $\rightarrow$ Shared memory vs. Message passing

Speicherhierarchie

- Prozessorregister (Prozessortakt, KiB, ~1ns)
- Cache (einige Dutzend Taktzyklen abhängig von Level, 10-100ns) (Desktop und Server)
  - Level 1 Cache (je Kern, aufgesplittet in Befehlscache und Datencache, von 128KiB bis 480KiB je Cache)
  - Level 2 Cache (je Kern, von 1MiB bis 3.5MiB)
  - Level 3 Cache (je Prozessor, von 8MiB bis 37.5MiB)
- Arbeitsspeicher (Hunderte Taktzyklen, GiB, ~1μs)
- NUMA-Speicher
- Disk-Speicher (TiB, >10ms)

Beispiele

- Google
  - unterschiedliche Anforderungen: möglichst schnelles Lösen einer Aufgabe vs. möglichst viele Benutzeranfragen bearbeiten
  - Zeitpunkt ????: Gesamtenergieverbrauch 600MW
  - 2008: mehrere Hunderttausend Server
    - 36 Datencenter
    - 150 Racks pro Datencenter
    - 40 Server pro Rack
    - $\rightarrow$ mehr als 200000 Server! …und jeden Tag mehr!!!

Beispiele – 2

- NSA
  - Rechenzentrum in Utah
    - 60 MW Einspeisung, 250W/Mainboard $\rightarrow$ 150000 Rechner
  - mind. 3 Rechenzentren!

- Supercomputer in Wuxi, Jiangsu, China: 93.0146 PFLOPS = 93014.6 TFLOPS = 93014600 GFLOPS
  - bei 15.37 MW!
- Desktop: Intel Core i7, 3.2GHz, 4 Kerne ca. 45 GFLOPS

Parallelität in der SW

- Prozesse und Threads
- Parallelisierende Compiler

  OpenMP Open Multi-Processing, Erweiterung zu C, C++ und FORTRAN. Parallelisierung der Schleifen auf Thread-Basis

  CilkPlus basierend auf C und C++. Parallelisierung der Schleifen auf Thread-Basis

  OpenCL basierend auf C, um heterogene Prozessoren zu programmieren (meist CPU & GPU).
- Parallele Bibliotheken

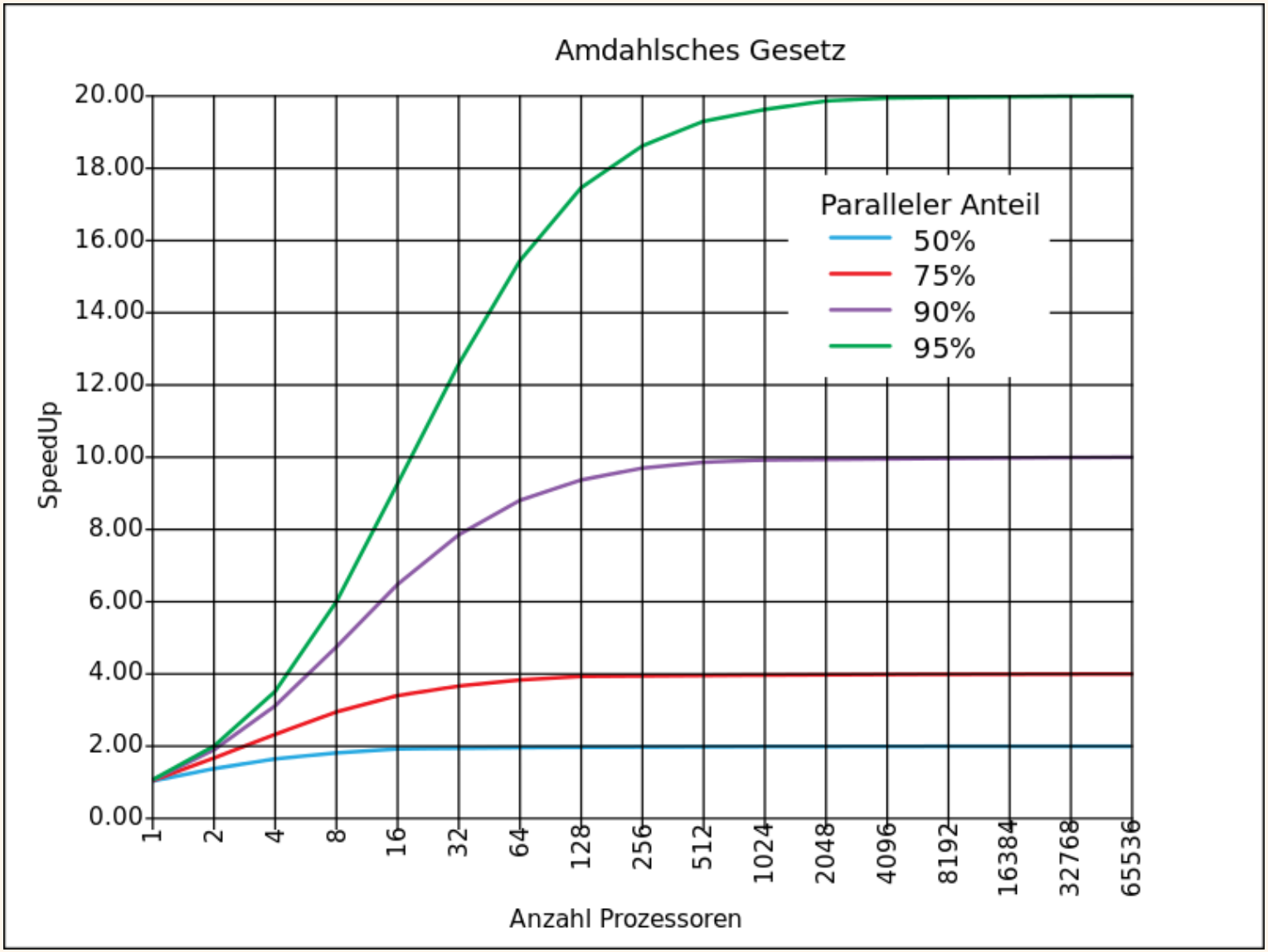
  TBB Threading Building Blocks, C++ Bibliothek, $\rightarrow$ Multicore-Software effizient entwickeln.

  MPI Message Passing Interface, C, C++, Fortran, Java, C#, Python. API, Nachrichten zwischen parallelen Prozessen

Amdahlsches Gesetz

- beschreibt die Grenzen der Parallelisierbarkeit
  - Programm: sequentieller und paralleler Anteil
  - egal wie gut wir parallelisieren (unabhängig von der # der Prozessoren): das parallele Programm ist nicht schneller als der sequentielle Anteil
- paralleler Anteil P (in Prozent durch 100), z.B. 75% kann parallelisiert werden $\rightarrow P = 0.75$
- Beschleunigung (engl. speedup) eines Programmes mit N Kernen: $S(N) = \frac{T_1}{T_N} \leq N$
- Herleitung von $S(N)$:
  $S(N) = \frac{T_s + T_p}{T_s + \frac{T_p}{N}} = \frac{T(1-P)+TP}{T(1-P)+\frac{TP}{N}} = \frac{1}{(1-P)+\frac{P}{N}} \leq \frac{1}{1-P} = S_{max}$
- z.B. $P = 0.75, S_{max} = 4$

Amdahlsches Gesetz – 2



Quelle: Wikipedia

Amdahlsches Gesetz – 3

- zu pessimistisch: u.U. größerer Cache $\rightarrow$ Verbesserung der Leistung (da u.U. gesamter Code im Cache)
- zu optimistisch: Koordination, Synchronisation und Kommunikation nicht in Betracht gezogen

  Erweiterung um diesen Anteil:

$$S(N) = \frac{1}{(1-P)+o(N)+\frac{P}{N}}$$

Nebenläufigkeit (engl. concurrency)

- verbesserter Durchsatz $\rightarrow$ mehr (Teil-)Aufgaben je Zeiteinheit
  - Taskparallelität (engl. task parallelism): Aufteilung der Gesamtfunktion in verschiedene Teilfunktionen und jeder Thread bearbeitet eine Teilfunktion.
  - Datenparallelität (engl. data parallelism): Aufteilung der zu bearbeitenden Daten in verschiedene Datenpakete und jeder Thread bearbeitet ein Datenpaket (gleiche Funktion!)
- verbessertes Antwortzeitverhalten: I/O-intensive Anwendungen warten oft auf Ein- bzw. Ausgabe $\rightarrow$ Prozess (anderer Thread) kann andere Aufgabe erledigen (z.B. GUI, Webserver,...)
- bessere Programmstruktur $\rightarrow$ Separation of concerns (”Trennung von Belangen”)

Nebenläufig vs. parallel

- Die Anweisungen zweier Prozesse werden parallel bearbeitet, wenn die Anweisungen unabhängig voneinander zur gleichen Zeit ausgeführt werden.
  - $\rightarrow$ 2 Kerne oder 2 Prozessoren notwendig
- Zwei Prozesse heißen nebenläufig, wenn ihre Anweisungen unabhängig voneinander abgearbeitet werden (können).
  - $\rightarrow$ auch auf einem Kern (Prozessor) möglich (preemptive multitasking)
- $\rightarrow$ parallel *ist* nebenläufig

Anforderungen an die Entwicklung

- Effizienz der Softwareentwicklung
  - parallele SW ist komplexer $\rightarrow$ Aufwand!
  - Programmiersprachen, z.B. Python vs. C++
    - ”Performance speed is no longer the primary worry. Time to market speed is.” – Hui Ding (Instagram engineer)
  - Bsp: Python als Programmiersprache bei Instagram (6/2017)
    - 95 Millionen Photos und Videos
    - 600 Millionen registrierte Benutzer, davon 400 Millionen aktiv je Tag!
- Portierbarkeit
  - meistens abhängig von HW
- Skalierbarkeit
  - Steigerung der parallel arbeitenden Prozessor(kerne)