Parallelisierung

Verteilte Softwaresysteme

Prof. Dr. Oliver Braun

Letzte Änderung: 30.10.2018 09:49

Leistungsmaße für parallele Programme

Laufzeit (1/3)

- bei der Bewertung paralleler Algorithmen bzw. Implementierung
  - Laufzeit eines Programmlaufs auf der Zielplattform

$$T_p(n)$$

  - Zeit zwischen Start und Beendigung auf allen beteiligten Prozessoren
- Laufzeit in Abhängigkeit
  - der Anzahl p der beteiligten Prozessoren und
  - der Problemgröße n

Laufzeit (2/3)

- setzt sich zusammen aus:
  - **Rechenzeit** (T_CPU): Berechnung mit lokalen Daten der einzelnen Prozessoren
  - **Kommunikationszeit** (T_COM): Zeit für Austausch von Daten zwischen Prozessoren
  - **Wartezeit** (T_WAIT): wegen ungleicher Last oder Datenabhängigkeiten
  - **Synchronisationszeit** (T_SYN): Synchronisation beteiligter Prozesse bzw. Prozessoren
  - **Platzierungszeit** (T_Place): Zeit für Allokation der Tasks auf die einzelnen Prozessoren, sowie mögliche dynamische Lastverteilung zur Laufzeit
  - **Startzeit** (T_Start): Zeit zum Starten der parallelen Taks auf allen Prozessoren

Laufzeit (3/3)

- zur Reduktion der Laufzeit muss **Overheadzeit**

$$T * CWS = T * COM + T * WAIT + T * SYN$$

  reduziert werden

- Platzierungszeit und Startzeit nennt man zusammen **Rüstzeit**

$$T * Setup = T * Place + T_Start$$

Messen der Laufzeit

Vermesse mit einer Maus einen Elefanten und vermesse nie mit einem Elefanten eine Maus.

- soll heissen: die Laufzeit des Messprogramms muss vernachlässigbar klein im Vergleich zur Laufzeit des zu messenden Programms sein
- ist dies nicht möglich, entweder
  - Problemgröße erhöhen, so dass sich die Laufzeit des zu messenden Programmes erhöht, oder
  - Laufzeit des Messprogramms ermitteln und von der Gesamtlaufzeit abziehen

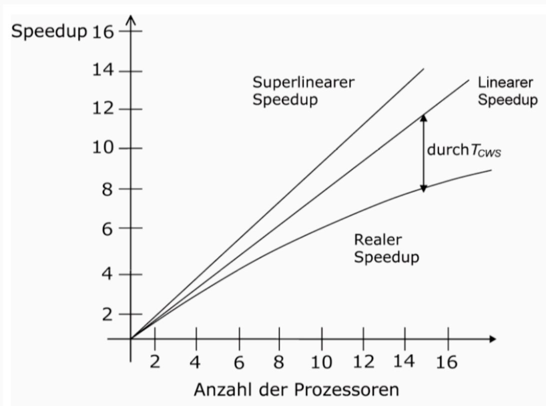
Speedup (1/2)

- Reduktion der Laufzeit für das Gesamtproblem bei einer Parallelisierung, gibt der **Speedup** an (auch **Beschleunigung** oder **Leistungssteigerung**):

$$S_p(n) = \frac{T'(n)}{T_p(n)}$$

- $T'(n)$ ist Laufzeit des schnellsten bekannten sequentiellen Algorithmus
- $T_1(n)$ ist die Laufzeit des parallelen Programms auf einem Prozessor
  - entspricht nicht notwendigerweise $T'(n)$

- der Speedup ist normalerweise beschränkt durch die Anzahl der Prozessoren: $S_p(n) \leq p$



Superlinearer Speedup: Erklärung später

Kosten und Overhead

- **Kosten** eines parallelen Programms sind Maß für die von allen Prozessoren durchgeführte Arbeit:

$$C_p = T_p(n) \cdot p$$

- ein paralleles Programm ist **kostenoptimal**, wenn es genauso viele Operationen ausführt wie das sequentielle Programm mit Laufzeit $T'(n)$
  - bei Kostenoptimalität gilt: $C_p = T'(n)$
- der **Overhead** ist die Differenz der Kosten:

$$H_p(n) = C_p(n) - T'(n) = p \cdot T_p(n) - T'(n)$$

Effizienz (1/2)

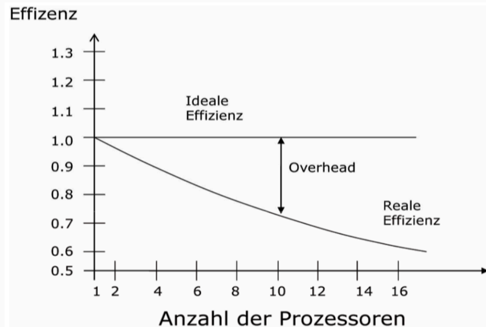
- die **Effizienz** eines parallelen Programms gibt die relative Verbesserung der Verabeitungsgeschwindigkeit bezogen auf die Anzahl der Prozessoren an:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T'(n)}{p \cdot T_p(n)} = \frac{T'(n)}{C_p(n)}$$

- die Effizienz zeigt also an
  - wie effizient wurde der ursprüngliche sequentielle Algorithmus parallelisiert und
  - wie effizient werden die Prozessoren ausgenutzt
- die Effizienz bewertet also Zusatzlast und Redundanz (Overhead)

Effizienz (2/2)

1. $E_p(n) < 1$:
   Suboptimal, aber der Normalfall, Werte nahe 1 sind anzustreben
2. $E_p(n) = 1$:
   Kostenoptimal
3. $E_p(n) > 1$:
   Superlinear, besser als sequentieller Algorithmus



Amdahls Gesetz (1/3)

- benannt nach dem Computer-Architekten *Gene Amdahl*
- wird benutzt um für ein Gesamtsystem die maximal zu erwartende Verbesserungen vorherzusagen
- Festlegung des theoretisch maximal erreichbaren Speedups
- bei einer Implementierung eines parallelel Algorithmus existiert für jede Problemgröße ein bestimmter **sequentieller Anteil** f, der nicht parallelisiert werden kann
- für f gilt: $0 \leq f \leq 1$
- der parallel ausführbare Anteil ist: $(1 - f)$
- ist $f = 0$ ist der Algorithmus vollständig parallel ausführbar

Amdahls Gesetz (2/3)

- die gesamte Laufzeit ist dann größer oder gleich der Summe der Zeit für den sequentiellen Anteil und den parallelen Zeiten dividiert durch die Anzahl der Prozessoren:

$$T_p(n) \geq f \cdot T'(n) + \frac{(1-f) \cdot T'(n)}{p}$$

- für den Speedup folgt **Amdahls Gesetz**:

$$S_p(n) \leq \frac{T'(n)}{T_p(n)} = \frac{T'(n)}{f \cdot T'(n) + \frac{(1-f) \cdot T'(n)}{p}} = \frac{1}{f + \frac{(1-f)}{p}}$$

- daraus folgt, dass schon ein relativ kleiner, nicht parallelisierbarer Anteil den maximalen Speedup begrenzt

Amdahls Gesetz (3/3)

- für $f > 0$ und eine große Anzahl von Prozessoren $p \to \infty$ folgt näherungsweise

$$S_p(n) \leq \frac{1}{f}$$

- d.h. z.B. beträgt der sequentielle Anteil 10%, so kann das Problem nur um den Faktor 10 beschleunigt werden

  bei 5% sequentiellem Anteil nur um den Faktor 20

- d.h. parallele Berechnungen sind nur für eine kleine Anzahl von Prozessoren oder sehr kleine Werte von f sinnvoll

Gustafsons Gesetz (1/2)

- Amdahl hält Problemgröße konstant
- Jon L. Gustafsons nimmt an, dass in der Praxis die Problemgröße mit der Anzahl der Prozessoren anwächst
  - der parallelisierbare Anteil wächst linear mit der Problemgröße und der Prozessoranzahl
- sei f_1 der sequentielle Anteil bei einer festen Problemgröße
- dann lässt sich die Verringerung des sequentiellen Anteils bei größeren Problemgrößen und Erhöhung der Anzahl der Prozessoren p ausdrücken durch

$$f = \frac{f_1}{p \cdot (1 - f_1) + f_1}$$

Gustafsons Gesetz (2/2)

- mit Amdahls Gesetz folgt

$$S_p(n) \leq \frac{1}{f + \frac{(1-f)}{p}} = p \cdot (1 - f_1) + f_1$$

- aus dem Gesetz von Gustafson folgt
  - unter Voraussetzung fester Laufzeit f_1 und Begrenzung der Problemgröße durch die Prozessoranzahl
  - dass der Speedup mit konstantem sequentiellen Teil annähernd linear mit der Prozessoranzahl wächst
- damit ist bei sehr großem p und entsprechenden Problemgrößen ein Speedup nahe p möglich

Skalierbar

- eine Anwendung heisst **skalierbar**, falls
    - wachsende Problemgröße
    - durch wachsende Anzahl eingesetzter Prozessoren

  so kompensiert werden kann, dass
    - die zur Problemlösung benötigte Zeit **nahezu konstant** bleibt

Amdahl oder Gustafson?

- ist die Anwendung **nicht skalierbar**
  - gilt das Gesetz von Amdahl
- ist die Anwendung **perfekt skalierbar**
  - gilt das Gesetz von Gustafson

Karp-Flatt-Metrik (1/2)

- bei den Gesetzen von Amdahl und Gustafson spielt die Overheadzeit T_{CWS} keine Rolle
- der sequentielle Anteil f ist dabei unabhängig von der Anzahl der Prozesse p
- zur Bestimmung des Overheads:
  - für mehrere Werte von p experimentell bestimmt
  - durch Auflösung Amdahls Gesetz nach f bestimmt
- nach Amdahl gilt

$$\frac{1}{S_p(n)} = f + \frac{(1-f)}{p}$$

- aufgelöst nach f ergibt die sich die **Karp-Flatt-Metrik**:

$$f = \frac{1/S_p(n) - 1/p}{1 - 1/p}$$

Karp-Flatt-Metrik (2/2)

- da f abhängig von $p \Rightarrow$ Aussagen über T_{CWS} möglich
- die Karp-Flatt-Metrik zeigt an, ob die Effizienz
  - durch den inhärenten sequentiellen Anteil (f bleibt für wachsendes p nahezu konstant) oder
  - durch den parallelen Overhead (f wächst in Abhängigkeit von p)

  dominiert wird

Parallelisierungstechniken

Inhärenter Parallelismus (1/3)

- besteht das Problem aus vielen Teilproblemen
- die voneinander total unabhängig sind
- d.h. keine funktionalen Abhängigkeiten
- keine gemeinsamen Daten
- so besitzt es einen **inhärenten Parallelismus**
- d.h. das Ausgangsproblem zerfällt in n unabhängige Teilprobleme
- diese können dann auf p Prozessoren verteilt werden

Inhärenter Parallelismus (2/3)

- Kommunikation ist lediglich vor und nach der eigentlichen Berechnung notwendig
  - Daten verteilen
  - Ergebnisse einsammeln
- durch die minimale Kommunikation ist (fast) linearer Speedup möglich

Inhärenter Parallelismus (3/3)

- es können zwei Probleme auftreten:
    1. Anzahl der Rechenoperationen der Teilprobleme kann stark variieren und nicht konstant sein.
    2. Prozesse können auf unterschiedlich schnellen CPUs laufen.
- Gesamtlaufzeit hängt bei statischer Lastverteilung vom aufwändigsten Teilproblem oder von der langsamsten CPU ab
- während ein Teil noch rechnet, kann ein Teil schon fertig sein und muss warten

Superlinearer Speedup

- Landesbank Baden-Würtemberg hat Monte-Carlo-Simulationen zur Risikoanalyse eingesetzt
- mehrfache Monte-Carlo-Simulationen sind vollkommen unabhängig voneinander
- wurden parallelisiert auf 16 Windows-NT-Rechnern
- dabei wurde mehr als linearer Speedup erreicht
- Grund:
  - mit jedem Rechner der dazu kam reduzieren sich die Daten für die Monte-Carlo-Simulation
  - dadurch verringerte sich der Overhead der virtuellen Speicherverwaltung

Zerlegungsmethoden

- vorherrschende Technik: **Divide and Conquer**
- Problem wird in zwei oder mehrere einfachere Unterprobleme von ungefähr gleicher Größe aufgeteilt
  - Unterprobleme müssen völlig unabängig voneinander sein
- die Unterprobleme werden rekursiv weiter aufgeteilt, solange bis keine Aufteilung mehr möglich ist
- verschiedene Zerlegungsmethoden (werden im Folgenden ausführlicher behandelt)
  - Funktionale Zerlegung
  - Datenzerlegung
  - Funktions- und Datenzerlegung

Divide and Conquer

- gegeben zu lösendes Problem mit Größe n
- vernachlässigen wir zunächst Aufteilen und Kombinieren der Ergebnisse
- gehen von binären Baum aus mit p Unterprobleme an den Blättern, dann gilt

$$T'(n) = n \text{ und } T_p(n) = \frac{n}{p}$$

- das ergibt folgenden Speedup

$$S_p(n) = \frac{T'(n)}{T_p(n)} = \frac{n}{n/p} = p$$

- also nicht ganz lineare Speedup, wenn wir Aufteilen und Kombinieren mit berücksichtigen

Funktionale Zerlegung

- Zerteilung des Problems in mehrere Arbeitsschritte (engl. **function decomposition**)
- Standardvorgehen in der funktionalen Programmierung
- Funktionen können an verschiedenen unabhängigen Teilen arbeiten
    - analog zu inhärentem Parallelismus
- gibt es eine Datenabhängigkeit zwischen den Schritte
    - **Pipeline** oder **Fließbandverarbeitung**

Pipeline (1/2)

- Problem zerfällt in n voneinander unabhängige Teilprobleme
- Verteilen von je n/p Teilprobleme auf jeden von p Prozessoren
- Datenabhängigkeit muss durch Synchronisation und Kommunikation gelöst werden
- wenn Teilprobleme sehr unterschiedlich, hängt die Gesamtlaufzeit wieder vom aufwändigsten Teilproblem ab

Pipeline (2/2)

- sei p die Stufen der Pipeline bzw. die Anzahl der Teilprobleme, die für jedes Element gelöst werden
- sei n die Problemgröße bzw. Anzahl der Elemente die in die Pipeline eingefüttert werden und die Pipeline am anderen Ende verlassen
- dann gilt ohne parallele Verarbeitung: $T'(n) = p \cdot n$
- im parallelen Fall mit Pipeline gilt: $T_p(n) = p + n$
- somit ist der Speedup:

$$S_p(n) = \frac{T'(n)}{T_p(n)} = \frac{p \cdot n}{p + n} = \frac{p \cdot n}{n \cdot (\frac{p}{n} + 1)} = \frac{p}{\frac{p}{n} + 1}$$

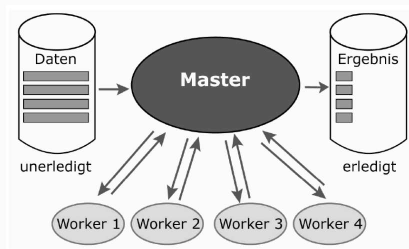
- für große n ($n \to \infty$) folgt: $S_p(n) = p$
- negativ bei der Pipeline ist, dass Ausfall einer Stufe der Pipeline alles stoppt

Datenzerlegung

- einmalige Zerlegung der Daten oder des Eingabebereichs (engl. **domain decomposition**)
- Aufteilung auf mehrere Prozessoren auf denen der selbe Algorithmus läuft
- **Master-Worker-Schema**
- bei rekursiver Datenzerlegung ergeben sich **Berechnungsbäume**

Master-Worker-Schema (1/2)

- der **Master**
  - verteilt die Daten an die Worker und
  - nimmt die Ergebnisse der Worker entgegen
- die **Worker**
  - fragen nach einem unerledigten Datenbereich



- Overhead der Parallelisierung ist die Rüstzeit
  - also Programmstart, Verteilen und Einsammeln

Master-Worker-Schema (2/2)

- Annahmen
    - Regularität der Daten und Uniformität der Arbeit
    - d.h. für alle Teildaten die gleiche Anzahl von Rechenoperationen und
    - alle Algorithmen sind gleich und laufen auf gleich schnellen CPUs
- wir vernachlässigen außerdem die Kommunikation zwischen Master und Workers und die Arbeit des Masters
- sei n die Größe der einzelnen Datenzerlegung und p die Anzahl der Worker, dann gilt: $T'(n) = p \cdot n$
- die Laufzeit im parallelen Fall beträgt für jeden Worker: $T_p(n) = n$
- somit gilt:

$$S_p(n) = \frac{T'(n)}{T_p(n)} = \frac{p \cdot n}{n} = p$$

- je feiner die Granularität der Datenzerlegung ist
    - umso mehr steigt die Kommunikation und die Arbeit des Masters
    - Master wird zum leistungsbeschränkenden Flaschenhals

Berechnungsbäume

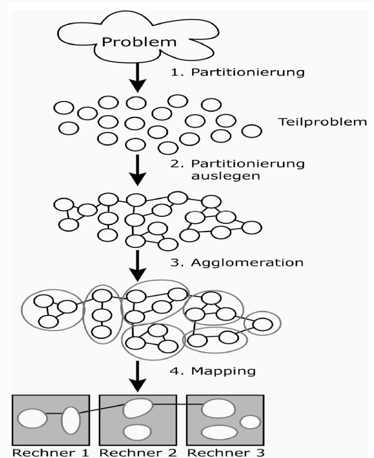
- rekursive Datenzerlegung führt zu Berechnungsbäumen
- Beispiel: Mergesort
    - beim Absteigen des Baumes wird das Feld in zwei gleich große Teile zerlegt
    - diese können parallel sortiert werden
    - nach dem Sortieren werden dann zwei Felder verschmolzen
- auch Berechnungsbaum, aber nicht so symmetrisch: Quicksort
    - zunächst aufteilen und dann parallel weiter sortieren
- es gibt noch weitere *Data Parallel Algorithms*

Funktions- und Datenzerlegung

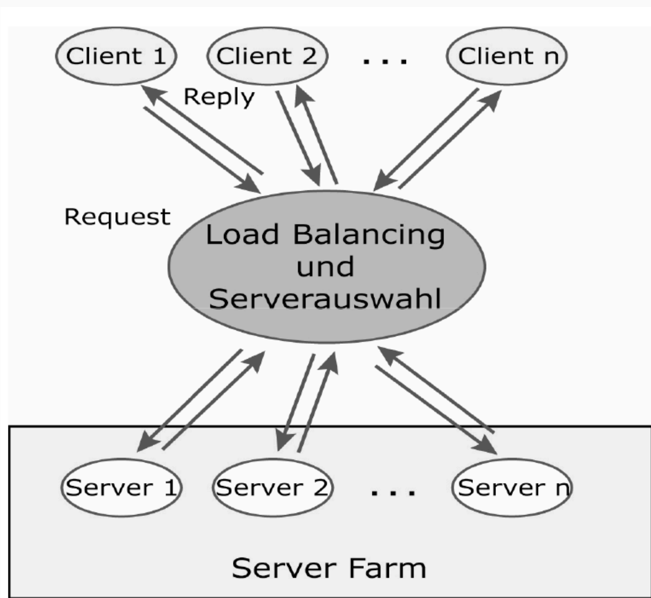
- Zerlegung nach Gesichtspunkten einer funktionalen Zerlegung und/oder einer Datenzerlegung
- dann entweder
    - statisch allokieren, d.h. vor der Laufzeit auf Prozessoren abbilden oder
    - dynamische allokieren durch Lastausgleicher

Methodisches Vorgehen bei der Zerlegung

1. Partitionierung, Zerlegung
   - Funktions- und Datenzerlegung
   - identische Daten, zur Verringerung des Kommunikationsbedarfs, replizieren
2. Auslegung der Kommunikation
   - möglichst effizient und ohne Blockierung der Prozesse
3. Agglomeration (Zusammenballung)
   - falls zu viele Teilaufgaben
4. Mapping



Dynamische Allokation



Weitere parallele Verfahren und Algorithmen

- es gibt noch viele weitere Gebiete und viel Literatur, z.B. zu
  - allgemeine parallele Algorithmen
  - parallele Sortier- und Suchalgorithmen
  - parallele Graph-Algorithmen
  - parallele Numerik
  - parallele Metaheuristiken