Parallel Computing I

Einführung in das
Hochleistungsrechnen

Parallele Performance

7. Mai 2012
Paralleles Rechnen
SS 2012
Thorsten Grahs

Parallele Performance

- Skalierung
- Speedup
- Effizienz
- Amdahlsches Gesetzt
- Gesetz von Gustafson
- Einfache Performancemodell
- Beispiele (Summation, Innere Produktbildung)

Parallele Performance

Übersicht

- Skalierung
- Effektivität (Speedup)
- Performanzmodelle

- Kann Effektivität einer Parallelisierung vorausgesagt werden?
- Welche Einflüße auf die Effektivität haben
  - CPU-Geschwindigkeit
  - Speicher
  - Netzwerk
- Welchen Einfluß haben verschiedene Modellierungsarten?

Beschleunigung durch Parallelisierung

Fragestellung:

Kann man also 1000 Prozessoren, welche jeweils 1 GFlop leisten, in den Bereich des TFlop-Computing vorstossen?

im Prinzip ja, aber...

- Technische Beschränkungen
- Nicht alle Programmteile lassen sich parallelisieren
- Einfluss von CPU, Speicher, Netzwerk
- Speed-Up?

Parallele Laufzeit

$T_p(n)$ p # **Prozessoren,** n **Modellgröße**

Zeit zwischen dem Start der Abarbeitung des parallelen Programms und der Beendigung der Abarbeitung aller beteiligten Prozesse

Die Laufzeit eine parallelen Programms auf Systemen mit verteiltem Speicher setzt sich folgendermaßen zusammen:

- **lokale Berechnungen**
  Berechnungen einen Prozessors unter Verwendung lokaler Daten
- **Austausch von Daten**
  Ausführung von Kommunikationsoperationen
- **Wartezeiten**
  (z.B. aufgrund ungleicher Verteilung der Rechenlast)
- **Synchronisation**
  Abgleich zwischen den ausführenden Prozessoren

Kosten eines parallelen Programms

Kosten $C_p(n)$

Die Kosten (Arbeit, Prozessor-Zeit-Produkt) eines parallelen Programms sind definiert als

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

Kosten $C_p(n)$

- berücksichtigen die Zeit, die **alle** an der Ausführung beteiligten Prozessoren zur Abarbeitung des Programms verwenden
- sind ein **Maß für die von allen Prozessoren durchgeführte Arbeit**.
- sind **kostenoptimal**, wenn insgesamt genauso viele Operationen ausgeführt werden, wie vom schnellsten sequentiellen Verfahren mit Laufzeit $T^*(n)$, d.h. $C_p(n) = T^*(n)$.

Paralleler vs. serieller Anteil – Speedup

Speedup $S_p(n)$

Der Speedup eines parallelen Programmes mit Laufzeit $T_p(n)$ ist definiert als

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

$T^*(n)$ Laufzeit der optimalen sequentiellen Implementierung.

Maß für den

- Vergleich von sequentieller und paralleler Implementierung.
- Maß für den relativen Geschwindigkeitsgewinn
- Idealfall: linearer Speedup $T^* = T_p \cdot p \quad \Rightarrow \quad S_p = p$

Paralleler vs. serieller Anteil – Effizienz

Effizienz (Alternativ zum Speedup)

Die Effizienz eines parallelen Programmes ist definiert als

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$$

- Maß für den Anteil der Laufzeit, den ein Prozessor für Berechnungen benötigt, die auch im sequentiellen Programm vorhande sind.

  $\Rightarrow$ kleine Effizienz = hoher paralleler Overhead
- Idealer/linearer Speedup $S_p = p$ entspricht einer Effizienz $E_p = 1$
- $\varepsilon_{100}(n) = 0.4$ bedeutet, dass jeder der 100 Prozessoren 60% mit Kommunikation verbringt.

Amdalsches Gesetz – Einbeziehung sequentieller Anteile

Im allgemeinen hat ein paralleler Algorithmus immer auch inhärente sequentielle Anteile, welche auch dementsprechend ausgeführt werden müssen. Dies hat Auswirkungen auf den Erreichbaren Speedup.

Wir setzen an:

- f relativer Anteil serieller Anwendungen
- $1 - f$ relativer Anteil ideal-parallelisierbarer Anwendungen

Ahmdalsches Gesetz

Wenn bei einer Implementierung ein **relativer Anteil** f $(0 \leq f \leq 1)$ sequentiell ausgeführt werden muss, dann setzt sich die Laufzeit der parallelen Implementierung zusammen, aus der

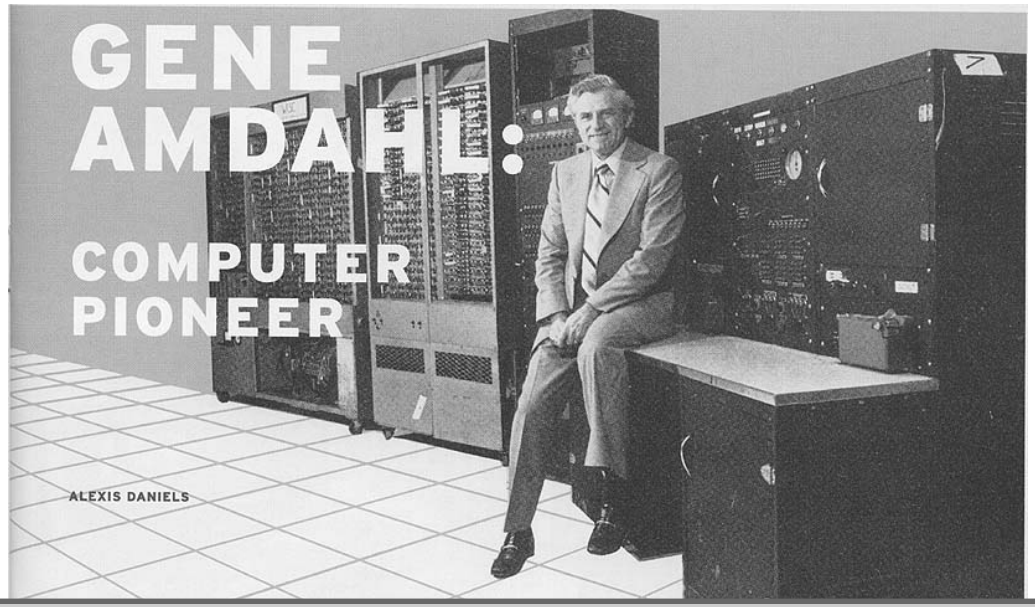
- Laufzeit $f \cdot T^*(n)$ des sequentiellen Anteils und der
- Laufzeit $(1 - f)/p \cdot T^*(n)$ des parallelisierbaren Anteils.

Amdahlsches Gesetz – Speedup

Für den Speedup gilt

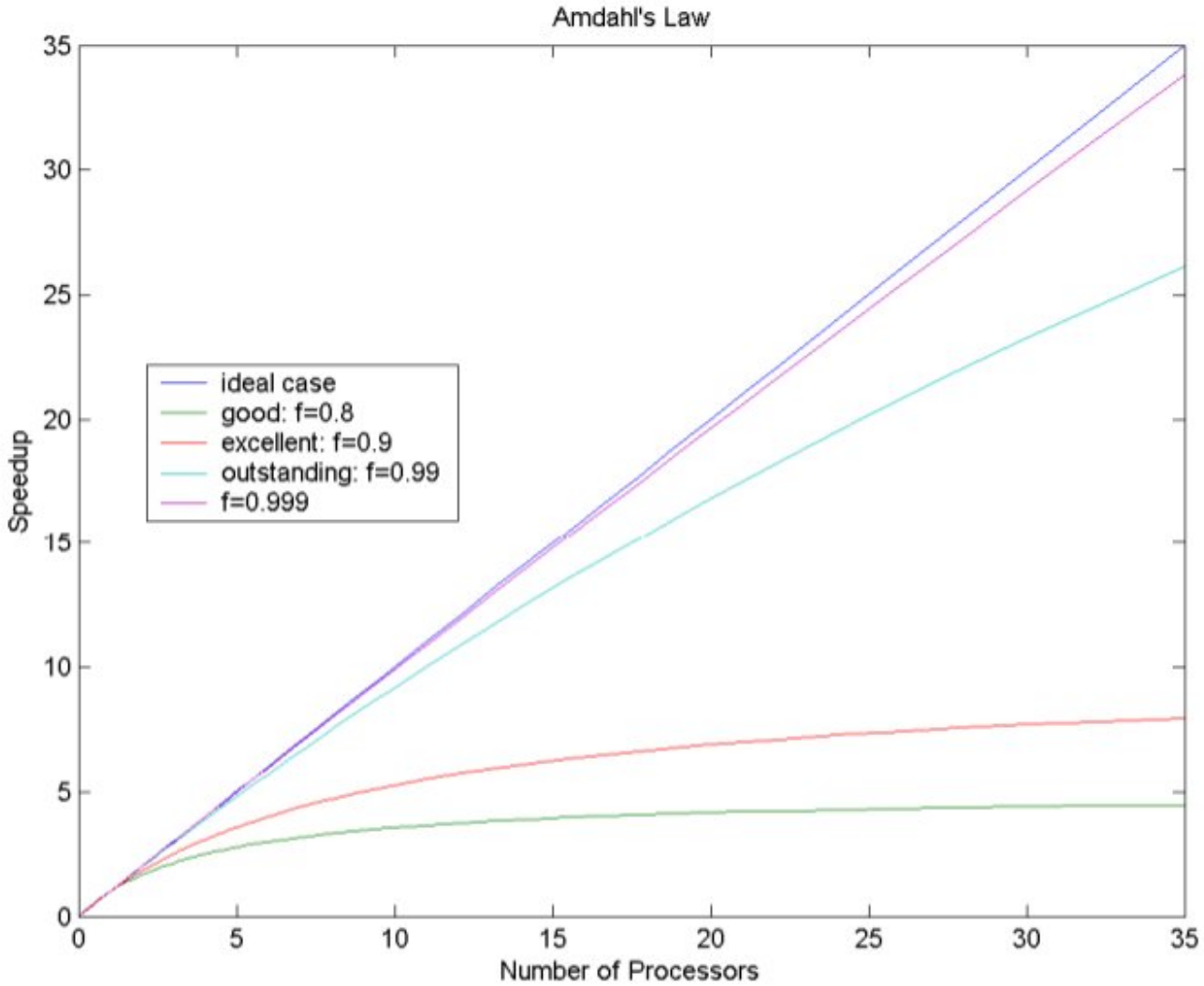
- f relativer Anteil sequentieller Anwendungen
- $1 - f$ relativer Anteil ideal-parallelisierbarer Anwendungen

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



Gene Ahmdahl (geb. 1922) Computer-Architekt und Hi-Tech- Unternehmer. Zu internationaler Bekanntheit kam er durch seine Arbeit im Bereich der Großrechner bei IBM. Gründer der Amdahl Corporation.

Amdahlsches Gesetz – Beispiel



Amdahls Gesetz – Konsequenz

Konsequenz

- Der serielle Anteil bleibt von massiver Parallelität unberührt.
- Bei einem sequentiellen Anteil von 1 %, beträgt der maximale Speedup gleich 100, egal wieviel Prozessoren man einsetzt.
- Bei einem sequentiellen Anteil von 20 %, beträgt der maximal erreichbare Speedup 5, egal wieviel Prozessoren man einsetzt.

Aus diesem Argument wurde lange der Schluss gezogen, dass sich

massive Parallelität

nicht lohnt.

Kommunikation

Weitere Quellen der Beschränkung:

- Lastverteilung (load Balancing)
- Kommunikation (ev. Warten auf Daten/Prozesse)

- Berücksichtigung der paralleliserungsbedingte Kommunikationszeit $T_p^c(n)$.

$$\Rightarrow \qquad T_p(n) = T^*(n) \cdot [(1-f) + f/p] + T_p^c(n)$$

Im Allg. steigt $T_p^c(n)$ mit # p monoton an.

Kommunikation – Master-Worker

Annahmen

- Lastverteilung sehr gut
- Kommunikation zwischen **Master** und **Worker**

  $\Rightarrow$ $T_p^c(n)$ lineare Funktion von p.
- Ein Prozessor führt keine Selbstgespräche

  $\Rightarrow$ $T_2^c(n)$ minimale Kommunikationszeit (Master und Worker)

$$T_p^c(n) = T_2^c(n)(p-1)$$

d.h.

$$S(p) = \frac{1}{f-r+(1-f)/p+rp}$$

mit $r = \frac{T_c(2)}{T(1)}$ Verhältnis min. Kommunikationszeit zur seq. Rechenzeit.

Kommunikation – Maximaler Speedup

Für große p kann der Speedup nun sogar fallen

Maximaler Speedup:

$$S(p^\star) = \frac{1}{f - r + 2\sqrt{(1-f)r}}$$

bei $p^\star = \sqrt{\frac{1-f}{r}}$

Nur für

$$p \ll p^\star$$

ist $S(p) = p$, d.h. nur in diesem Bereich zahlt sich Parallelität aus.

Skalierung

Beispiel

- Ein Maler tapeziert ein Zimmer in einer Stunden
- Zwei Maler tapezieren ein Zimmer in einer halben Stunde
- Wie lange brauchen 60 Maler für das Zimmer?

Abhilfe:

- Nutze die 60 Maler für ein Hotel mit 60 Zimmern...

Skalierung von Problemen

Mit zunehmender # Prozessoren sollte auch die Problemgröße wachsen, um effizientes Arbeiten zu gewährleisten

Skalierbarkeit

Skalierbarkeit (scalability)

Nach dem Amdahlschen Gesetz tritt für eine feste Problemgröße n bei steigender Prozessoranzahl p eine Sättigung des Speedups ein.

(60 Maler in einem Zimmer...)

- Oft ist man im Scientific Computing daran interessiert, ein größeres Problem in gleicher Zeit zu lösen
- Wachsende Problemgröße n kombiniert mit steigender # Proz. p
- Statt einem Maler ein Zimmer, 60 Maler das ganze Hotel (60 Zimmer) renovieren lassen.
- Diese Verhalten/Speedup (wachsendes n) wird durch das Amdahlsche Gesetzt **nicht** erfasst!

Skalierter Speedup

Annahme

Der sequentielle Programmanteil eines parallelen Programms nimmt mit der Modellgröße ab. D.h. er bildet nicht einen konstanter Anteil der Gesamtberechnung wie beim Amdahlschen Gesetz.

- Für jede # Prozessoren p kann maximaler Speedup $S_p(n) \leq p$ erreicht werden
- und zwar durch entsprechnde große Modellgröße

Gustafsonsches Gesetz

Verhalten der Laufzeit T bei größerem Problem und entsprechend erhöhter Anzahl von Prozessoren

Gustafsonsches Gesetz

Einbeziehung der Modellgröße n, f sequentieller Anteil konstant

$$T_s(n) = f + n(1-f)$$

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

Skalierter Speedup (Gustafson):

$$S_p(n) = \frac{T_s(n)}{T_p(n)} = \frac{f+n(1-f)}{f+\frac{n(1-f)}{p}} = \frac{f+n(1-f)}{fp+n(1-f)}p, \qquad \lim_{n\to\infty} S_p(n) = p$$

Konstante Laufzeit

Will man die Laufzeit bei wachsender Modellgröße konstant halten so gilt $n = p$, d.h. bei doppelter Modellgröße (z.B. # Gitterpunkte) muß man auch die Anzahl der Prozessoren verdoppeln

Performanceanalyse

Amdahl bzw. Gustafson

Einfache Aussagen zum Verhalten eines Modells bei Änderung

- # Prozessoren p
- Modell/Problemgröße n

Weitere Analyse-Modelle

- PRAM-Modell (Parallel Random Access Maschine)
- BSp-Modell (Bulk-Synchronous Parallel)

Abstrakte Rechnermodelle zum Design und Performance-Analyse von parallelen Algorithmen. Hier nicht weiter behandelt.

Timing-Modell

$\alpha - \beta$-Modell (z.B. Van der Velde)

Einfaches Modell zur Laufzeitanalyse

Annahmen

- alle Task können ungestört voneinander verlaufen und gleichzeitg beginnen
- alle Prozessoren sind identisch
- alle arithmetischen Operationen benötigen die gleiche Zeit t_a
- Datenaustausch findet in Einheitswortlänge statt (16 Bit)
- Kommunikation und Berechnung überlappen nicht
- Kommunikation beeinflußt sich nicht gegenseitig störend
- keine globale Kommunikation, sondern nur Punkt-zu-Punkt Kommunikation

Timing-Modell – Modellierung Datenaustausch

Datenaustausch

Kommunikation zwischen Sender und Empfänger linear, d.h.

$$t_k(l) = \alpha + \beta \cdot m$$

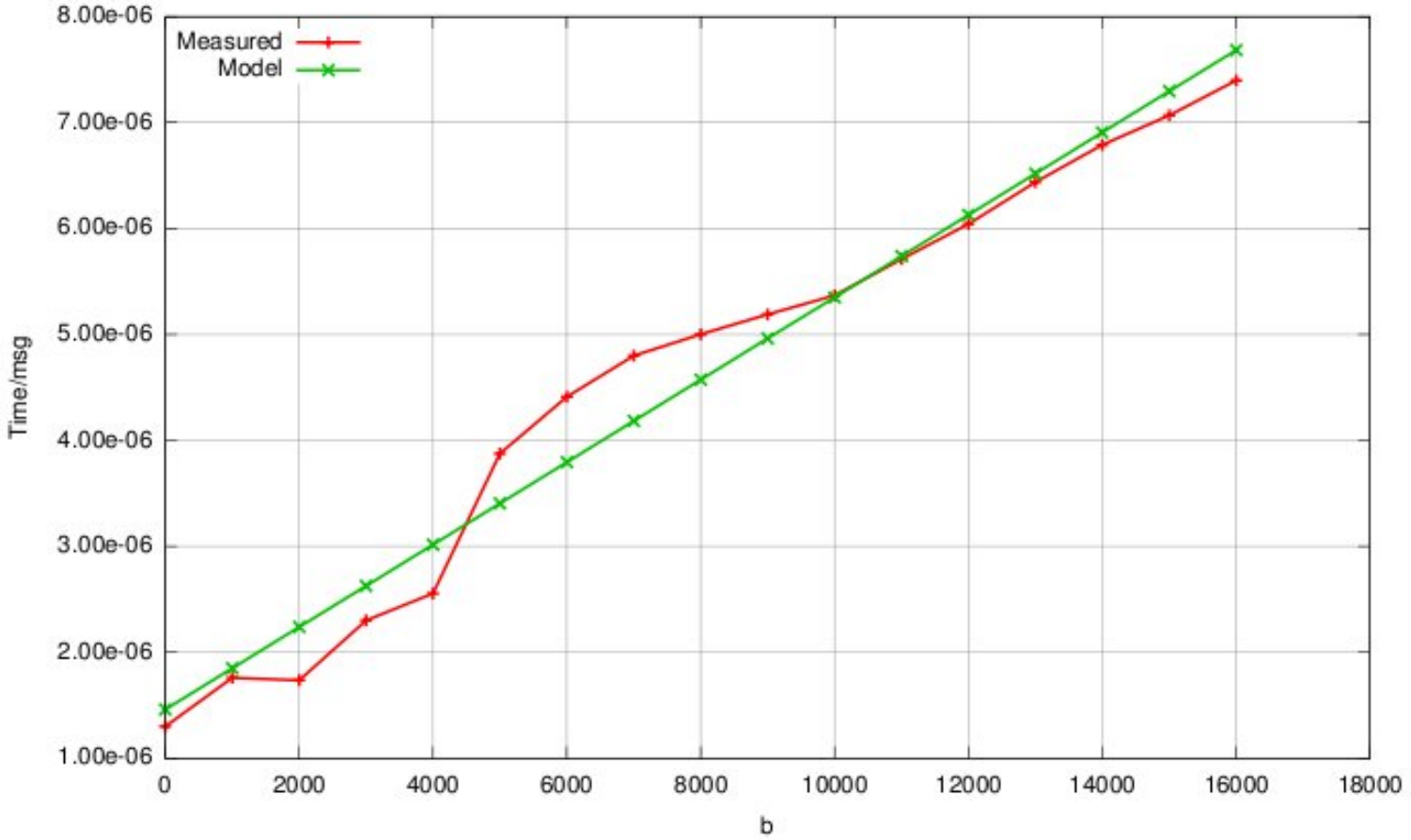
Eigenschaften

- α Startzeit (Latenzzeit)
  Zeit, welche im Kommunikationsnetz benötigt wird, um eine Datenpaket zu „schnüren“, mit den entsprechenden Adressinformationen u versehen, und wieder „auszupacken“.
- β reziproke Bandbreite des Busses (Kommunikationsnetzwerkes)
- m zu versendende Nachricht

Allgemein gilt $\alpha \gg \beta \gg t_a$

Kommunikationszeit hängt linear von der Nachrichtenlänge ab.
Steigung der Geraden ist die reziproke Bandbreite.

Timing-Modell – Beispiel



$$\alpha \approx 1.46 \times 10^{-6}, \beta \approx 3.89 \times 10^{-10}$$

$\alpha - \beta$-Parameter (Laptop 2 Kerne)

Speicherbandbreite

Speicherzugriffszeiten

Ggf. ist die Speicherbandbreite zu berücksichtigen, d.h. die Zeit, welche für Speicherzugriffe auf Datenelemente benötigt wird.
(z.B. auf Vektorkomponenten).

Die Zeit für Speicherzugriffe hängt von der Schrittweite/Inkrement s ab:

- Kleine Schrittweiten ($s \leq 16$)
  linearer Ansatz: $\frac{t_m(s,m)}{m \cdot t_a} \approx 2s \quad \Rightarrow \quad t_m(s,m) = \alpha + \beta \cdot s \cdot m$
- Große Schrittweiten
  asymptotisches Verhalten: $\frac{t_m(\infty,m)}{m \cdot t_a} > 160 \Rightarrow t_m(s,m) = (\alpha + \beta) \cdot m$

Liegen die Daten kompakt vor?

Moderne Speicher sind auf kontinuierliches Speicherbelegung ausgelegt.
Bei festen Inkrement veringert sich die Bandbreite.

Beispiel: Summation zweier Vektoren

Parallelisierung der Schleife

```
for(i=0;i<n;i++) z[i] = a*x[i] + b*y[i];
```

Zeit auf einem Prozessor

$$T_1 = 3 \cdot N \cdot t_a$$

Parallelisierung

- Zerlegung der Vektoren in Teilabschnitte n_p
- Idealwerweise liegen alle Daten/Teildaten allen Prozessoren vor

  $\Rightarrow$ Rechenzeit Prozessor $3 \cdot n_p \cdot t_a$

Parallele Zeit auf dem Gesamtsystem

Mit $N = \max_p n_p$ können wir die Gesamtzeit angeben:

$$T_p = 3 \cdot N \cdot t_a$$

Beispiel: Summation zweier Vektoren II

Rechenzeit bzw. Terminierung

Hängt ab von der Aufteilung n auf die Anzahl der Prozessoren p.
(Vorausetzung: Alle Prozessoren starten gleichzeitig)
Aufteilung: Gleichmäßig, eventuellen Divisionsrest entsprechend verteilen.

Speedup

$S(p) = \frac{T_1}{T_p} = \frac{n}{N}$ und für $N = n/p$ idealen Speedup $S(p) = p$

Beispiele

- $n = 3.000.001, \quad p = 16, \quad n_p = 187.500,0625, \; N = 187.501$
  $\Rightarrow$ $\mathbf{S(16) \approx 16}$

  aber
- $n = 33, \quad p = 16, \quad n_p = 2,0625, \; N = 3 \quad \Rightarrow$ $\mathbf{S(16) = 11}$

Lastverteilung und Granularität

Lastverteilung (load balancing)

Die Effektivität eines parallelen Programms hängt von der möglichst gleichmäßigen Verteilung der Arbeit auf die einzelnen Prozessoren ab.

Paralleles Rechnen bedeutet (zumindestens für distributed memory maschines), dass man sich Gedanken über die Aufteilung und Zuordnung der Daten machen muß.

Granularität

Die Größe eines Teilproblems pro Prozessor bestimmt die optimale bzw. maximale Prozessorzahl, welche sinnvoll eingesetzt werden kann.

grobgranular: Großes Problem, rel. wenig Prozessoren
$\Rightarrow$ wenig Kommunikation nötig (distributed memory)

feingranular: Kleine Probleme, rel.viele Prozessoren
$\Rightarrow$ viel Kommunikation nötig (shared memory)

Beispiel: Innere Produktbildung

Parallelisierung der Schleife

```
for(s=0.,i=0;i<n;i++) s += x[i] * y[i];
```

s soll auf allen Prozessoren bekannt sein.

Wichtig:

- Aufteilung der Daten
- Aufteilung der Arbeit
- Algorithmus

Arbeit auf einem Prozessor

$$T_1 = 2 \cdot n \cdot t_a$$

Fragestellung

Wie kann s synchronisiert werden? (da: `s = s + x[i] * y[i];`

Beispiel: Innere Produktbildung – Variante 1

Variante 1

```
s=0.;
for(i=0;i<np;i++) s += x[i] * y[i];
for(i=0;i<p; i++) if(i != selbst) sende (s an Proz. i);
for(i=0;i<p; i++) if(i != selbst) s+= empfange (von Proz. i);
```

- lokale Summenbildung: $2 \cdot np \cdot t_a$
- Wert verschicken: $(p-1) \cdot t_k(1)$
- Wert empfangen: $(p-1) \cdot t_k(1)$

$$\Rightarrow \quad T_p = 2 \cdot N \cdot t_a + (p-1)t_k(1) + (p-1)t_a$$

Speedup

$$S = \frac{2n \cdot t_a}{(2N+p-1)t_a + (p-1)t_k(1)} = \frac{2n}{(2N+p-1) + (p-1)t_r} \quad \text{mit } t_r = t_k(1)/t_a.$$

Beispiel: Innere Produktbildung – Variante 1

Variante 1

```
s=0.;
for(i=0;i<np;i++) s += x[i] * y[i];
for(i=0;i<p; i++) if(i != selbst) sende (s an Proz. i);
for(i=0;i<p; i++) if(i != selbst) s+= empfange (von Proz. i);
```

- lokale Summenbildung: $2 \cdot np \cdot t_a$
- Wert verschicken: $(p-1) \cdot t_k(1)$
- Wert empfangen: $(p-1) \cdot t_k(1)$

$$\Rightarrow \quad T_p = 2 \cdot N \cdot t_a + (p-1)t_k(1) + (p-1)t_a$$

Speedup

$$S = \frac{2n \cdot t_a}{(2N+p-1)t_a + (p-1)t_k(1)} = \frac{2n}{(2N+p-1) + (p-1)t_r} \quad \text{mit } t_r = t_k(1)/t_a.$$

Beispiel: Innere Produktbildung – Variante 1

Variante 1

```
s=0.;
for(i=0;i<np;i++) s += x[i] * y[i];
for(i=0;i<p; i++) if(i != selbst) sende (s an Proz. i);
for(i=0;i<p; i++) if(i != selbst) s+= empfange (von Proz. i);
```

- lokale Summenbildung: $2 \cdot np \cdot t_a$
- Wert verschicken: $(p-1) \cdot t_k(1)$
- Wert empfangen: $(p-1) \cdot t_k(1)$

$$\Rightarrow \quad T_p = 2 \cdot N \cdot t_a + (p-1)t_k(1) + (p-1)t_a$$

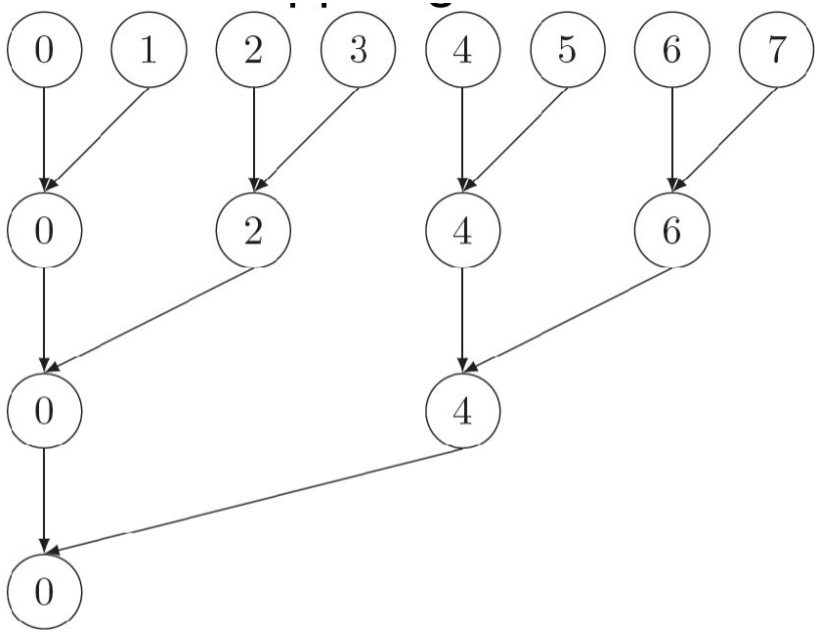
Speedup

$$S = \frac{2n \cdot t_a}{(2N+p-1)t_a + (p-1)t_k(1)} = \frac{2n}{(2N+p-1) + (p-1)t_r} \quad \text{mit } t_r = t_k(1)/t_a.$$

Beispiel: Innere Produktbildung – Variante 2

Variante 2 – Rekursive Verdopplung

Es werden paarweise rekursiv die lokalen Summen gebildet.



$$\Rightarrow \quad T_p = 2 \cdot N \cdot t_a + \log_2 p \cdot (t_a + 2t_k(1))$$

Speedup

$$S = \frac{2n \cdot t_a}{(2N + \log_2 p)t_a + 2\log_2 p \cdot t_k(1)} = \frac{2n}{(2N + \log_2 p) + 2\log_2 p \cdot t_r} \quad \text{mit } t_r = t_k(1)/t_a.$$

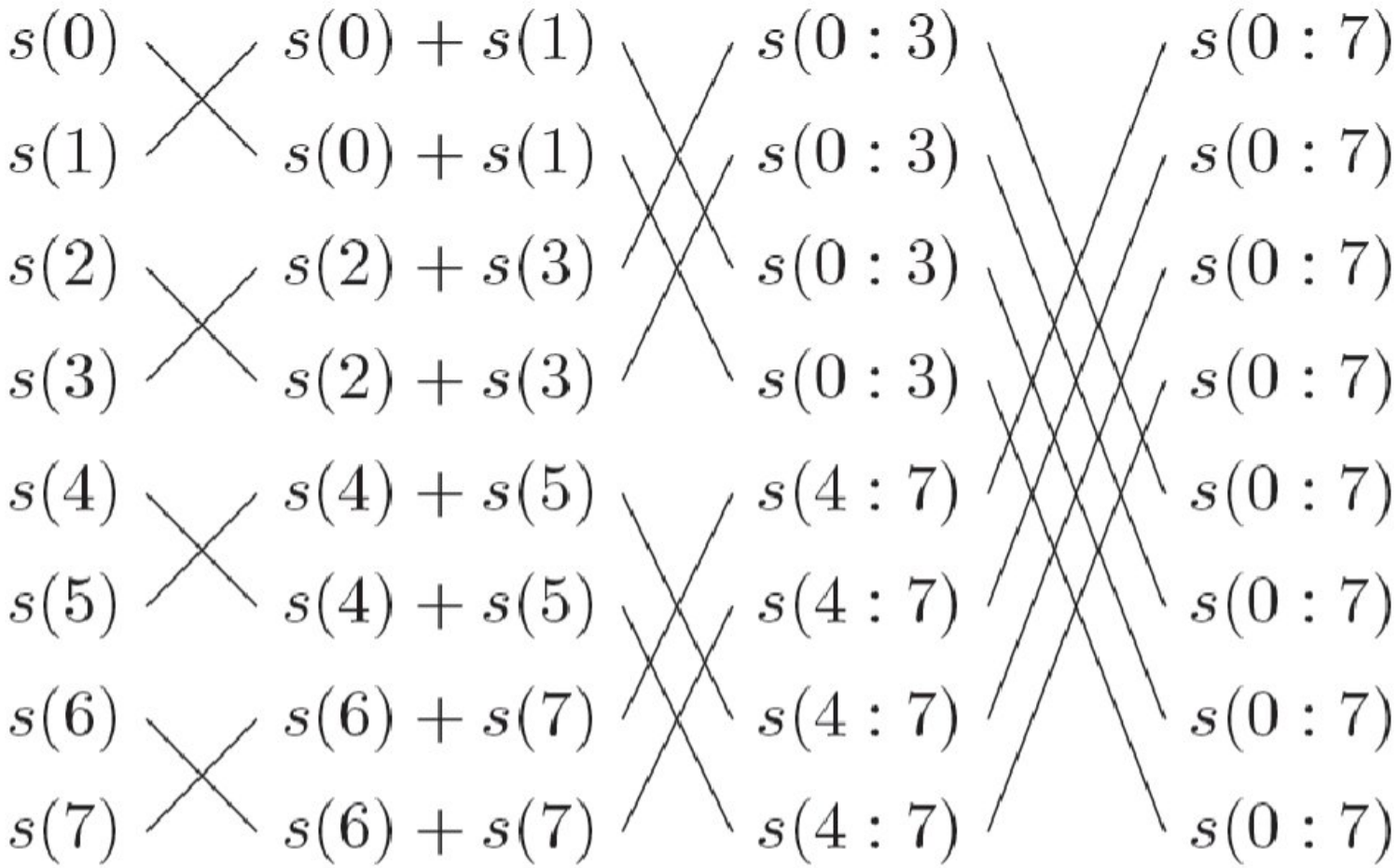
Beispiel: Innere Produktbildung – Variante 3

Variante 3 – FFT-like

```
s=0.;
for(i=0; i<np; i++) s += x[i] * y[i];
for(i=0; i<(int)log2(p); i++) {
   k = 1<<i;   /*Bit-Verschiebung nach links = Folge 1, 2, 4,
   j = myself^k;    /*Exklusives Oder */
   sende (s an Proz. j);
   s += empfange von Proz. j)
}
```

- $\log_2(p)$-Schritte
  - Jeweils p unterschiedliche Paare tauschen s aus
  - Jeder Prozessor bildet Teilsumme
- Algorithmus aus der schnellen Fourier-Transformation (FFT) bekannt.

Beispiel: Innere Produktbildung – Variante 3



$s(0:3)$ bedeutet hier: $\sum_{i=0}^{3} s(i)$

Beispiel: Innere Produktbildung – Variante 3

Variante 3 – FFT-like

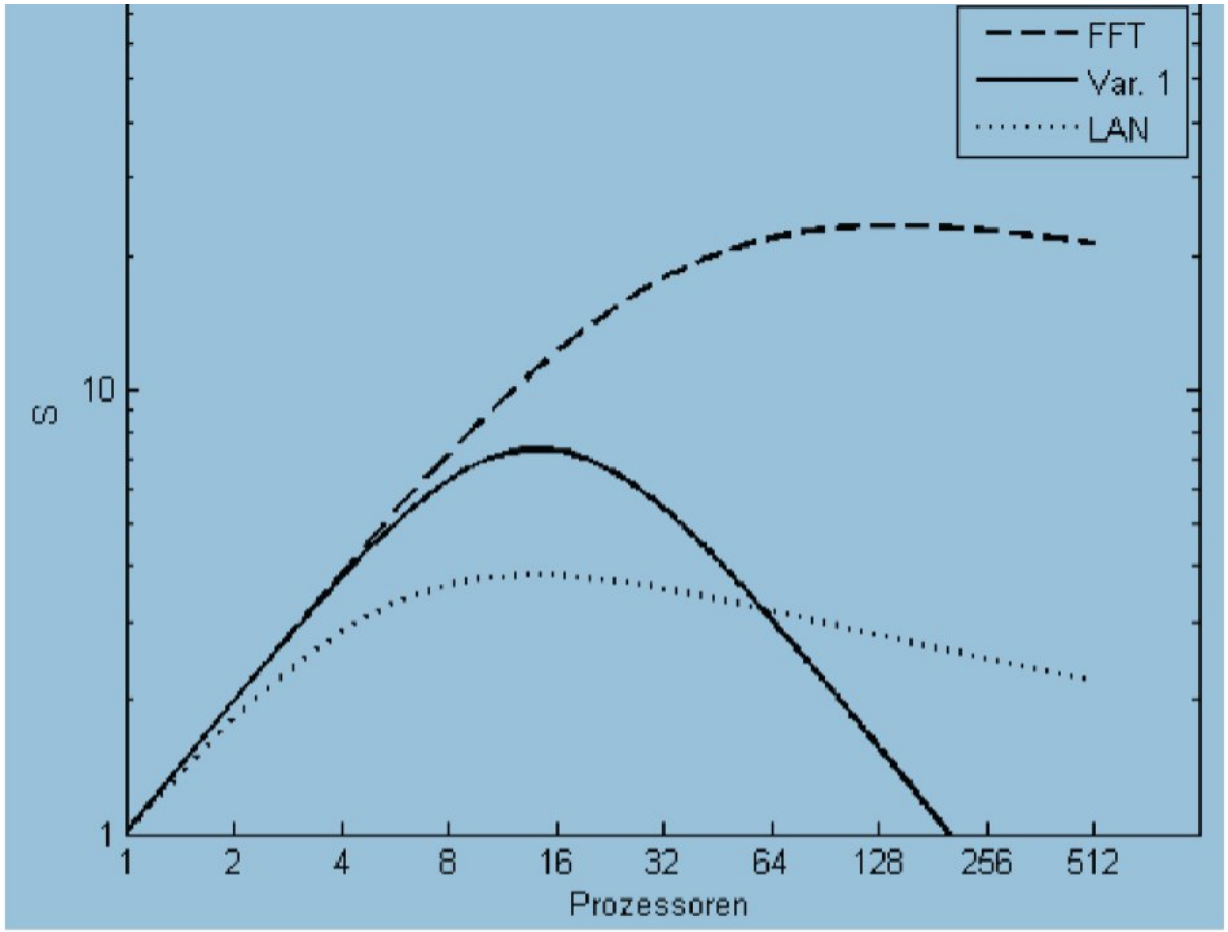
$$\Rightarrow \quad T_p = 2 \cdot N \cdot t_a + \log_2 p \cdot (t_a + t_k(1))$$

- logarithmische Abhängigkeit
- halbe Kommunikationszeit wie rekursive Verdopplung, falls das Ergebnis überall bekannt sein muss.

Speedup

$$S = \frac{2n \cdot t_a}{2N \cdot t_a + \log_2 p (t_a \cdot t_k(1)} = \frac{2n}{2N + 2\log_2 p \cdot (1 + t_r} \quad \text{mit } t_r = t_k(1)/t_a.$$

Beispiel: Innere Produktbildung – Vergleich



Vergleich der Varianten 1 und 3 mit $N = 100.000$

LAN: FFT mit LAN ($t_r = t_k/t_a \approx 10^5$) sonst schneller: $t_r = t_k/t_a \approx 10^3$)

Beispiel: Innere Produktbildung – Fazit

Betrachtete Algorithmen Innere Produktbildung

- Einfache Performancemodelle hilfreich
- **Variante 1**

  lineare Abhängigkeit der Rechenzeit von Prozessoranzahl p verhindert massive Parallelität
- **Variante 2** (rekursive Verdopplung)

  logarithmische Abhängigkeit der Rechenzeit ermöglicht besseren Speedup
- **Variante 3** (FFT-artiges Schema)

  zusätzlich geringere Kommunikationsaufwand