



Kapitel 4

Leistungsaspekte

4.1 Das einzelne Programm

- Sei
 - $T(1)$ benötigte Programmbearbeitungszeit auf einem Einprozessorsystem
 - $T(p)$ benötigte Programmbearbeitungszeit auf einem n -Prozessorsystem
- Der Zeitgewinn durch Parallelverarbeitung wird ausgedrückt durch
 - $S(p) := T(1) / T(p)$ Geschwindigkeitsfaktor (*Speed-up*)
- Normiert man den Speed-up auf die Anzahl eingesetzter Prozessoren p , so entsteht die sogenannte *Effizienz*:

$$E(p) := S(p) / p \quad \text{Effizienz}$$

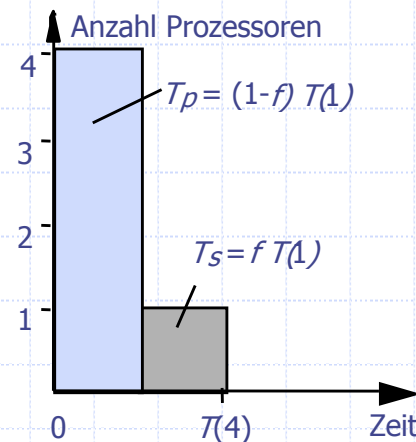
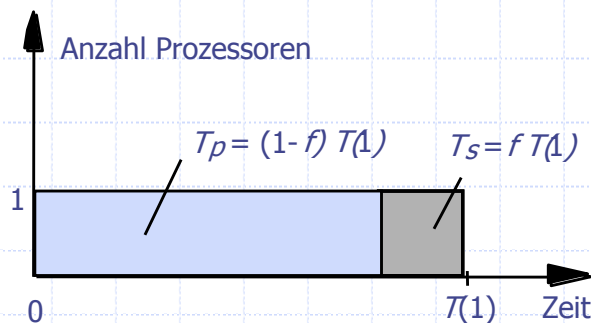
Amdahls Gesetz

- Parallele Programme enthalten auch sequentielle Anteile.
- Zerlegung der Bearbeitungszeit in einen sequentiellen und einen parallelen Teil ergibt:

$$T(1) = T_s + T_p$$

- Sei $f := 1 / (T_s + T_p)$, ($0 \leq f \leq 1$) der sequentielle Anteil eines Programms. Dann gilt:

$$T(p) = fT(1) + \frac{(1-f)T(1)}{p} = T_s + \frac{T_p}{p} \quad (\text{Amdahls Gesetz})$$



- Mit Amdahls Ansatz erhalten wir für den Geschwindigkeitsfaktor:

$$S(p) = \frac{T(1)}{T(p)} = \frac{p}{1 + f(p-1)} = \frac{1}{f + \frac{1-f}{p}}$$

- und für die Effizienz

$$E(p) = \frac{S(p)}{p} = \frac{1}{1 + f(p-1)}$$

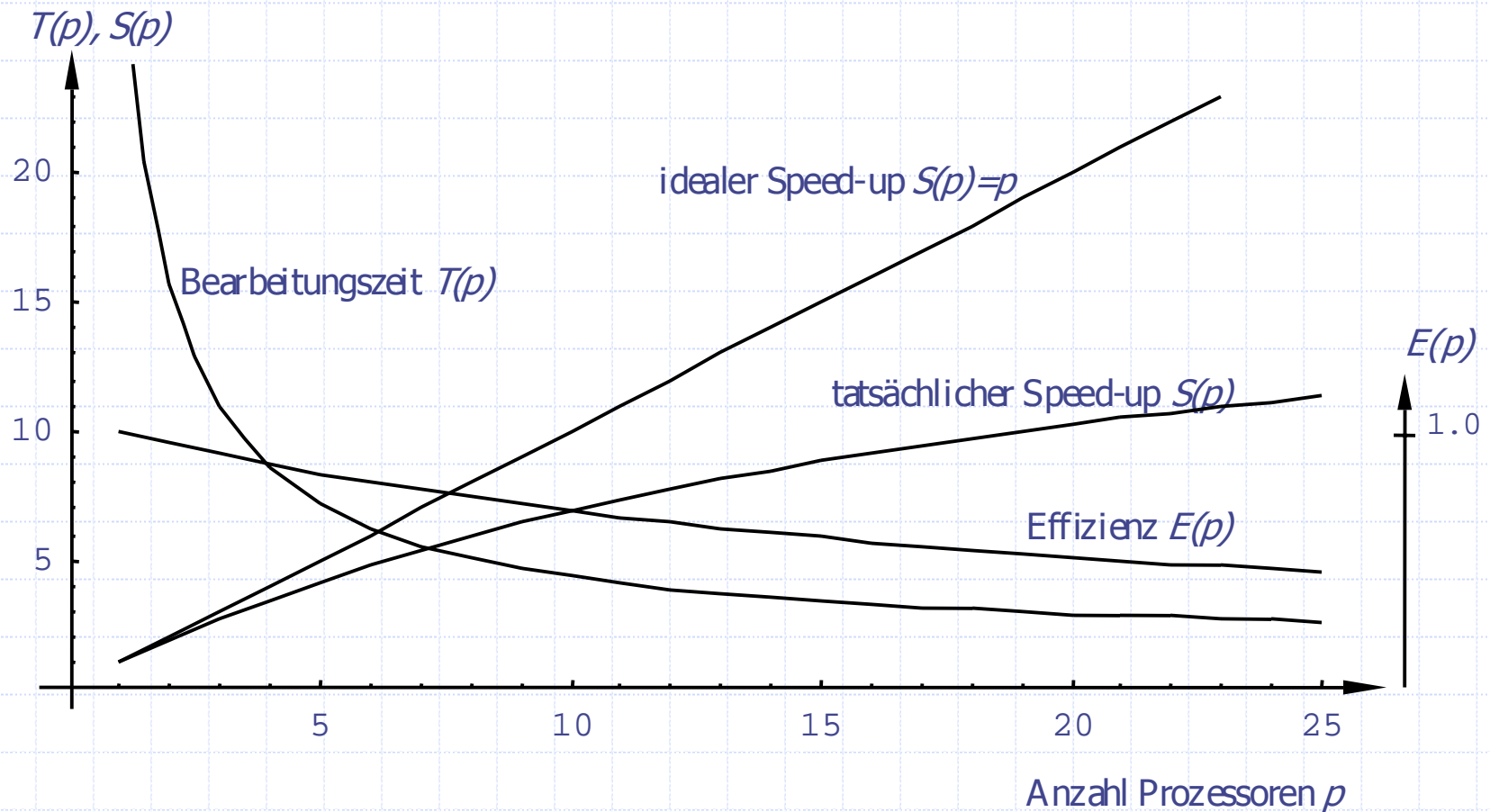
- Es gilt: $T(\infty) := \lim_{p \rightarrow \infty} T(p) = fT(1) = T_s$

$$S(\infty) := \lim_{p \rightarrow \infty} S(p) = 1/f$$

$$E(\infty) := \lim_{p \rightarrow \infty} E(p) = 0$$

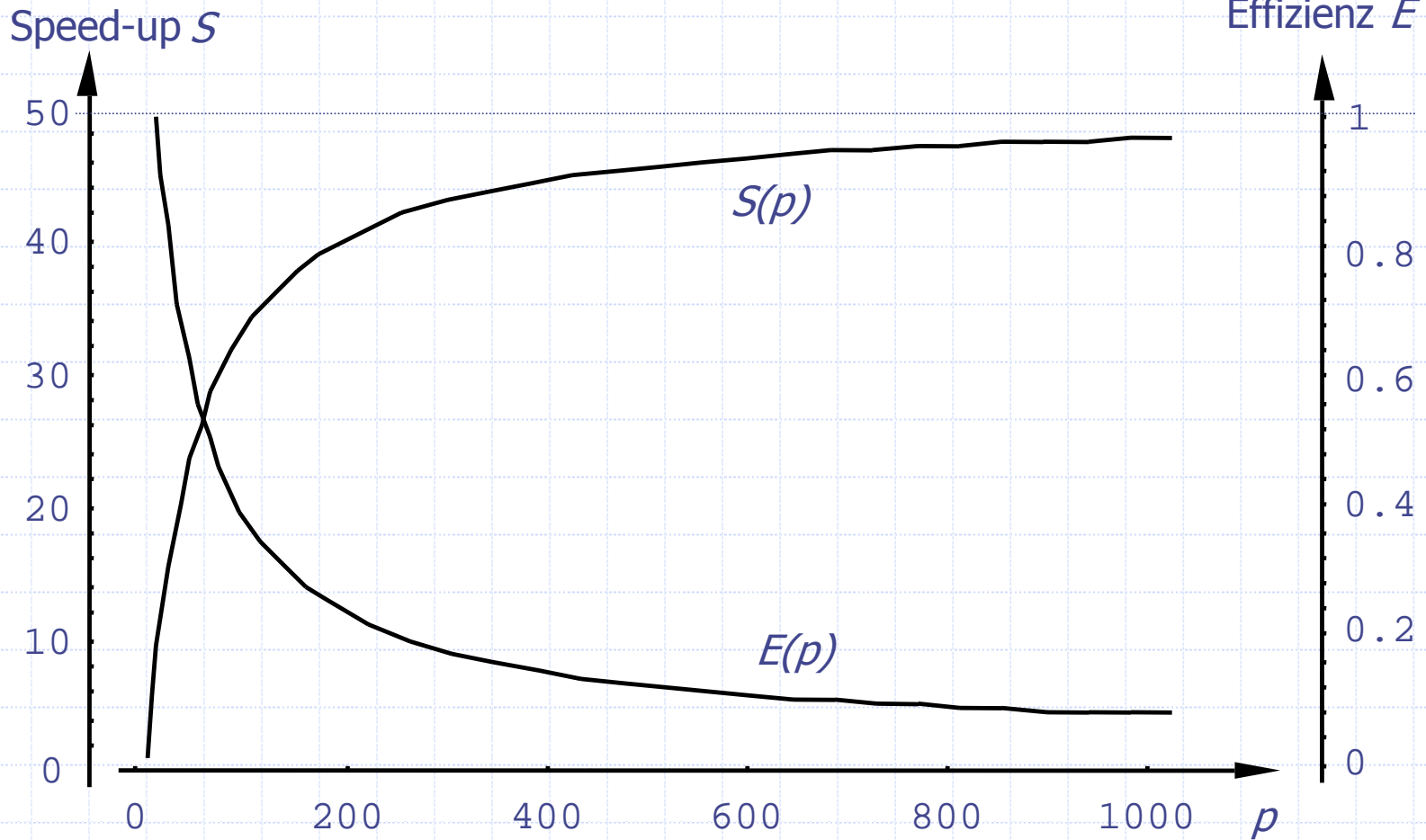
Verhalten der verschiedenen Größen unter Amdahls Modell

für $T(1)=30$ und $f = 0.05$.



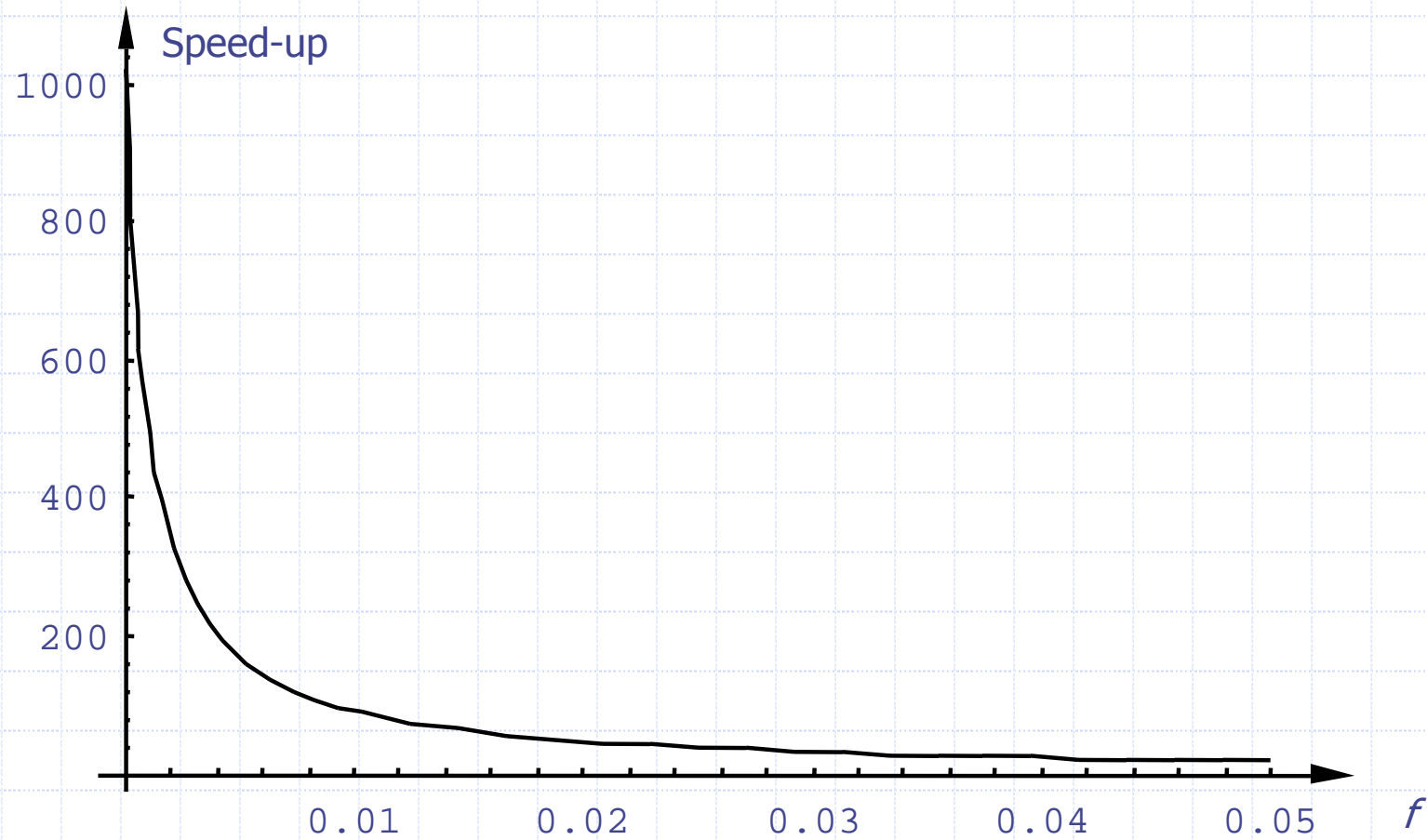
Speed-up S und Effizienz E als Funktionen der Anzahl Prozessoren p

für $f = 0.02$



Speed-up als Funktion des sequentiellen Anteils f

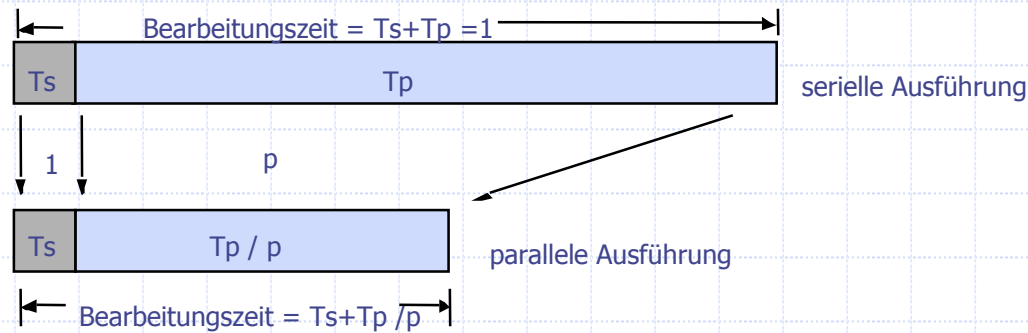
bei $p = 1024$.



Problemgrößenfaktor (Scale-up)

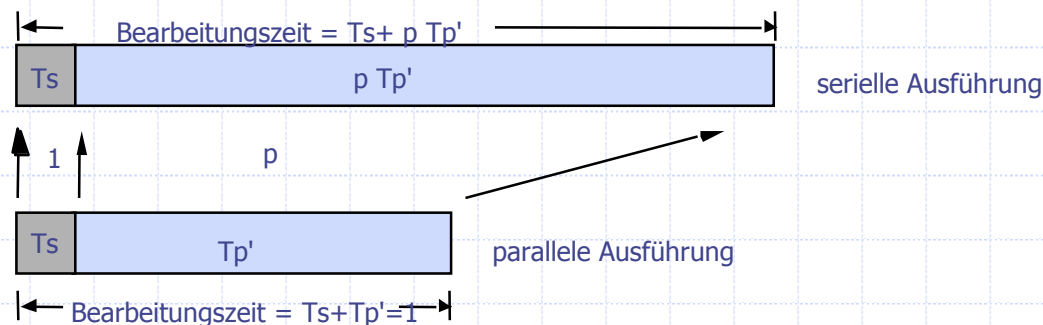
Speed-up:

Um wieviel schneller kann ein gegebenes Problem mit p Prozessoren gelöst werden



Scale-up (scaled Speed-up):

Um wieviel größer kann ein Problem sein, wenn es in gleicher Zeit mit p Prozessoren gelöst wird

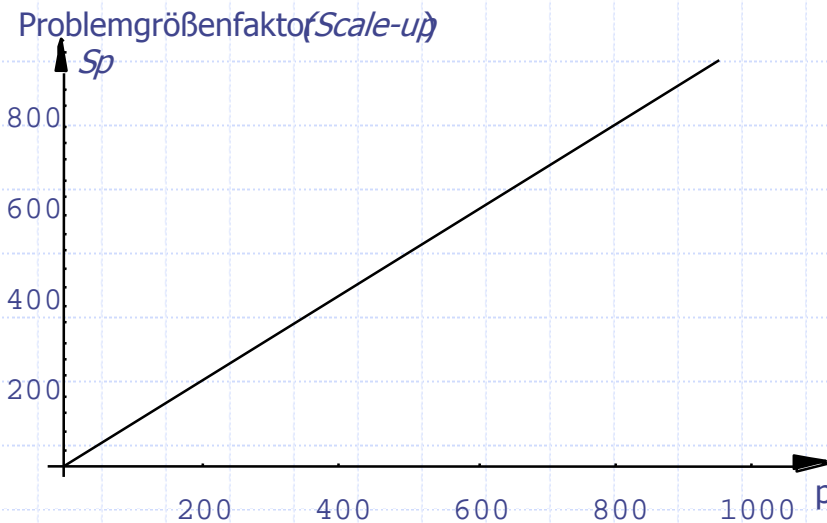


Wir gehen für den Scale-up von einer auf 1 normierten parallelen Bearbeitungszeit aus: $T(p) = f + (1 - f) \cdot p = 1$

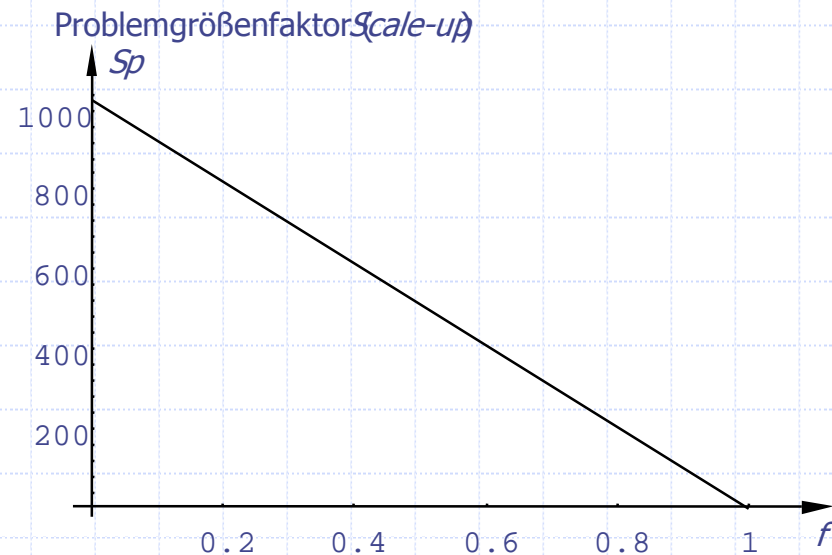
Bei nur einem Prozessor muss der parallele Anteil sequentiell ausgeführt werden, und wir erhalten $T(1) = f + (1 - f) \cdot 1$

Dadurch ergibt sich ein Problemgrößenfaktor $S_p(p)$ (*scale-up* oder *scaled speed-up*) von

$$S_p(p) = T(1) / T(p) = f + (1 - f) \cdot p = p - (p - 1) \cdot f$$



S_p als Funktion von p bei $f=0,02$ bzw.



S_p als Funktion des sequentiellen Anteils f bei $p=1024$

- Bei der Ausführung eines Programms wird ein Problem einer bestimmten Größe gelöst.
- Die Problemgröße w kann aufgefasst werden als die Zahl der insgesamt auszuführenden Instruktionen (Funktion der Größe der Eingabedaten n)
- Ist t_c Zeit zur Ausführung eine Instruktion, dann gilt:

$$w t_c = T(1)$$

- Sei $T(p)$ die Laufzeit eines Programms auf p Prozessoren bei Problemgröße w
- Bei der Ausführung eines Programms bei Eingabe einer Problemgröße von w auf p Prozessoren wird insgesamt eine Arbeit von $p T(p)$ erbracht.
- Nur ein Teil dieses Aufwands ist „nützliche“ Arbeit, der Rest ist „Overhead“ T_o .

$$T_o(p) = p T(p) - T(1) \text{ bzw.}$$

$$T(p) = (T(1) + T_o(p)) / p$$

- Der Speed-up ist dann

$$S(p) = \frac{T(1)}{T(p)} = \frac{pT(1)}{T(1) + T_o(p)}$$

- Die Effizienz:

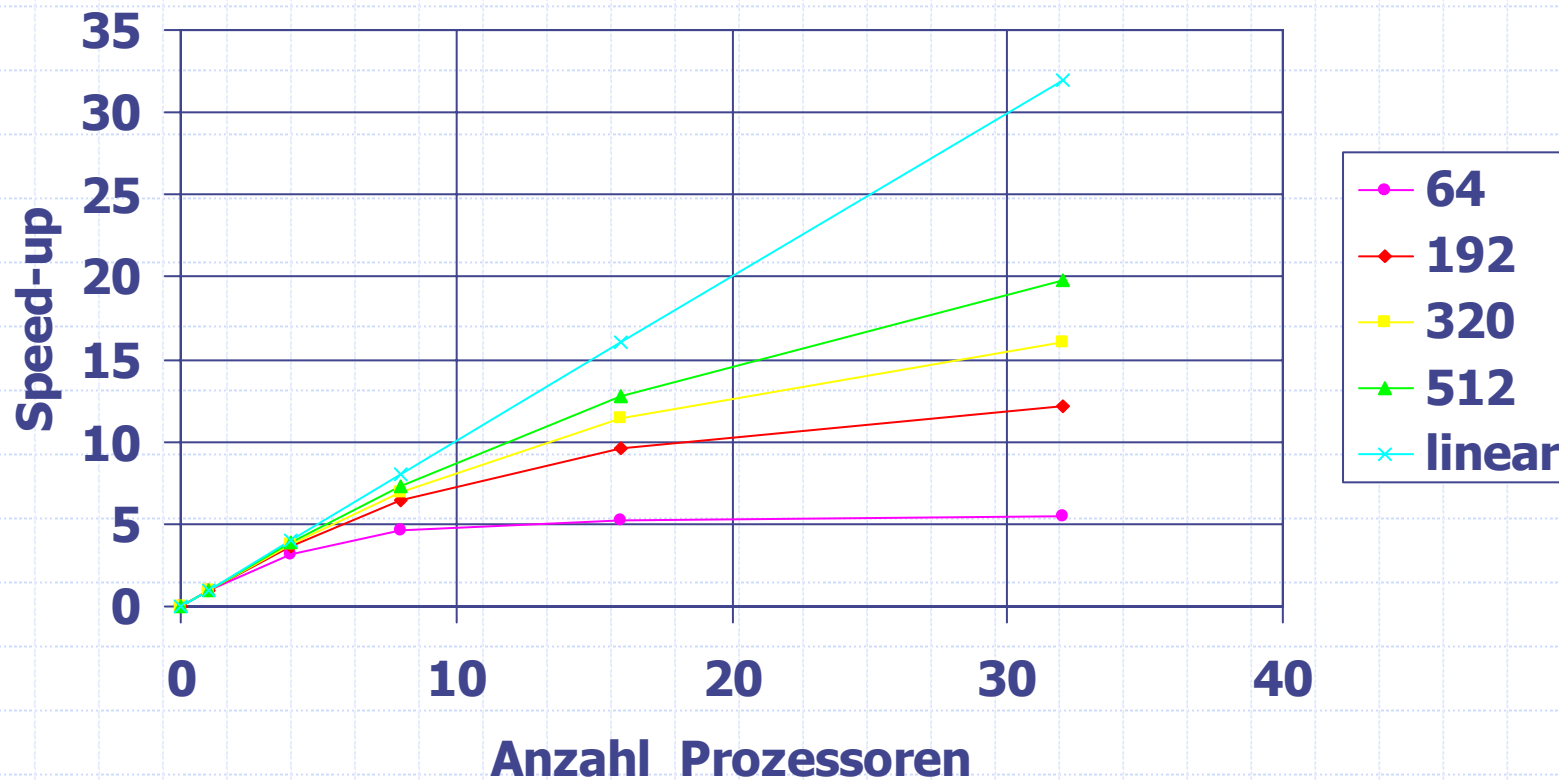
$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{T(1) + T_o(p)} = \frac{1}{1 + \frac{T_o(p)}{T(1)}}$$

- Die Effizienz hängt vom Verhältnis zwischen parallelem Overhead und sequentieller Laufzeit ab.
- Sie sinkt üblicherweise bei wachsender Prozessorzahl und steigt mit wachsender Problemgröße.

- Unter Skalierbarkeit versteht man das Leistungsvermögen eines Systems bei wachsender Prozessoranzahl.
- Intuitiv kann man ein System als skalierbar bezeichnen, wenn seine Leistungsfähigkeit bei gleichzeitiger Erhöhung von Problemgröße und Prozessoranzahl steigt.
- Speed-up und Scale-up versuchen, die Skalierbarkeit eines parallelen Programms auf einer bestimmten Maschine zu messen.
- Die Diskussion um den Scale-up hat deutlich gemacht, dass man durch Erhöhung der Problemgröße die Effizienz steigern kann.
- Die Speed-up-Kurve charakterisiert die Skalierbarkeit eines Programms bei einer bestimmten Eingabe.
- Üblicherweise flacht die Speed-up-Kurve ab, aber der Grenzwert wächst mit der Problemgröße.

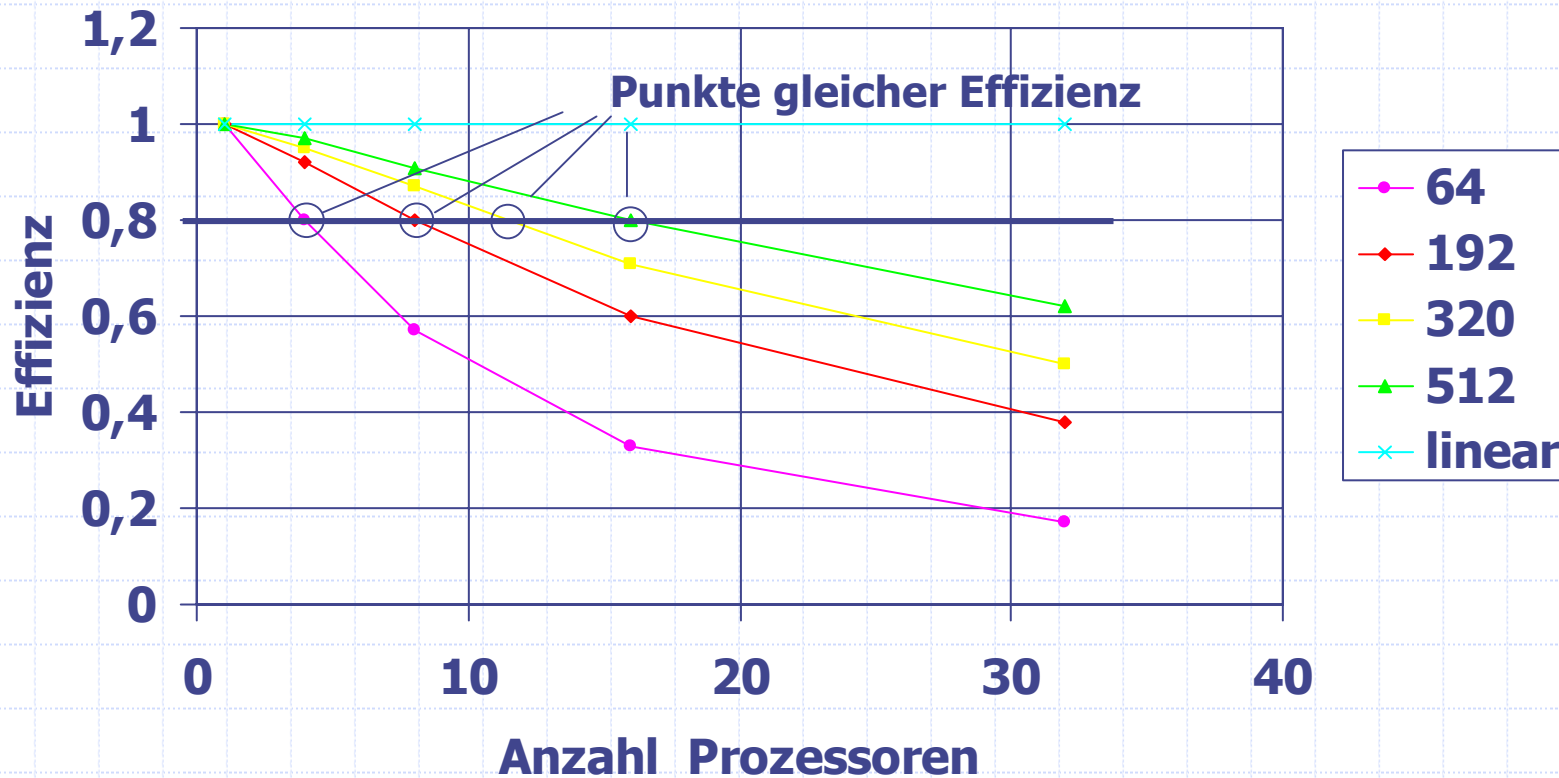
Beispiel

Paralleles Addieren von $n=64, 192, 320, 512$ Zahlen in einem Hypercube mit $p=1, 4, 8, 16, 32$ Prozessoren



Beispiel

Wieviel größer muss das Problem werden, dass eine bestimmte Effizienz (z.B. 0.8) gehalten wird?



- Setzen wir $T(1) = w t_c$ in die Gleichung aus 4-11 ein, so erhalten wir

$$E(p) = \frac{1}{1 + \frac{T_o(p)}{T(1)}} = \frac{1}{1 + \frac{T_o(p)}{w t_c}}$$

- Umformen und Auflösen nach w ergibt

$$w = \frac{1}{t_c} \left(\frac{E(p)}{1 - E(p)} \right) T_o(p)$$

- Setzen wir $K_E = E(p) / (t_c (1 - E(p)))$ als eine von einer bestimmten Effizienz abhängige Konstante, erhalten wir

$$w = K_E T_o(p)$$

- Dadurch ist ein funktionaler Zusammenhang zwischen der Problemgröße w und der Prozessoranzahl gegeben.
- Diese Funktion wird **Isoeffizienzfunktion** genannt und gibt an, wie stark die Problemgröße wachsen muss, um bei steigender Prozessoranzahl ein bestimmtes Effizienzniveau zu halten.

Eigenschaften der Isoeffizienz

- Gilt z.B. $w(p) = \Theta(p \log p)$ und soll die Prozessoranzahl von p auf p' erhöht werden, so muss w um einen Faktor von $(p' \log p') / (p \log p)$ vergrößert werden, um dieselbe Effizienz zu erzielen.
- Je geringer die asymptotische Komplexität der Isoeffizienz, desto besser die Skalierbarkeit, z.B.:
 - lineares Wachstum: gute Skalierbarkeit
 - exponentielles Wachstum: sehr schlechte Skalierbarkeit
- Beispiele:

Algorithmus	Isoeffizienz	Architektur
Faktorisierung dünn besetzter Matrizen	$\Theta(p \log^2 p)$	Clique
All-pairs-shortest Path (Dijkstra)	$O(p \log p)^{3/2}$	Hypercube
All-pairs-shortest Path (Dijkstra)	$O(p^{9/5})$	Gitter

Begrenzung des Parallelitätsgrads

- Programme lassen sich in der Regel nicht beliebig parallelisieren.
- Für jedes Programm gibt es einen maximalen Parallelitätsgrad

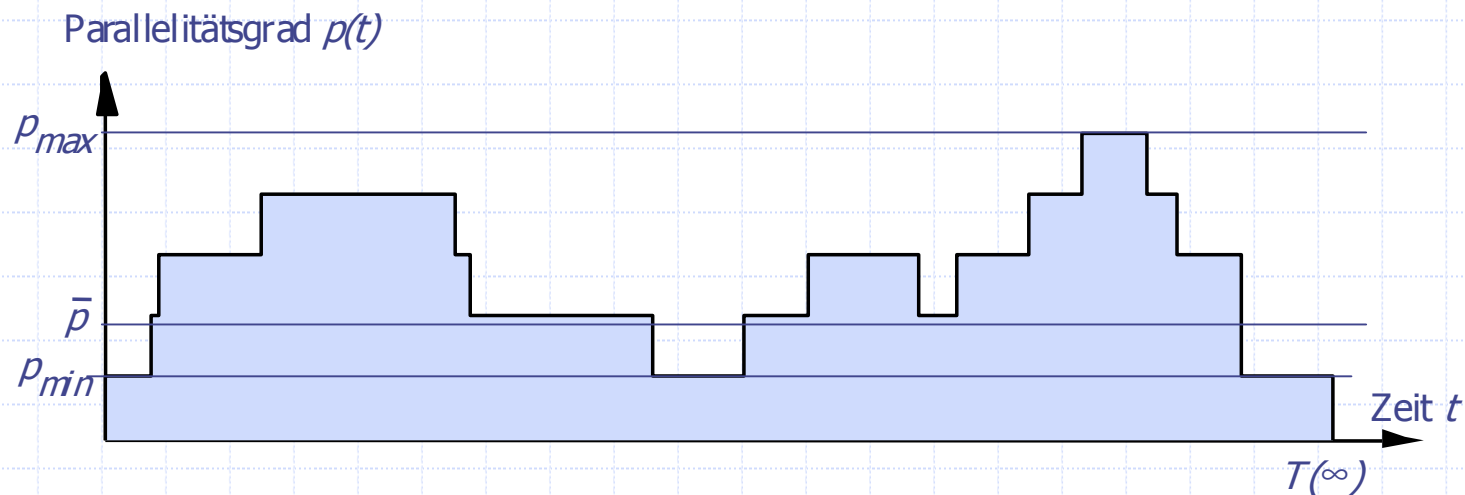
$$T(\rho) = \begin{cases} T_s + \frac{T_\rho}{\rho} & \text{für } \rho < \rho_{\max} \\ T_s + \frac{T_\rho}{\rho_{\max}} & \text{für } \rho \geq \rho_{\max} \end{cases}$$

- Insbesondere gilt dann

$$T(\infty) = \lim_{\rho \rightarrow \infty} (T(\rho)) = T_s + \frac{T_\rho}{\rho_{\max}}$$

Parallelitätsprofil eines parallelen Programms

- Zeichnet man den Parallelitätsgrad über die Zeitachse so erhält man das *Parallelitätsprofil*



- Man kann $p(t)$ interpretieren als die Anzahl von Prozessoren, die bei der Bearbeitung des Programms aktiv sind - unter der Annahme, dass beliebig viele Prozessoren zur Verfügung stehen.

Parallelitätsprofil

- Es gilt dann

$$\int_0^{T(\infty)} p(t) dt = T(1)$$

- d.h. die Fläche unter $p(t)$ gibt die benötigte Rechenzeit an und entspricht daher der Bearbeitungszeit im Einprozessorfal.
- Der *mittlere Parallelitätsgrad* lässt sich dann definieren als

$$\bar{p} = \frac{1}{T(\infty)} \int_0^{T(\infty)} p(t) dt$$

- Damit gilt:

$$\bar{p} = \frac{T(1)}{T(\infty)} = S(\infty) = \lim_{p \rightarrow \infty} S(p)$$

d.h. der asymptotische Speed-up ist gleich dem mittleren Parallelitätsgrad.

- Da $S(p)$ monoton wachsend ist, können wir unter Verwendung von Amdahls Gesetz nun schreiben:

$$S(p) \leq \min \left\{ \bar{p}, \frac{1}{f + (1-f)/p} \right\}$$

Aufwand durch Prozessinteraktion

- Die Prozesse als Teile eines parallelen Programms müssen interagieren.
- Der zeitliche Aufwand $T_C(p)$ (Index „C“ für „Communication“) für solche Interaktionsvorgänge ist eine streng monoton wachsende Funktion von p .
- Damit erhalten wir einen neuen, erweiterten Ansatz für die Bearbeitungszeit:

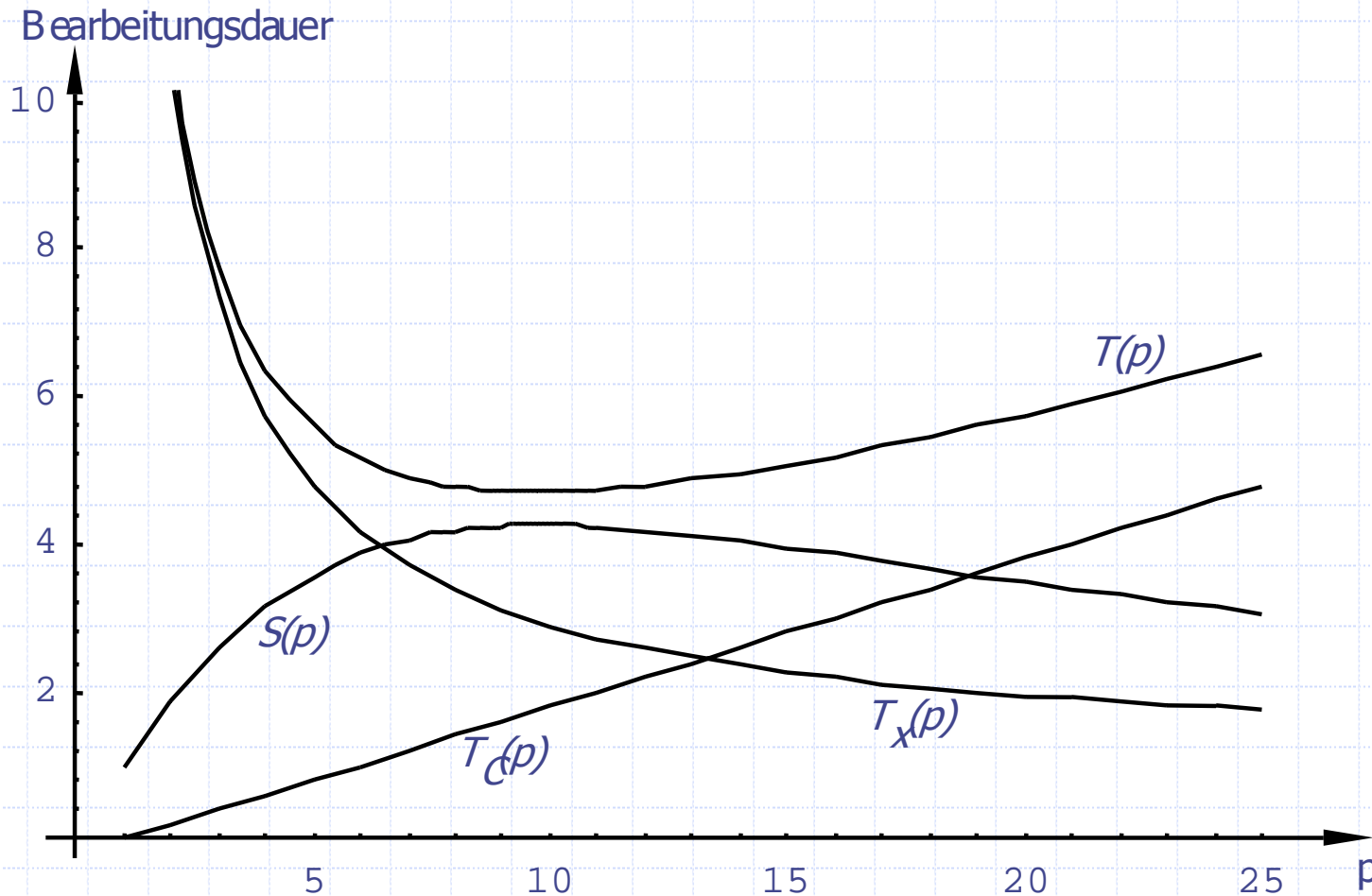
$$T(p) = T_x(p) + T_C(p)$$

$$\text{mit } T_C(1) = 0 \quad \text{und} \quad T_x(p) = \begin{cases} T_s + T_p / p & \text{für } p < p_{\max} \\ T_s + T_p / p_{\max} & \text{für } p \geq p_{\max} \end{cases}$$

als reine Rechenzeit des Programms (Index „x“ für „execution“).

- T_x ist eine monoton fallende, nach unten beschränkte Funktion.
- Ist T_C streng monoton wachsend und nicht nach oben beschränkt, dann muss $T(p)$ notwendigerweise von einem bestimmten p an steigen.

Zusammenwirken von Rechenzeit und Interaktionsaufwand



Bearbeitungszeit $T(p)$ zusammengesetzt aus Rechenzeit $T_x(p)$ und Interaktionsaufwand $T_c(p)$.

4.2 Mehrprogrammbetrieb in Parallelrechnern

Gegeben sei eine Menge paralleler Programme. Parallelität kann

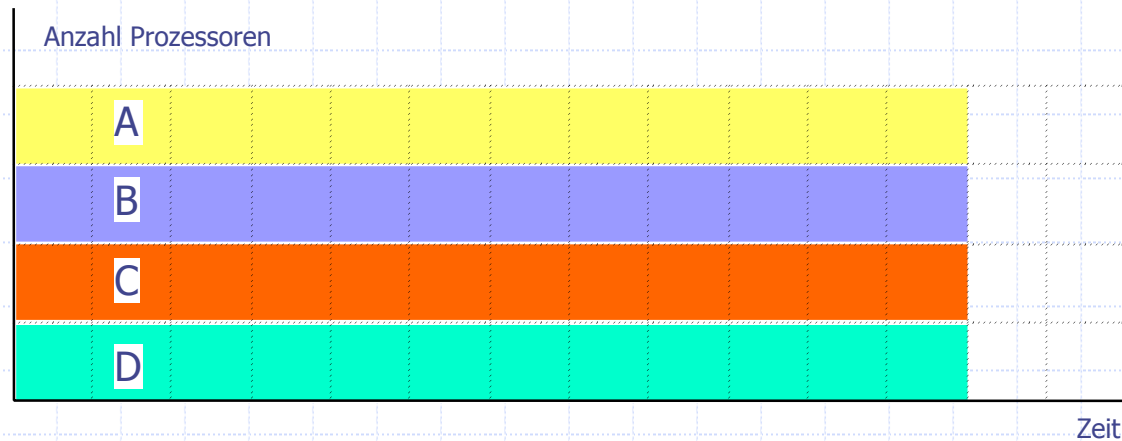
- (a) extern, d.h. zwischen den Programmen (*Interprogramm-Parallelität*)
und/oder
- (b) intern, d.h. innerhalb eines Programms (*Intraprogramm-Parallelität*)
stattfinden.

Dadurch ergeben sich die folgenden vier Kombinationen, wobei die ESIS Variante aufgrund fehlender Parallelität für Mehrprozessorsysteme uninteressant ist

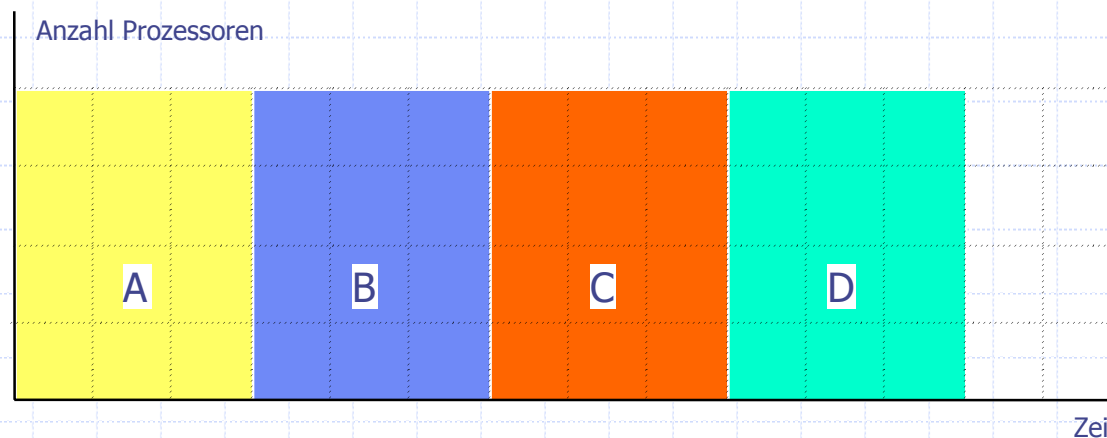
	intern sequentiell (Prozessebene)	intern parallel (Prozessebene)
extern sequentiell (Programmebene)	ESIS	ESIP
extern parallel (Programmebene)	EPIS	EPIP

Mehrprogrammbetrieb

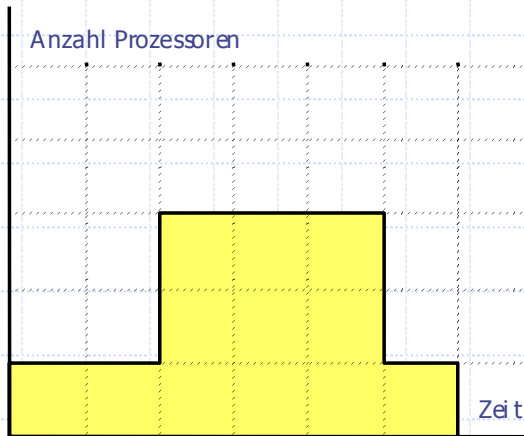
- EPIS: Parallele Bearbeitung in sich sequentieller Programme (ausschließlich Interprogramm-Parallelität)



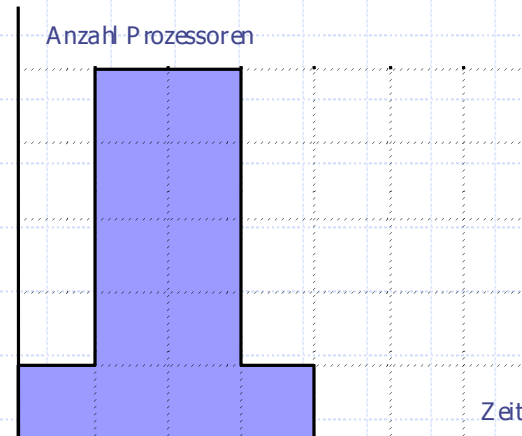
- ESIP: Sequentielle Bearbeitung in sich paralleler Programme (ausschließlich Intraprogramm-Parallelität)



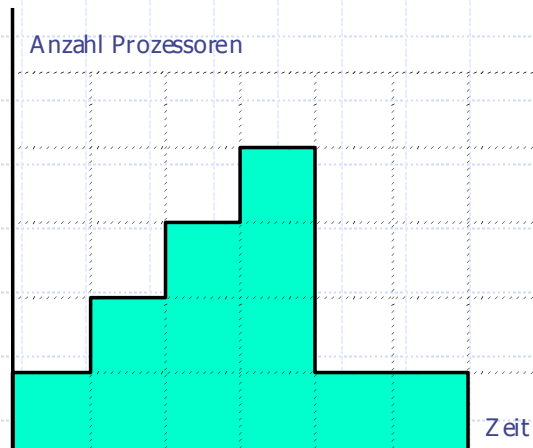
Externe und interne Parallelität



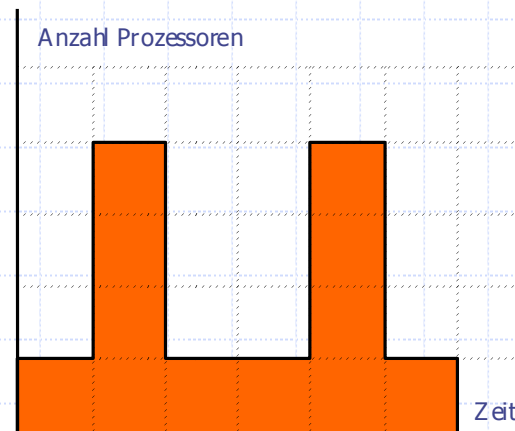
Programm A



Programm B

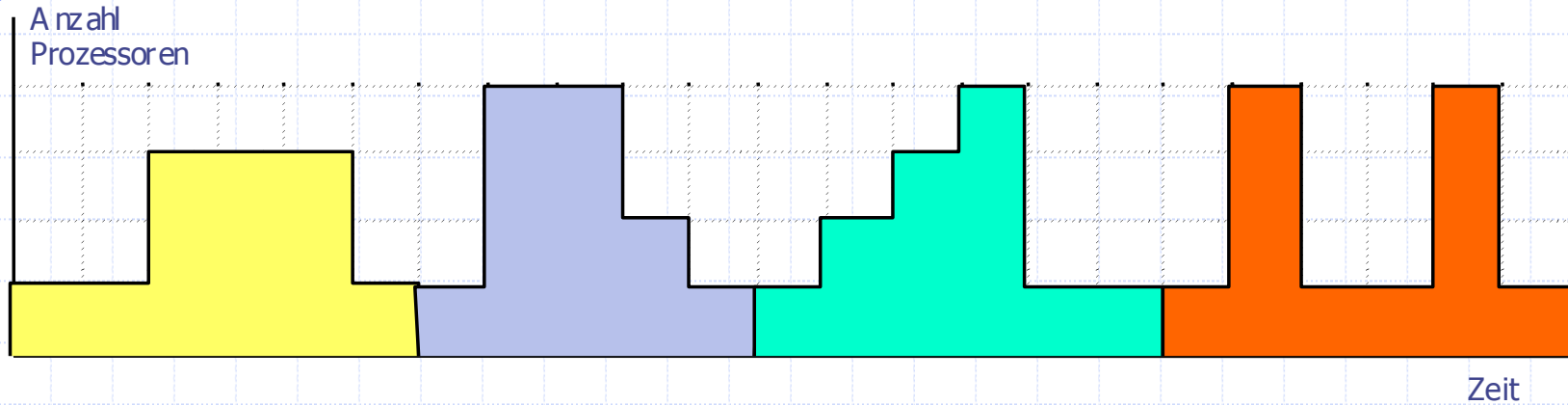


Programm C

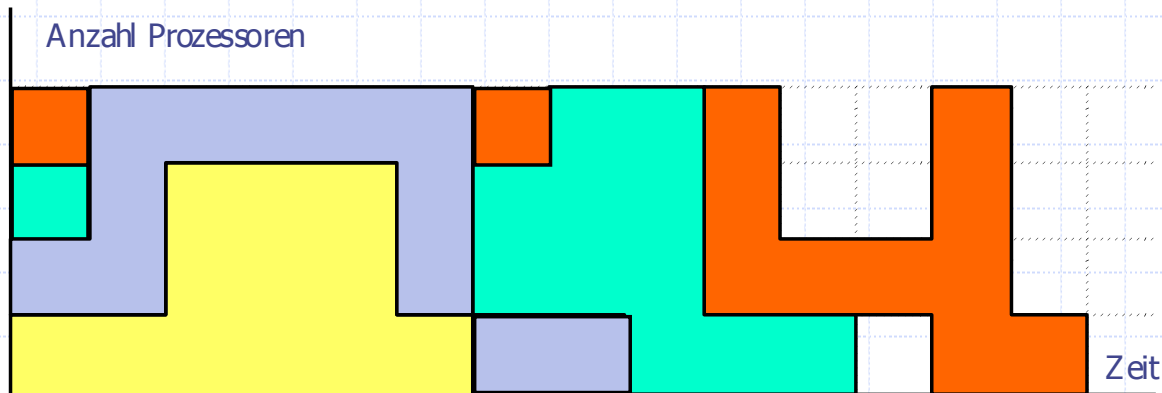


Programm D

Externe und interne Parallelität



ESIP: Verlängerung der Gesamtbearbeitungszeit infolge unzureichender Parallelität der Programme



EPIP: externe und interne Parallelität: mögliche Aufteilung in Raum und Zeit

4.3 Einfluss von Kommunikationsaufwand

- Ein paralleles Programm bestehe aus m Prozessen, von denen jeder R Zeiteinheiten Rechenzeit benötigt.
- Die m Prozesse werden auf die p Prozessoren verteilt:

$$\sum_{i=1}^p m_i = m$$

wobei m_i die dem Prozessor i zugeteilte Zahl von Prozessen darstellt.

- Jeder Prozess kommuniziert mit jedem anderen, wofür jeweils C Zeiteinheiten zu veranschlagen sind.
- Sind die zwei kommunizierenden Prozesse demselben Prozessor zugeteilt, so fallen keine (bzw. vernachlässigbare) Kommunikationskosten an.
- Bei beliebiger Verteilung der m Prozesse erhalten wir eine Gesamtrechenzeit $T_x(p)$ von

$$T_x(p) = R \max_i(m_i)$$

und eine Gesamtkommunikationszeit von

$$T_c(p) = \frac{1}{2} \sum_{i=1}^n m_i(m - m_i)C = \frac{C}{2} \left(m^2 - \sum_{i=1}^n m_i^2 \right)$$

Kommunikationsaufwand

- Beide Teile zusammen ergeben eine Gesamtbearbeitungszeit von

$$T(p) = T_x(p) + T_c(p) = R \max_i(m_i) + \frac{C}{2} \left(m^2 - \sum_{i=1}^p m_i^2 \right)$$

- Bei gleichmäßiger Verteilung der Prozesse auf die Prozessoren ($m_i = m/p$) vereinfacht sich der Ausdruck für $T(p)$ zu

$$T(p) = R \frac{m}{p} + \frac{C}{2} m^2 \left(1 - \frac{1}{p} \right)$$

- Man kann nun fragen: Lohnt sich Parallelverarbeitung, d.h. gilt $T(p) < T(1)$?

$$T(p) < T(1)$$

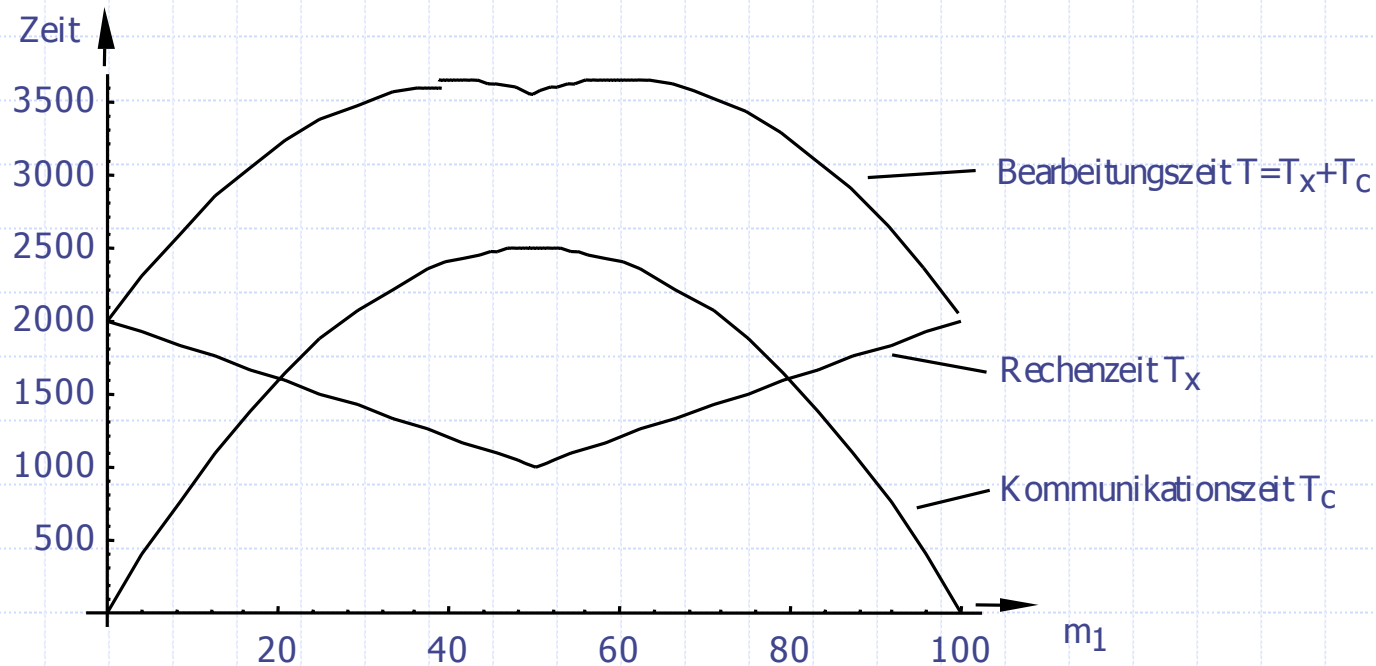
$$R \frac{m}{p} + \frac{C}{2} m^2 \left(1 - \frac{1}{p} \right) < R m$$

$$\frac{C}{2} m^2 \left(1 - \frac{1}{p} \right) < R m \left(1 - \frac{1}{p} \right)$$

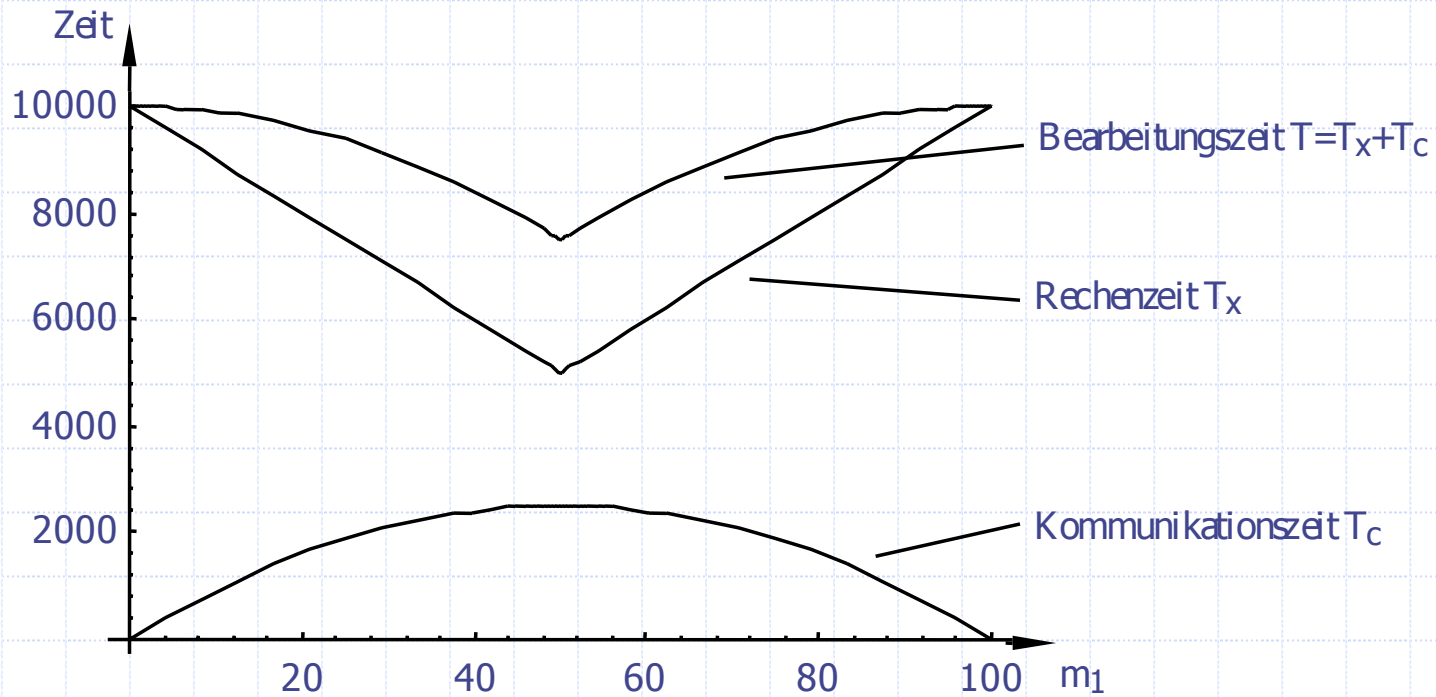
$$\frac{m}{2} < \frac{R}{C}$$

Kommunikationsaufwand

- Eine Verteilung und damit echte Parallelität lohnt sich in diesem Fall also nur, wenn das Verhältnis von Rechenzeit R zu Kommunikationszeit C größer ist als $m/2$.
- Das Verhältnis R/C stellt daher eine kritische Größe dar und entscheidet über die *Granularität* der Parallelverarbeitung
- Beispiel ($p=2$ Prozessoren, $m= 100$ Prozesse, $C =1$ (Kommunikationszeit))



Bearbeitungszeit als Funktion der Verteilung bei $R=20$ (d.h. $R/C = 20$)
 Verteilung lohnt sich nicht !



Bearbeitungszeit als Funktion der Verteilung bei $R=100$ (d.h. $R/C = 100$)
 Verteilung lohnt sich (beide Prozessoren je 50 Prozesse)

