Einführung in die Parallelität

Seminarvortrag

23.01.2007

Inhaltsübersicht:

1. Klassifikation von Parallelrechnern
2. Besonderheiten von parallelen Algorithmen
3. Grundlegende globale Operationen
4. Leistungsbewertung paralleler Algorithmen

1.1 Klassifikation nach Flynn

Unterscheidungskriterium: Organisation von Befehls- und Datenströme

	Instruction Stream		
	Single	Multiple	
Single	SISD	MISD	Data Stream
Multiple	SIMD	MIMD	

SISD :
ein Befehl verarbeitet einen Datensatz (herkömmliche PCs)

MISD :
mehrere Befehle verarbeiten den gleichen Datensatz (nicht realisiert)

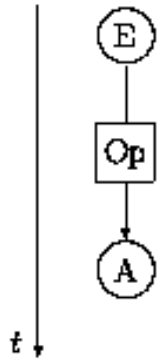
SIMD :
ein Befehl verarbeitet mehrere Datensätze

MIMD :
unterschiedliche Befehle verarbeiten unterschiedliche Datensätze (Konzept fast aller moderner Parallelrechner)

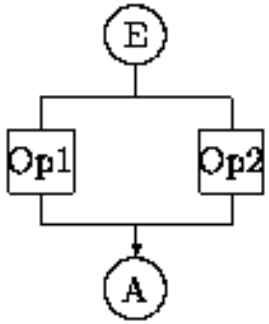
1.1 Klassifikation nach Flynn

MIMD-Codes laufen nicht völlig unabhängig von einander ab :

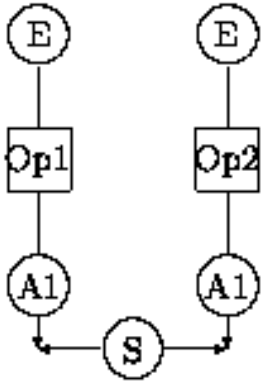
- Konkurrierende Prozesse : Nutzen einen gemeinsamen Speicher
- Kommunikative Prozesse : Jeder Prozess besitzt seinen eigenen Datenstrom, der Datenaustausch an einem bestimmten Punkt erforderlich macht



einzelne Operation



zwei parallele Operationen mit konkurrierendem Zugriff auf die Daten



zwei parallele Operationen auf verteilten Daten, die an einem bestimmten Punkt synchronisiert werden

1.2 Klassifikation durch Speicherzugriff

Shared Memory (gemeinsamer Speicher): Speicher, auf den mehrere konkurrierende Prozesse „gleichzeitig“ zugreifen

Vor-/ Nachteile:

+ Jeder Prozess hat auf sämtliche Daten Zugriff
  Serielles Programm kann ohne größere Schwierigkeiten zum Laufen gebracht werden.
  Führt in der Regel bei kleinen Prozessorzahlen schnell zu einer ersten Leistungssteigerung

- Anwachsen der Zugriffskonflikte bei größeren Prozesszahlen Skalierbarkeit (d.h. Leistung ~ Prozessanzahl) nicht mehr gewährleistet

- Sehr effiziente Verwaltungszugriffe und Bussysteme sind nötig, um die Zugriffsfehler zu verringern (teuer)

1.2 Klassifikation durch Speicherzugriff

Modelle, um Shared Memory-Systeme zu konstruieren:

- **UMA** (Uniform Memory Access)
  großer gemeinsamer Speicher auf den alle Prozessoren mit gleicher Geschwindigkeit zugreifen können

- **NUMA** (Nonuniform Memory Access)
  Zu jedem Prozessor gehört ein lokaler Speicher, auf den

  die anderen Prozessoren auch zugreifen können
  Zugriffszeit abhängig von Speicherort

  Variante : **COMA** (Cache Only Memory Access)
  Speicher besteht ausschließlich aus Cache

1.2 Klassifikation durch Speicherzugriff

Distributed Memory (verteilter Speicher):

Zusammenstellung von Speicherteilen, auf die jeweils nur durch einen Prozessor zugegriffen werden kann (Kommunikation nötig)

Vor-/ Nachteile:

+ keine Zugriffsfehler zwischen Prozessoren
+ Hardware ist relativ günstig
+ fast beliebig skalierbar
- kein direkter Zugriff auf Daten, die auf anderen Prozessoren gespeichert sind
  Kommunikation durch spezielle Kanäle (Links) nötig
  Serielles Programm nicht lauffähig,
  spezielle parallele Algorithmen benötigt

1.2 Klassifikation durch Speicherzugriff

DSM: (Distributed Shared Memory)

Kompromiss zwischen beiden Speichermodellen

Der Speicher ist auf alle Prozessoren verteilt, aber das Programm kann den Speicher als gemeinsamen Speicher behandeln (virtuelles geteiltes Speichermodell)

Vorteil:

- serieller Code läuft sofort auf diesem Speichermodell
- gute Skalierbarkeit erreichbar, bei Ausnutzen der lokalen Anordnung von Daten (d.h. viele Datenzugriffe eines Prozesses können von den eigenen lokalen Speichern geleistet werden)

1.3 Kommunikationstopologien

Frage: Wie lassen sich Prozessoren eines Parallelrechners verbinden? Insbesondere ist dies für die Klasse der Parallelrechner mit verteiltem Speicher von Interesse.

Link: Verbindung zwischen zwei Prozessoren

- ungerichteter Link: zu einer Zeit nur in eine Richtung nutzbar
- bigerichteter Link: zu jeder Zeit in beide Richtungen nutzbar

Topologie: Allgemein eine Vernetzung der Prozesse

Physische Topologie: Vernetzung von Prozessoren, die durch Hersteller in der Hardware vorgegeben ist

1.3 Kommunikationstopologien

Logische Topologie [virtuelle Topologie] :
vom Nutzer gewünschte oder vom Betriebssystem vorgegebene Vernetzung der Prozesse

Durchmesser eines Netzwerks:
Max. Anzahl von Linkverbindungen, die eine Nachricht benutzen muss, um von einem Knoten p zu einem Knoten q zu gelangen (p,q beliebige Knoten im Netzwerk).

2.1 Synchronisation

undefinierter Status: entsteht, wenn das Ergebnis einer Datenmanipulation nicht vorhergesagt werden kann

meist: wenn mehrere Prozesse auf beschränkte Ressourcen zugreifen (d.h. bei Shared Memory)

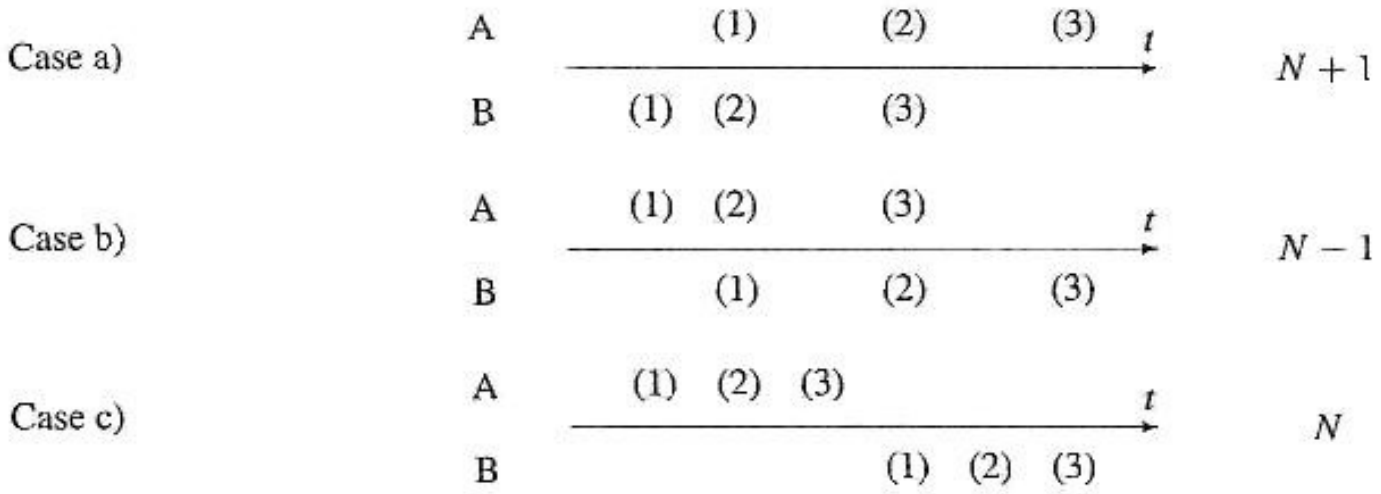
Beispiel:

process A: $N := N + 1$	clock	process B: $N := N - 1$
LOAD N	(1)	LOAD N
INC N	(2)	DEC N
STORE N	(3)	STORE N

Beide Prozesse laufen gleichzeitig auf unterschiedlichen Prozessoren ab und modifizieren dieselbe Variable

2.1 Synchronisation

Mögliche Ergebnisse:



Wert von N abhängig von Ausführungsgeschwindigkeit von Prozess A und B

Also keine Vorhersage möglich

2.1 Synchronisation

Synchronisation : Mechanismus, so dass die Prozesse auf wohldefinierte Weise auf die Variable zugreifen können

Shared Memory: Semaphorenkonzept

Distributed Memory: Nachrichtenübertragung

Schranke : spezielle Synchronisationsmechanismus, der aus einem bestimmten Punkt im Programm besteht, der von allen Prozessen (oder einer Gruppe von Prozessen) durchlaufen werden muss bevor die Ausführung fortgesetzt wird.

Jeder einzelne Prozess muss warten bis alle übrigen Prozesse diesen Programmpunkt erreicht haben

2.2 Message Passing

Message Passing (Nachrichtenübertragung): Mechanismus, um Daten direkt von einem Prozess zu einem anderen zu übertragen

Unterscheidung:

- **Blockkommunikation:**

  Blockkommunikation lässt alle (oft zwei) beteiligten Prozessoren warten bis alle Prozesse für einen Daten-/Nachrichtenaustausch bereit sind

- **Nichtblockkommunikation:**

  Nichtblockkommunikation erlaubt Prozessen ihre Daten/Nachrichten total unabhängig von den übrigen Prozessen zu versenden und zu erhalten

2.2 Message Passing

Vorteil (Nichtblockkommunikation):

Hohe Effizienz in der Kommunikation ohne Deadlock

Senderprozess erwartet keine Bestätigung, ob Operation erfolgreich ausgeführt wurde

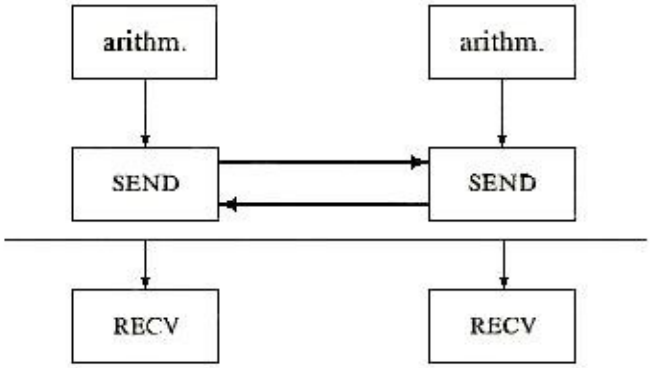
Andere Operationen können während des Aufrufs der Kommunikationsroutine ausgeführt werden

2.3 Deadlock

Deadlock (Verklemmung) **:** tritt auf, wenn mehrere Prozesser auf ein Ergebnis warten, das nur von einem dieser wartenden Prozesse kommen kann

Einfaches Beispiel:

Datenaustausch zwischen zwei Prozessoren bei Blockkommunikation



Problem: Kommunikationroutinen, die komplizierter sind als einfaches senden/empfangen führen zu Deadlocks, die nicht so offensichtlich sind

3.1 Send und Recv

Jedes Parallelprogramm braucht grundlegende Routinen zum Senden und Empfangen von Daten von einem Prozess zum anderen

Routinen:

- **SEND(nwords,data,ProNo)**
  Senden von „nwords“ von Daten, die in „data“ abgespeichert sind vom aufgerufenen Prozess zum Prozess „ProcNo“

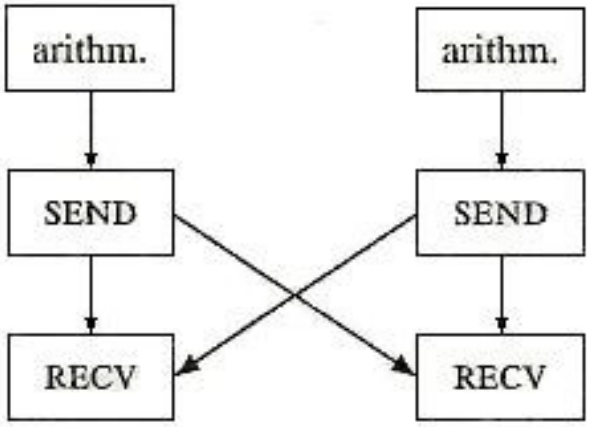
- Empfangen: **RECV(nwords,data,ProNo)**

3.2 Exchange

Zusammenfassen der Routinen SEND und RECV führt zur Routine **EXCHANGE (ProcNo,nWords, SendData,RecvData)**

Datenaustausch zwischen Prozessen

- Nichtblockkommunikation:



Die neben stehende Variante arbeitet nur im Fall von Nichtblockkommunikation

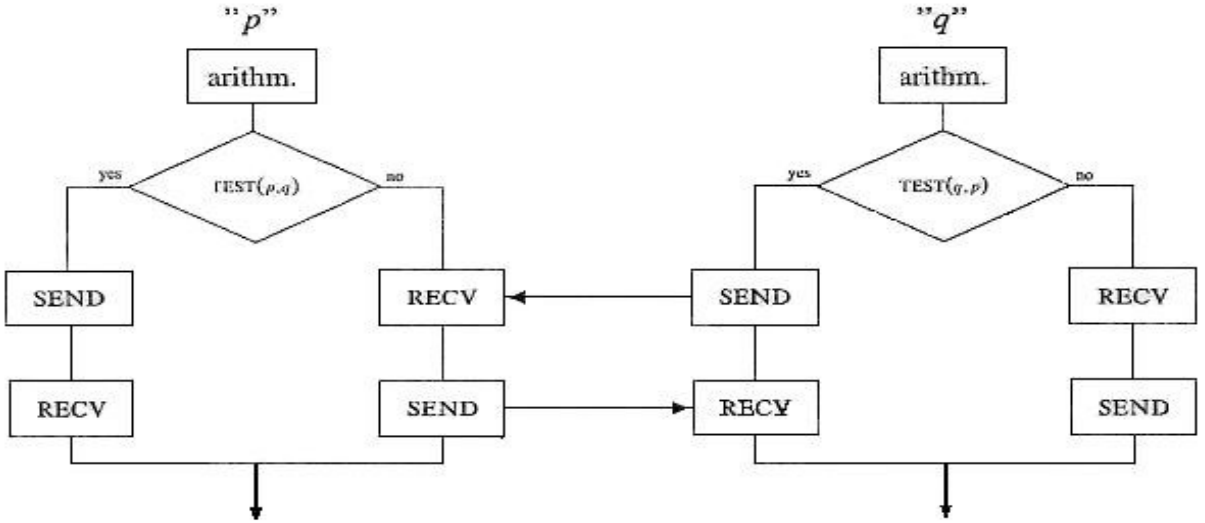
d.h. keine Bestätigung an den sendenden Prozess, dass die Nachricht empfangen wurde; Blockkommunikation führt zu Deadlock

3.2 Exchange

- Blockkommunikation:

  Für einen Deadlock-freien Austausch zwischen p und q braucht man eine eindeutige Funktion, die festlegt welche der beiden Prozesse zuerst sendet/empfängt.

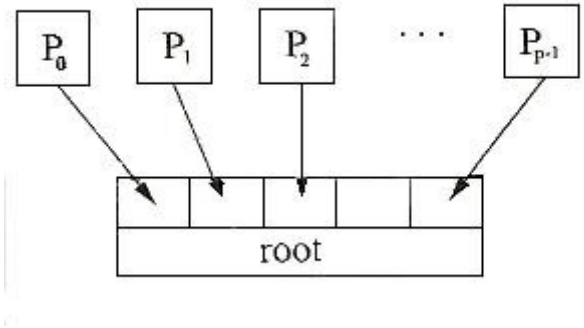
  Die Funktion TEST (p, q) = (p > q) sorgt für einen Ordnungsmechanismus, so dass beide Prozesse unabhängig dasselbe Ergebnis erhalten.



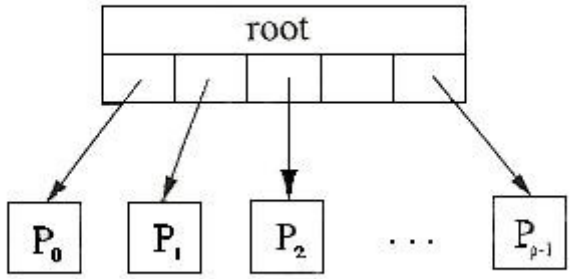
3.3 Gather und Scatter

Gewöhnlich besitzt ein Parallelprogramm einen Root-Prozess, der exklusiven Zugriff auf die Ressourcen hat. Dieser Prozess „sammelt“ und „verteilt“ spezifische Daten von und auf andere Prozesse.

- GATHER(root,RecvData, MyData)
- SCATTER(root, SendData, MyData)



gather



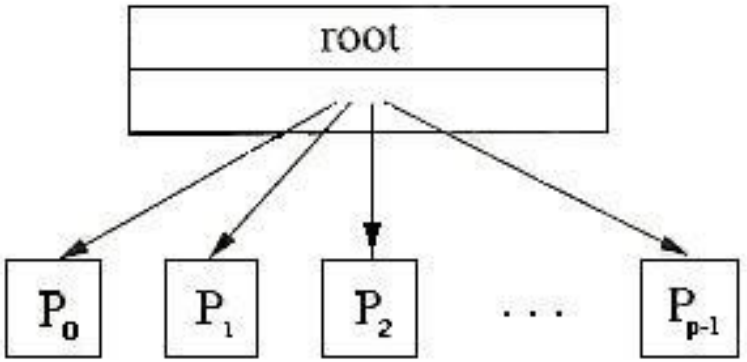
scatter

3.4 Broadcast

Oft erhalten alle Prozessoren dieselbe Information von einem Prozess Dann : Root-Prozess wie eine Sendestation

Routine: **BROADCAST(root,nWords,Data)**

Vereinfachung der Scatter-Routine, die individuelle Daten an die Prozesse verteilt



4.1 Speedup und Scaleup

Wie vergleichen wir die Eignung oder die Fehlerhaftigkeit von parallelen Algorithmen für eine große oder riesige Anzahl von Prozessoren?

Skalierbarkeit: Einfache Anpassung eines Programmes an eine real verfügbare Anzahl von Prozessoren

Körnigkeit: ist ein Maß für die Größe von Programmabschnitten, die ohne Kommunikation mit anderen Prozessen durchführbar sind

4.1 Speedup und Scaleup

Speedup (Beschleunigung): S_p Maß für den Gewinn bei der Ausführung eines parallelen Codes der auf P Prozessoren läuft im Vergleich zur seriellen Programmversion,

$$S_p = \frac{t_1}{t_p}$$

t_1:Ausführungszeit von Prozess A auf einem Prozessor

t_p: Ausführungszeit von Prozess A auf P Prozessoren

Ideal: Speedup von P (P Prozessoren sind P-mal schneller als einer)

N_{glob} globale Problemgröße bleibt unverändert, deshalb sinkt die lokale Problemgröße für jeden Prozessor

Beispiel: Arbeiter

4.1 Speedup und Scaleup

Amdahl´sches Gesetz :

Jeder Algorithmus enthält nicht parallelisierbare Teile

Sei s der serieller Anteil und p der parallele Anteil eines Algorithmus (beide normalisiert): $s + p = 1$

So lässt sich die Rechnerzeit auf einem Prozessor durch t_1 $s + p$ und die Rechnerzeit auf P Prozessoren durch t_p $s + p / P$ ausdrücken (optimale Parallelisierung)

obere Schranke für den maximal erreichbaren Speedup

$$S_{p,max} = \frac{s+p}{s+\frac{p}{P}} = \frac{1}{s+\frac{1-s}{P}} \leq \frac{1}{s}$$

4.1 Speedup und Scaleup

Annahme: 99% eines Codes lassen sich parallelisieren, dann beschränkt der übrige 1% sequentielle Anteil die Parallelausführung, so dass ein maximaler Speedup $S_{\infty,max} = 100$ erreicht werden kann

P	10	100	1000	10000
S_P	9	50	91	99

Nutzen von mehr als 100 Prozessoren scheint sinnlos :

Kosten von 1000 Prozessoren 10-mal höher als die Kosten von 100 Prozessoren,

aber die Leistung nicht mal verdoppelt so hoch

4.1 Speedup und Scaleup

Ist Amdahl`sche Gesetz für uns relevant ?

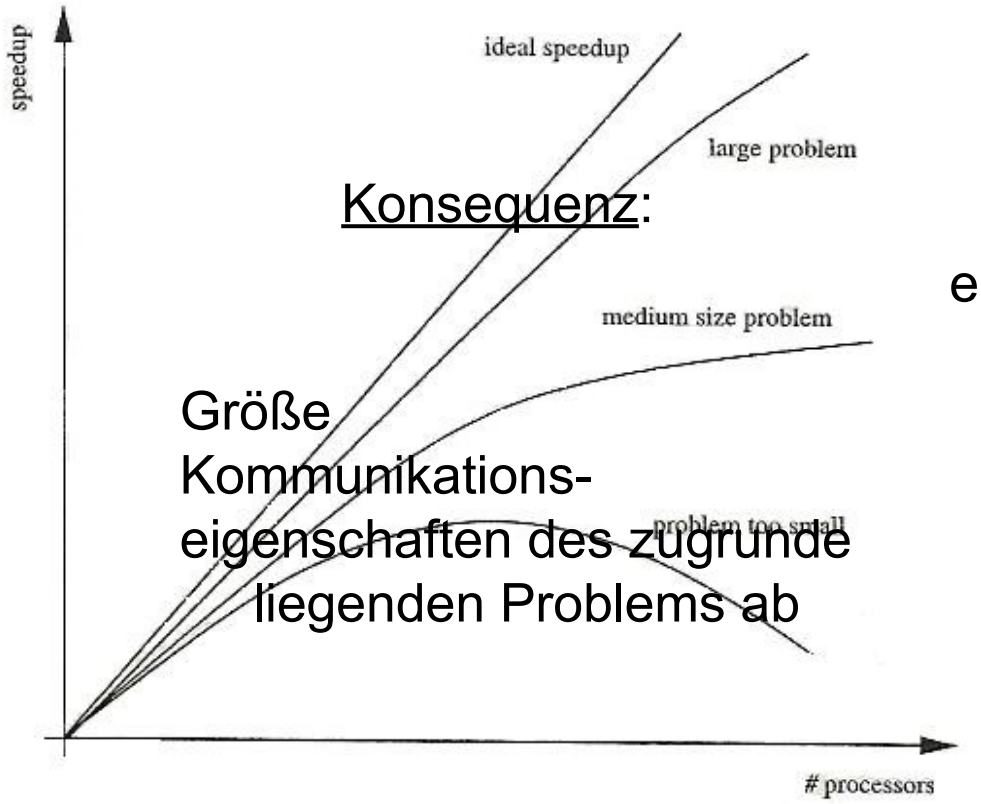
- Bei kleineren Prozessorzahlen (<20) ist bei bestimmten Algorithmen sogar ein höheren Speedup als P erreichbar.

- Keine Berücksichtigung von Kommunikationszeiten

  Realität: noch schlechterer Speedup

4.1 Speedup und Scaleup

Speedup bezüglich der Problemgröße



Konsequenz:

erreichbarer Speedup hängt von der Größe und den Kommunikations-eigenschaften des zugrunde liegenden Problems ab

4.1 Speedup und Scaleup

Scaled Speedup : S_c Abschätzung des Gewinn bei paralleler Ausführung, wenn die Größe des globalen Problems mit der Anzahl der genutzten Prozessoren sinkt

Quantitativ ausgedrückt:

$$S_C(P) = \frac{s_1 + P \cdot p_1}{s_1 + p_1} \overset{s_1+p_1=1}{=} s_1 + P(1 - s_1)$$

s_1+ p_1 : normalisierte Rechnerzeit auf einem Parallelrechner

s_1 + Pp_1 : Rechnerzeit auf einem sequentiellen Rechner

d.h. sequentieller Anteil von 1% führt zu einem Scaled Speedup von P

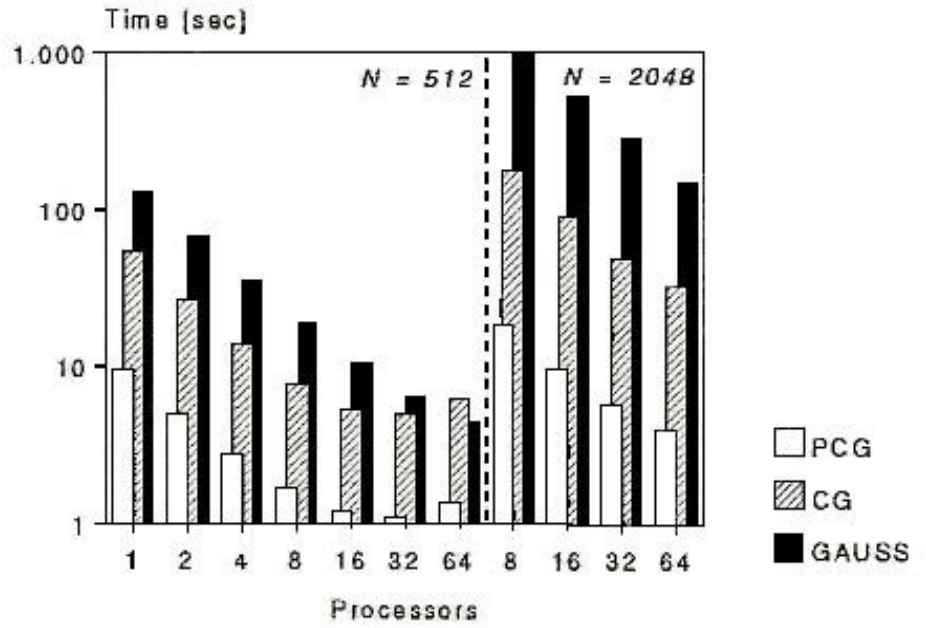
4.1 Speedup und Scaleup

(Scaled) Speedup sollte nicht das einzige Kriterium für den Nutzen eines Algorithmus sein:

z.B. gilt:

S_p (Gauß) > S_p (PCG)

ABER: t (Gauß) >> t (PCG)



gute Parallelisierung ist für numerisch ineffiziente Algorithmen wertlos

4.2 Effizienz

Parallele Effizienz: gibt Auskunft, wie nahe unsere parallele Implementierung an der optimalen Beschleunigung liegt

Formel für die skalierte Effizienz:

$$E_{C,par} = \frac{S_C(P)}{P}$$

Ideal: Effizienz von 100%

d.h. Nutzen von P Prozessoren beschleunigt den Prozess um den Faktor P

4.2 Effizienz

parallele Effizienz für die Beschleunigung lässt sich explizit angeben :

$$E_{C,par} = \frac{S_C(P)}{P} = \frac{s_1 + P(1 - s_1)}{P} = 1 - s_1 + \frac{s_1}{P} > 1 - s_1$$

Leichtes Abnehmen Effizienz bei wachsender Prozessorzahl

Effizienz jedoch mindestens so hoch wie paralleler Anteil im Programm

4.2 Effizienz

Numerische Effizienz: vergleicht den schnellsten seriellen Algorithmus mit dem schnellsten parallelen Algorithmus auf einem Prozessor

$$E_{num} = \frac{t_{parallel}}{t_{serial}}$$

4.2 Effizienz

Daraus folgt die skalierte Effizienz eines parallelen Algorithmus:

$$E = E_{C,par} \cdot E_{num}$$

Formel :

- Kein Verlust durch Kommunikation berücksichtigt
- Setzt gleichmäßige Verteilung des globalen Problems auf alle Prozessoren voraus, was in der Praxis a priori oder im Lauf der Rechnung nicht oft der Fall ist

geringere Effizienz in der Praxis als in der Theorie

(oftmals erheblich)

4.2 Effizienz

Loadbalancing-Strategien :

Versuch der gleichmäßigen Auslastung der Prozessoren

- Statisches Load-balancing (a priori)
  - Aufteilung der Netze, Daten nach Abzählen ist nicht effektiv und nutzt keine Nachbarschaftsbeziehungen aus
  - Aufteilung der Daten nach gewissen Bisektionstechniken

- Dynamische Load-balancing :

  Umverteilung der Daten während der Rechnung
  - großer Aufwand für die Leistungsverteilung und gute Heuristiken nötig, um Kommunikationsabstürze zu verhindern
  - Bei hoch adaptiven Verfahren notwendig

ENDE