Effiziente Nutzung von Hochleistungsrechnern in der numerischen Strömungsmechanik

Dr. Georg Hager
georg.hager@rrze.uni-erlangen.de

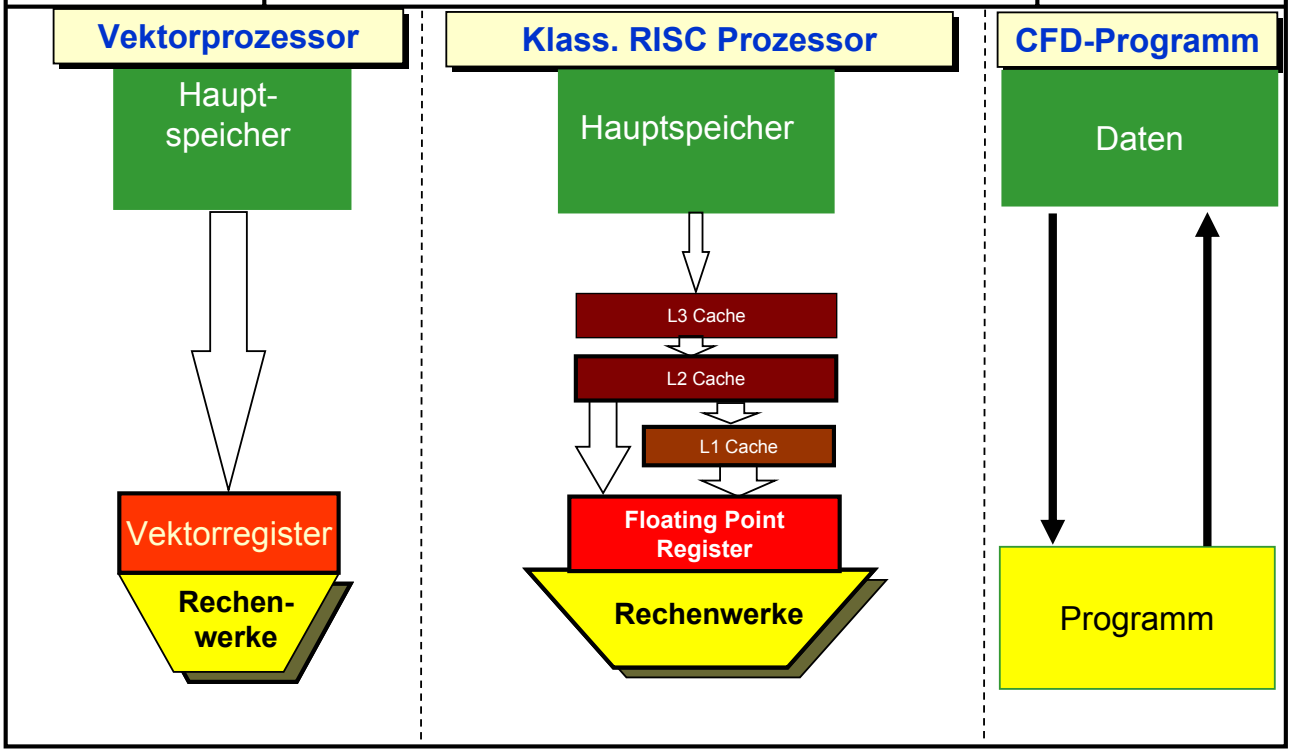
HPC Services
Regionales RechenZentrum Erlangen

Überblick

- Moderne Prozessoren für numerische Anwendungen
  - Allgemeine Optimierungsrichtlinien für CFD
- Parallelrechner
  - Möglichkeiten und Grenzen der Parallelität
  - Designprinzipien
  - effiziente Programmierung für CFD-Probleme
- Beispiel: SIPSolver nach Stone (Finite-Volumen)
- Zusammenfassung

Moderne Prozessoren für numerische Anwendungen

Speicherhierarchien moderner Prozessoren Schematischer Aufbau



- **Rechenleistung** in Fließkommaoperationen pro Sekunde: **Flop/s**
  - i.A. Multiplikationen und Additionen mit doppelter Genauigkeit
  - Divisionen, SQRT etc. sind sehr langsam
  - max. FLOP/s-Zahl ist üblicherweise das Doppelte oder Vierfache der Prozessor-Taktfrequenz
- **Bandbreite** (BW), d.h. Geschwindigkeit des Datentransfers: **GB/s**
  - in CFD-Anwendungen meist der limitierende Faktor: Bandbreite zwischen CPU und Hauptspeicher
- **Latenz**: Zeitdauer vom Anstoßen eines Datentransfers bis zum Eintreffen des ersten Bytes
  - CPU↔CPU, CPU↔Speicher, Rechenwerke↔Cache
  - dominant, wenn pro Transfer nur wenig Daten übertragen werden

		Intel Xeon	AMD Opteron	Intel Itanium2	NEC SX8
Spitzenleistung Taktfreq.		6.4 GFlop/s 3.2 GHz	4.8 GFlop/s 2.4 GHz	4.0 GFlop/s 1.0 GHz	16 GFlop/s 2.0 GHz
# FP Registers		16/32	16/32	128	8 x 256 (Vector)
L1	Size	16 kB	64 KB	16 KB (kein FP)	---
	BW	102 GB/s	51.2 GB/s	32 GB/s	---
	Latenz	12 Takte	3 Takte	1 Takt	---
L2	Size	1.0 MB	1.0 MB	256 KB	---
	BW	102 GB/s	51.2 GB/s	32 GB/s	---
	Latenz	18 Takte	13 Takte	5-6 Takte	---
L3	Size	---	---	3 MB	---
	BW	---	---	32 GB/s	---
	Latenz	---	---	12-13 Takte	---
Mem.	BW	6.4 GB/s r/w	6.4 GB/s r/w	6.4 GB/s r/w	64 GB/s r/w
	Latenz	~150 ns	~80 ns	~200 ns	Vektorisierung

- Caches bestehen aus **Cachelines**, die nur als Ganzes gelesen/geschrieben werden
  - typische Längen: 8/16/32 Worte (je 8 Byte)
- **Cache Miss**: Verwenden eines Datums, das noch nicht im Cache liegt
  - Folge: Cacheline wird aus Memory geladen (kostet Latenz + Zeit für Datentransfer)
  - alle weiteren Zugriffe auf Cacheline sind schnell
- Cachelines werden nach einem bestimmten Algorithmus wieder aus dem Cache entfernt (zurückgeschrieben und/oder invalidiert)
- **Worst Case**: Laden einer Cacheline, von der nur ein einziges Element benutzt wird
  - tritt auf bei irregulärem Speicherzugriff (z.B. indirekt)

- **Caches sind schnell und klein, Speicher ist groß und langsam!**
- Folge: Langsame Datenpfade sind zu vermeiden!

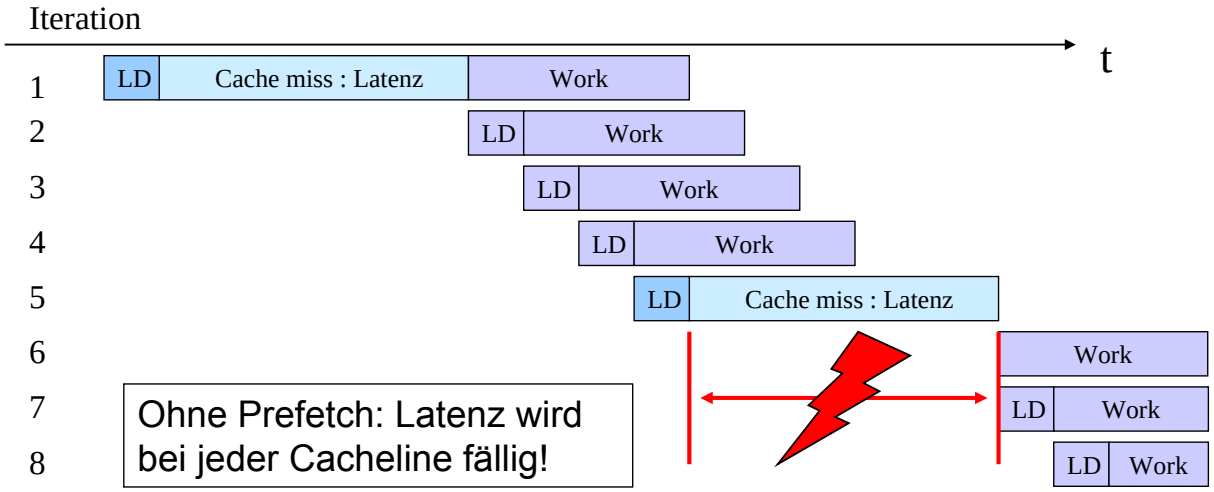
```
DO I=1,N
  A(I)=B(I)+C(I)
ENDDO
DO I=1,N
  A(I)=A(I)*D(I)
ENDDO
```

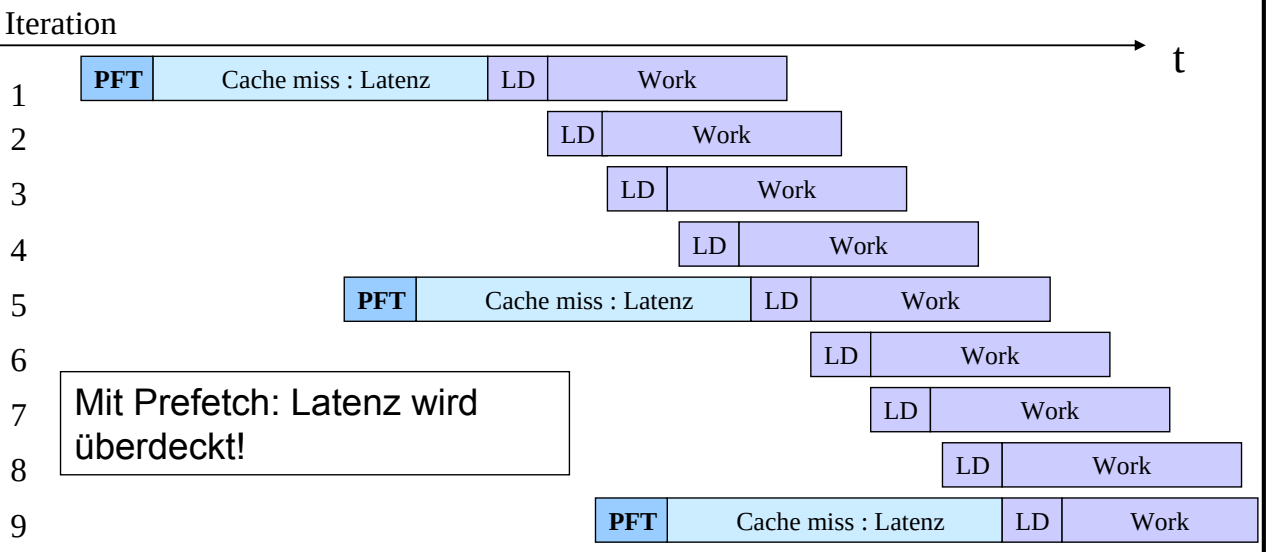
Optimierung

```
DO I=1,N
  A(I)=(B(I)+C(I))*D(I)
ENDDO
```

Effiziente Nutzung schneller Datenpfade ist absolut vorrangig!

- Wenn sich "Streaming" nicht vermeiden lässt (üblich in CFD), muss es wenigstens effizient geschehen: **Prefetching!**
  - **Prefetch** = Überdecken der Speicherlatenz durch rechtzeitiges Absetzen von Anfragen für Cachelines



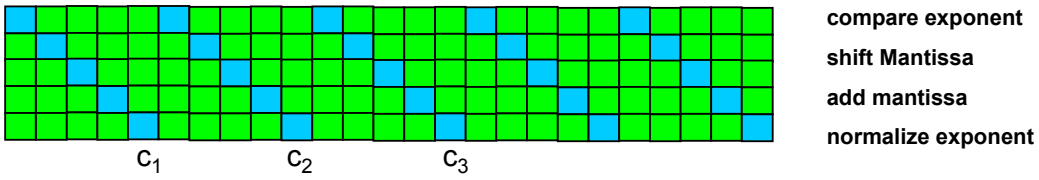


- Prefetching wird i.A. vom Compiler generiert (überprüfen!)
  - eingreifen u.U. im Quellcode möglich durch Compilerdirektiven
- Xeon, Opteron, Power4/5: zusätzlich **hardwarebasierter Prefetch**

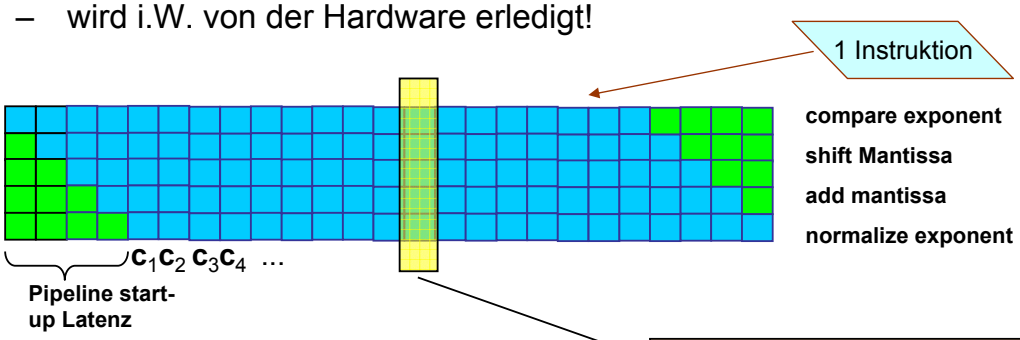
- Analog RISC: Pipelining ist das A und O
- Aufteilung einer komplexen Operation (hier Subtraktion) in:
  - compare exponent
  - shift mantissa
  - add mantissa
  - normalize exponent

```
1.77e4 - 9.3e3
1.77e4 - 0.93e4
0.84e4
8.40e3
```

- Ablaufschema für Array-Operation (C = A + B)
  - hier: ohne Pipelining

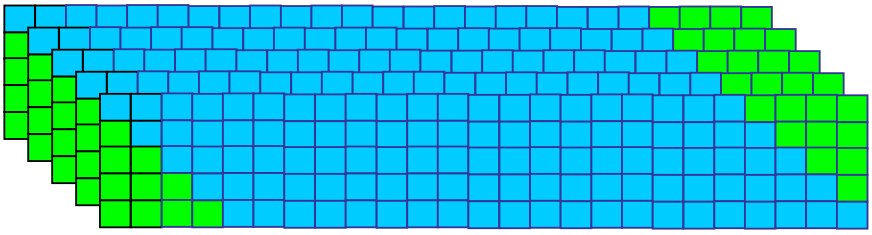


- Wie kann man das verbessern?
  - Pipelining: von 1 Ergebnis / 5 Takte zu einem Ergebnis / 1 Takt
  - wird i.W. von der Hardware erledigt!



- Vektor-CPUs haben die notwendige Speicherbandbreite, um Pipelines ohne Verzögerung zu "füttern"
- Prefetch erfolgt automatisch

- Noch besser:
  - mehrere parallele Pipelines
  - $\mathbf{N_p}$ "Tracks" → $\mathbf{N_p}$ Ergebnisse pro Takt



- Voraussetzungen für gute Performance:
  - Pipes beschäftigt halten!
  - Compiler muss Vektorstrukturen im Code erkennen können
    - Unabhängigkeit aufeinander folgender Operationen

- Woher weiß man, ob ein Code die Ressourcen des Systems effizient nutzt?
- In vielen Fällen ist eine Abschätzung der zu erwartenden Performance aus den Eckdaten der Architektur und der Codestruktur möglich

- Randbedingungen der Architektur (maschinenabhängig):
  Speicherbandbreite GWorte/s
  Fließkommaleistung GFlop/s
  daraus abgeleitet:

  Maschinenbalance $B_m = \frac{\text{Speicherbandbreite}\,[Worte/s]}{\text{Fließkommaleistung}\,[Flop/s]}$

- Typ. Werte (Hauptspeicher): 0.2 W/Flop (Itanium2 1.0 GHz)
  0.125 W/Flop (Xeon 3.2 GHz),
  0.5 W/Flop (NEC SX8)

- Zu erwartende Performance auf Schleifenebene?
- **Codebalance:**

$$B_c = \frac{\text{Datentransfer (LD/ST)}\,[Worte]}{\text{arithmetische Operationen}\,[Flops]}$$

- erwarteter Bruchteil der Peak Performance ("Lightspeed"):

$$l = \frac{B_m}{B_c}$$

- Beispiel?

- Kernel benchmarks:
  - Charakterisierung des Prozessors
  - Resultate können i.A. leicht verstanden und verglichen werden
  - Erlauben oft eine obere Schranke für die Leistungsfähigkeit eines Systems für eine spezifische Anwendergruppe abzuschätzen
  - Beispiele: *streams* , *cachebench*,…
- **Vektortriade**: Charakterisierung der erzielbaren Speicherbandbreiten

```
REAL*8 (SIZE): A,B,C,D
DO ITER=1,NITER
  DO i=1,N
    A(i) = B(i) + C(i) * D(i)
  ENDDO
  <OBSCURE>
ENDDO
```

Codebalance:
- Recheneinheiten:
  2 Flops / i-Iteration
- Bandbreite:
  (3 LD & 1 ST) / i-Iteration
- B_c = 4/2 = 2 W/Flop

- **Funktionseinheiten: Performance allein durch die arithm. Einheiten beschränkt:** **2 MULTIPLY-ADD (2 MADD) Pipelines**

  Vektortriade:

  2 Iterationen ⟺ 1 Takt

  4 Flops / Takt ⟺ **Peak Performance**

  ➡ **Die Daten liegen jedoch i.A. nicht in den Funktionseinheiten/Registern, sondern im**

- **L2 Cache:** Bandbreite von

  2 LD und (2 LD oder 2 ST) pro Takt (=32 GB/s bei 1 GHz)

  Maschinenbalance: **4/4 = 1 W/Flop**

  Vektortriade:

  B_m/B_c = 0.5 ➡ **½ Peak Performance (2 GFlop/s)**

- **L3 Cache:** komplexe Situation, weil immer aus L2 geladen wird
  Abschätzung: halbe Performance wie aus L2

- **Speicher**: Bandbreite von
  - 2 LD oder 2 ST mit 400 MHz (Frequenz des Speicherbuses!)
    Maschinenbalance: 0.8/4 = 0.2 W/Flop
  - Achtung: "Read for Ownership" benötigt einen zusätzlichen LD vor jedem ST, d.h. **effektive Codebalance: B_c = 2.5 W/Flop**

  Vektortriade:

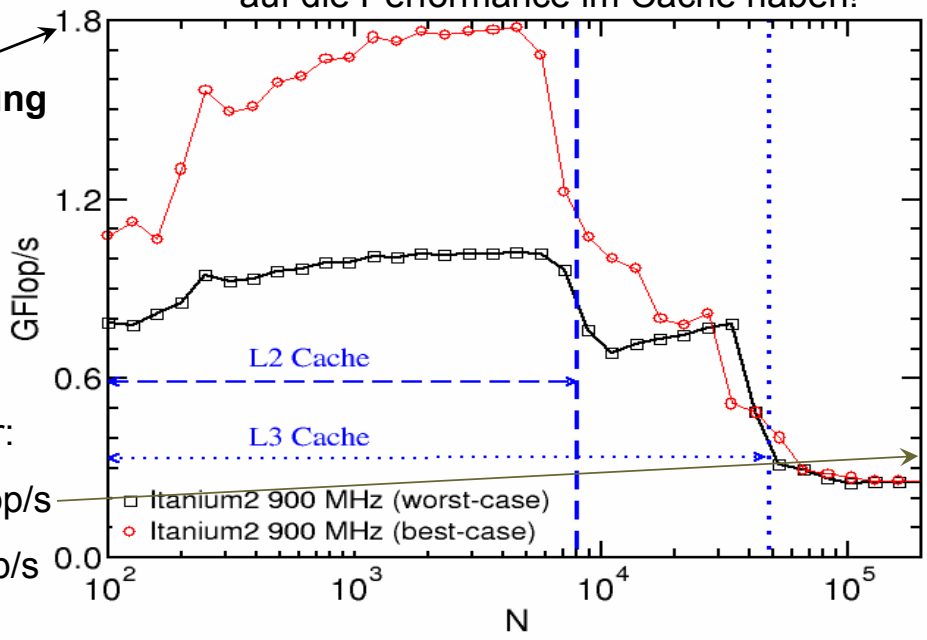
  B_m/B_c = 0.08 ➡ **8% der Peak Performance (320 MFlop/s)**

Kurzlehrgang NUMET	Lightspeed-Berechnung : Vektortriade auf Itanium2 – Messung	19 / 11

HP zx6000 – efc Compiler:

Memory-Layout kann starken Einfluss auf die Performance im Cache haben!

½ Spitzenleistung (L2 Cache)

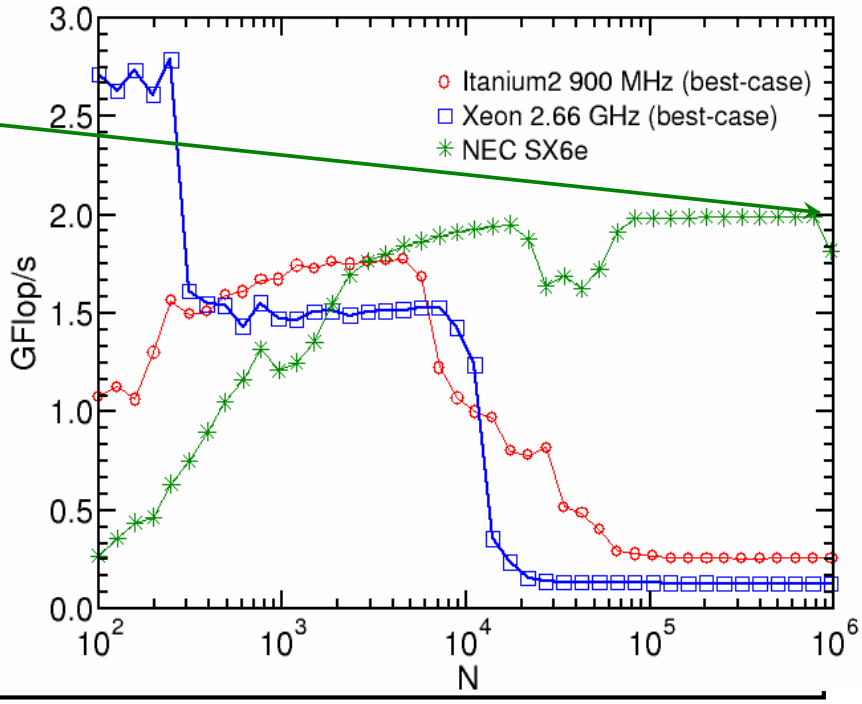


Hauptspeicher:

- Theorie:320 MFlop/s
- Praxis: 260 MFlop/s (81 %)

Kurzlehrgang NUMET	Kernel benchmarks: Vektor-Triade – Cache- vs. Vektorsysteme	20 / 11

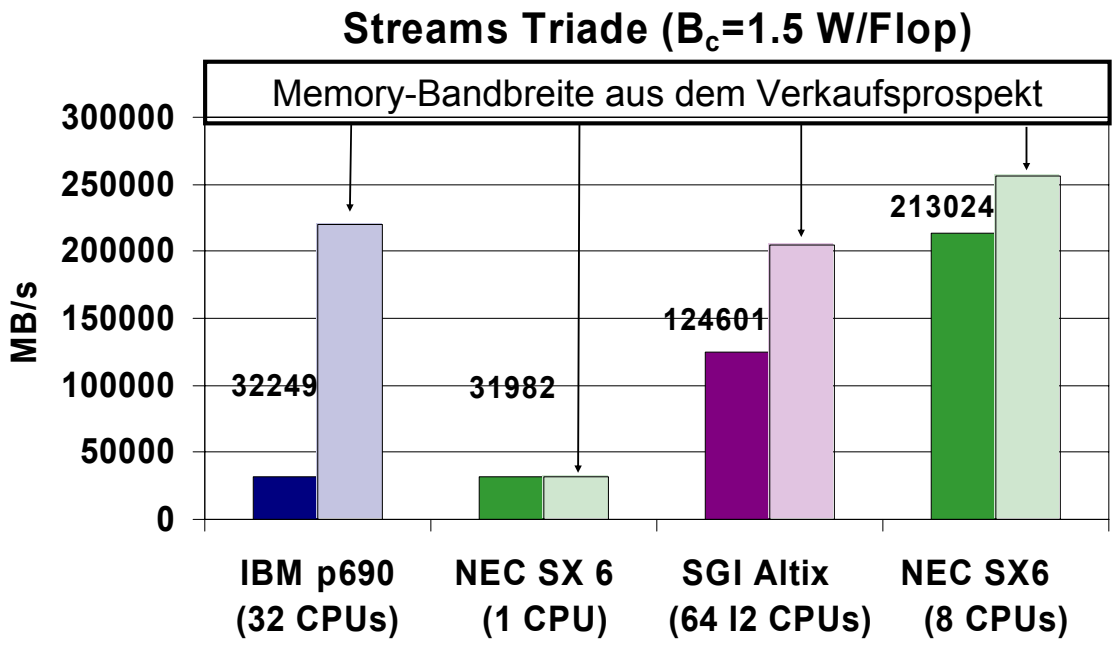
NEC SX6e
Max. Performance
2.0 GFlop/s



Welcher Rechner für welches Problem?

Cache basierte Systeme:
Häufige Nutzung kleiner Datensätze

Vektorrechner:
Seltener/unregelmäßiger Zugriff auf (sehr) große Datensätze

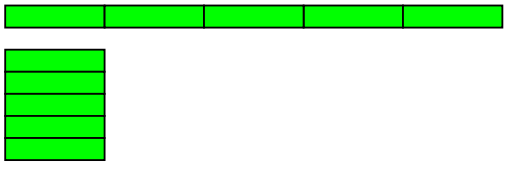


http://www.cs.virginia.edu/stream/standard/Bandwidth.html

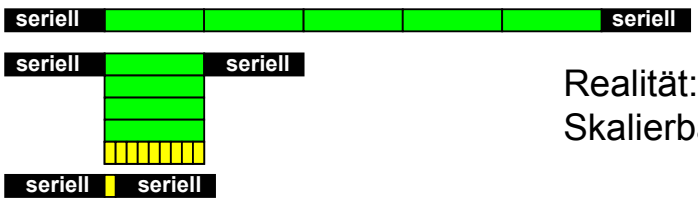
Möglichkeiten und Grenzen der Parallelität

"Parallelrechnen" = mehr als eine CPU zur Lösung eines numerischen Problems einsetzen

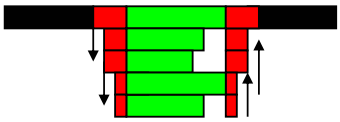
- Begriffe
  - **Skalierbarkeit** S(N): Wieviel mal schneller wird mein Problem gelöst, wenn es auf N CPUs läuft statt auf einer?
  - **Performance** P(N): Wieviele Rechenoperationen pro Sekunde macht das Programm mit N CPUs im Vergleich mit einer CPU?
  - **parallele Effizienz** ε(N)=S(N)/N: Welchen Bruchteil jeder CPU nutzt das parallele Programm?
- Viele numerische Aufgaben sind im Prinzip parallelisierbar
  - Lösen linearer Gleichungssysteme: **Ax = b**
  - Eigenwertberechnungen: **Ax = λx**
  - Zeitpropagation: **p(x,t+1)=$\mathcal{F}$(p(x,t))**



Idealfall: Arbeit ist komplett parallelisierbar

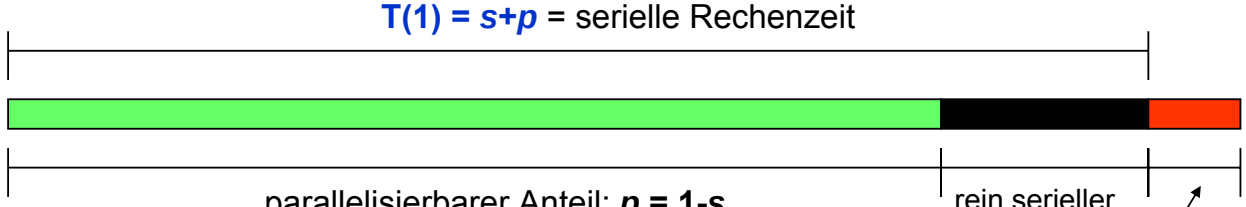


Realität: rein serielle Anteile begrenzen Skalierbarkeit



Kommunikation veschlimmert die Situation noch weiter

T(1) = *s*+*p* = serielle Rechenzeit

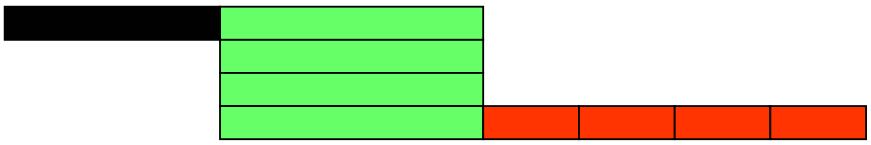


parallelisierbarer Anteil: ***p* = 1-*s***

rein serieller Anteil ***s***

parallel: T(*N*) = *s*+*p*/*N*+*Nk*

Bruchteil ***k*** für Kommunikation zwischen je 2 Prozessoren



Allgemeine Formel für Skalierbarkeit (worst case):
Fall *k*=0: "Amdahl's Law" (strong scaling)

$$S_p^k(N) = \frac{T(1)}{T(N)} = \frac{1}{s + \frac{1-s}{N} + Nk}$$

- Limes großer Prozessorzahlen:
  - bei k=0: Amdahl's Law!

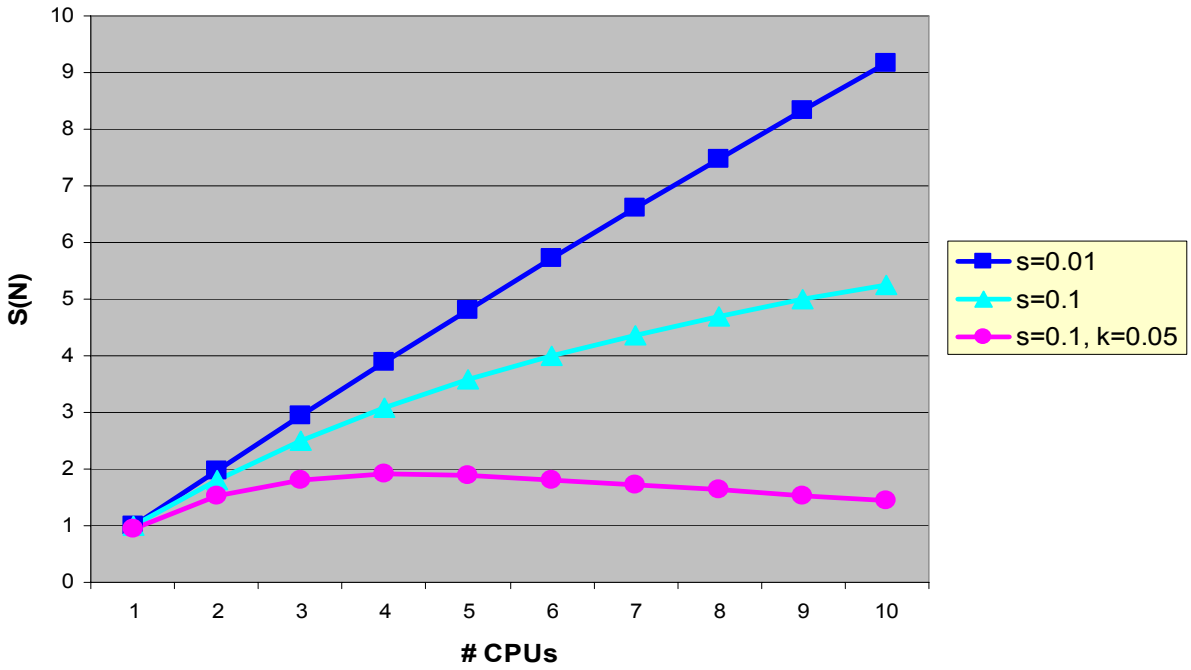
$$\lim_{N\to\infty} S_p^0(N) = \frac{1}{s}$$

unabhängig von *N*

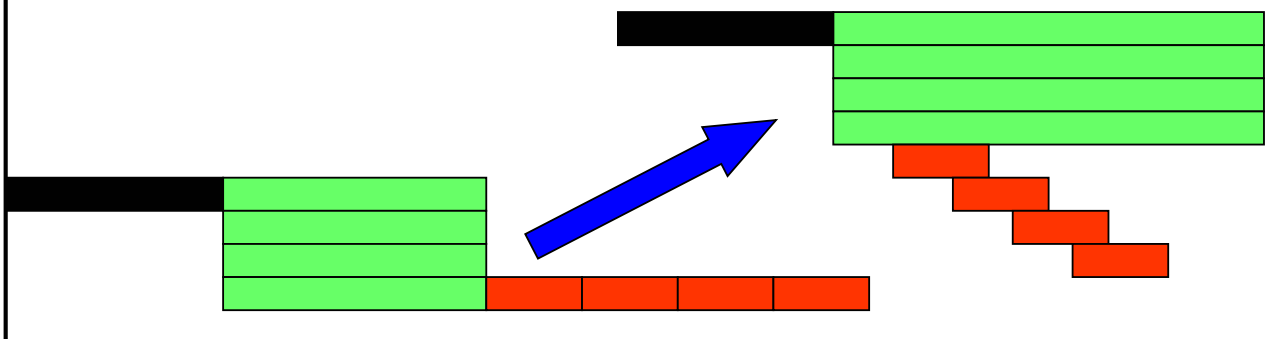
  - bei k≠0: einfaches Kommunikationsmodell liefert:

$$S_p^k(N) \xrightarrow{N>>1} \frac{1}{Nk}$$

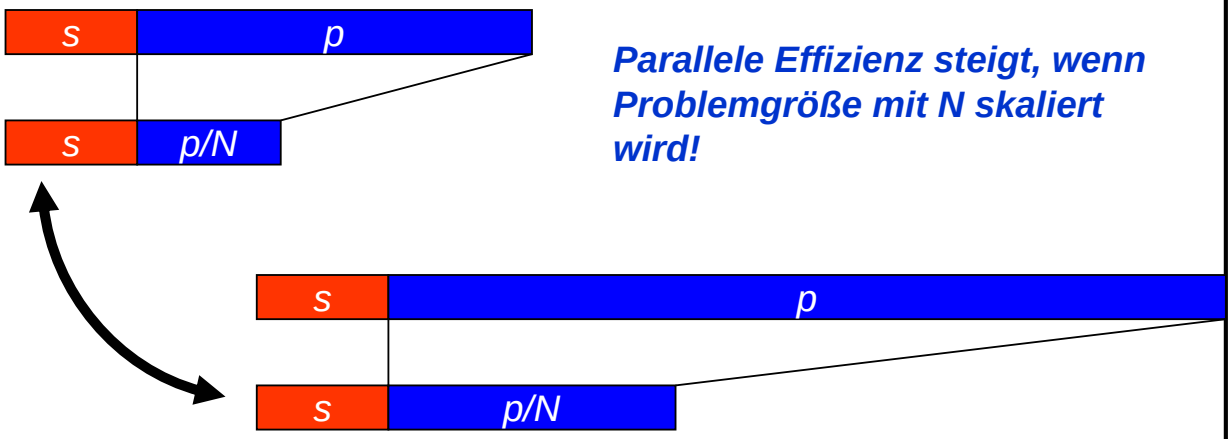
- Die Realität ist noch viel schimmer:
  - Load Imbalance
  - Overhead beim Starten paralleler Programmteile



- Kommunikation ist nicht immer rein seriell
  - nichtblockierende Netzwerke können gleichzeitig verschiedene Kommunikationsverbindungen schalten – Faktor *Nk* im Nenner wird zu log *k* oder *k* (technische Maßnahme)
  - u.U. kann Kommunikation mit nützlicher Arbeit überlappt werden (Implementierung, Algorithmus):



- Vergrößern des Problems hat oft eine Vergrößerung von *p* zur Folge
  - *p* skaliert, während *s* konstant bleibt
  - Anteil von *s* an der Gesamtzeit sinkt
  - Nebeneffekt: Kommunikationsaufwand sinkt eventuell



Parallele Effizienz steigt, wenn Problemgröße mit N skaliert wird!

- Quantifizierung? Betrachte zunächst *k=0*!
  - Sei wie vorher ***s+p=1***
  - Perfekte Skalierung: Laufzeit bleibt konstant bei Vergrößerung von *N*
  - **Skalierbarkeit ist keine relevante Größe mehr!**
    „Parallele Performance“ =

$$\frac{\text{Arbeit/Zeit für Problemgröße } N \text{ mit } N \text{ CPUs}}{\text{Arbeit/Zeit für Problemgröße 1 mit 1 CPU}}$$

$$P_s(N) = \frac{s + pN}{s + p} = s + pN = s + (1 - s)N$$

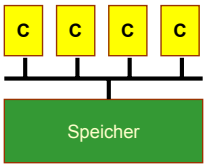
Gustafsson's Law ("weak scaling")

- Linear in *N* – aber die Defnition der "Arbeit" spielt eine große Rolle!

Designprinzipien moderner Parallelrechner

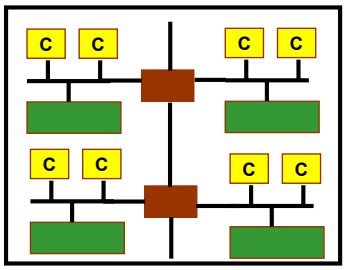
Moderne Parallelrechner Schematischer Aufbau

(1) Bus-basierte SMP-Systeme



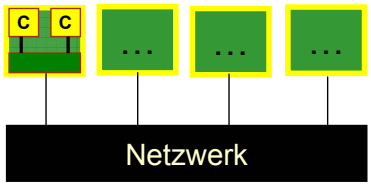
- 2-4 Prozessoren mit **einem** Speicherkanal
- Itanium / Xeon / Power4 / NEC SX

(2) Skalierbare „**Shared-memory**“ Systeme (**ccNUMA**)



- **Speicherbandbreite skaliert**
- Sehr schnelle interne Verbindungen (0.8 GB/s – 6.4 GB/s)
- SGI Origin / Altix, IBM p690 (ähnlich), Opteron-Knoten
- 4-256 Prozessoren

(3) Cluster von Rechenknoten (Architektur (1) oder (2))

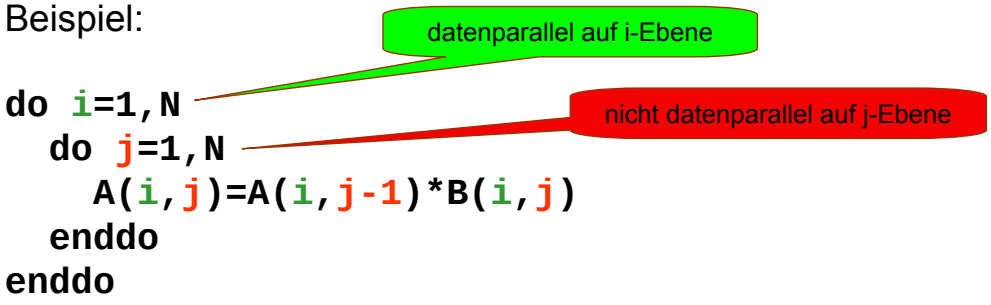


- **Speicherbandbreite skaliert, aber verteilter Speicher**
- Netzwerk: 50 – 1000 MByte/s
- 4 – 100000 Prozessoren

- Bei allen 3 Varianten (heute): **Dual-Core** möglich
  - d.h.: in einem "Sockel" steckt ein Chip mit 2 oder mehr (etwas langsameren) CPUs auf einem Stück Silizium
  - **Vorteil**: potenziell höhere Rechenleistung pro Sockel bei nahezu gleicher Leistungsaufnahme ("Macho-FLOPs")
  - **Nachteil**: Speicheranbindung der einzelnen CPU wird schlechter
- Dominant bei Parallelrechnern: **Cluster mit SMP-Knoten**
  - Kompromiss bzgl. Speicherbandbreite pro CPU
  - skalierend bis mehrere 10000 CPUs (akt. Rekord: 131072)
- Spezialfall: "**Constellation**"-Systeme
  - CPUs/Knoten > # Knoten
  - **SGI Altix (HLRB II): ccNUMA-Architektur**

Parallele Programmierung in der Strömungsmechanik

- Parallele Programmierung beginnt immer mit der Identifikation von Parallelität im Code
  - **Datenparallelität**: Welche Teile eines großen Datensatzes (Feld, Matrix, …) können im Prinzip gleichzeitig bearbeitet werden? Beispiel:




```
do i=1,N
  do j=1,N
    A(i,j)=A(i,j-1)*B(i,j)
  enddo
enddo
```

  - **funktionelle Parallelität**: Welche funktionalen Einheiten im Programm können im Prinzip gleichzeitig ablaufen?
    - in der Praxis selten benutzt

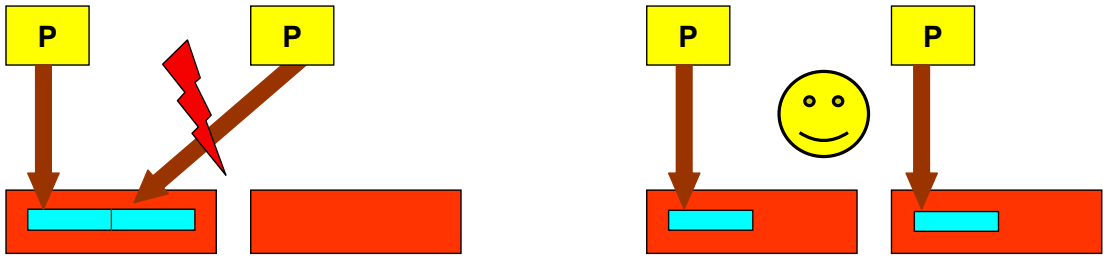
- Abhängig von der Rechnerarchitektur bieten sich 2 Zugänge an:
  - Distributed-Memory-Systeme

    Ein paralleles Programm besteht aus mehreren **Prozessen**. Jeder Prozess hat seinen eigenen Adressbereich und keine Zugriff auf die Daten anderer Prozesse. Notwendige Kommunikation findet durch Übermittlung von Messages statt. Standard: **MPI**

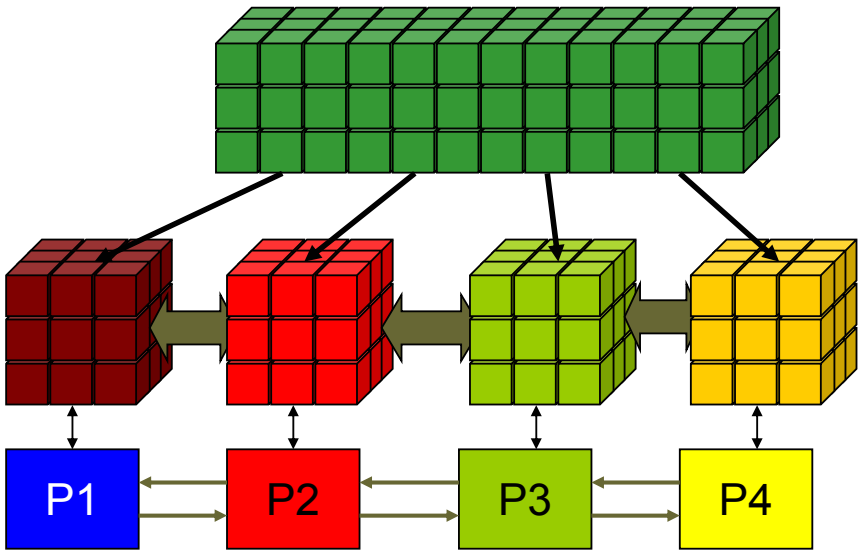
  - Shared-Memory-Systeme (auch ccNUMA)

    Ein paralleles Programm besteht aus mehreren **Threads**. Alle Threads haben Zugriff auf den kompletten Datenbereich. Standard: **OpenMP** (Compiler-Direktiven)

- Bei **ccNUMA-Systemen** (SGI Altix, Origin) sind Verbindungen zwischen Knoten i.A. langsamer als die lokale Speicherbandbreite
- Folge: Es muss darauf geachtet werden, dass jede CPU auf den benötigten Speicher **möglichst lokal** zugreifen kann
  - Problem dabei: "**First Touch**" Policy mappt Speicherseiten dort, wo sie das erste Mal angefasst werden (i.A. bei der Initialisierung)
  - Wenn die Initialisierung nicht in gleicher Weise parallelisiert ist wie die Rechenschleifen, folgt eine sehr ungünstige Zugriffsstruktur:



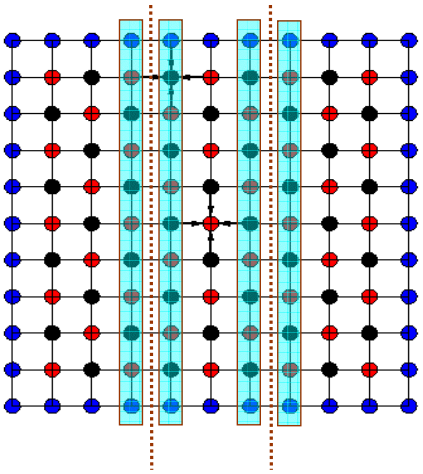
- Unabhängig von Programmiermodell: Parallelisierung von CFD-Codes läuft oft über "**Domain Decomposition**"
  - Zerlegung des Rechengebietes in N Teile – eines für jede CPU



- Problem: Temperaturverteilung auf quadratischer Platte mit Randbedingungen
  - Lösung der Laplace-Gleichung:

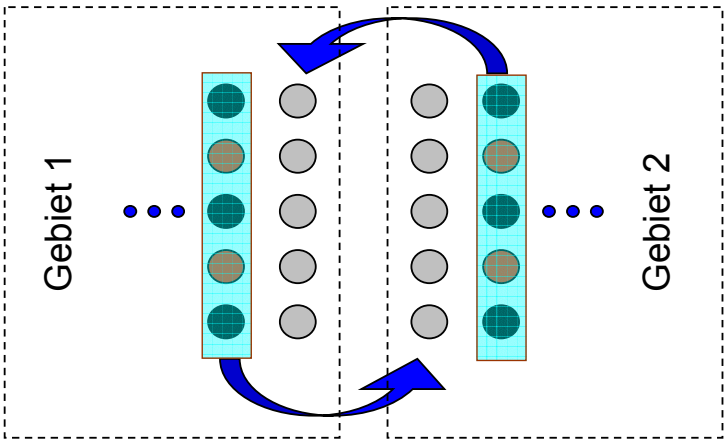
$$\Delta T = 0$$

  - Numerik: Abb. auf NxN-Gitter
  - Algorithmus: "Relaxation"
    T an einem Punkt = Mittelwert der 4 Nachbarn
  - Iteration: Update erst der roten, dann der schwarzen Punkte



- Parallelisierung mittels Gebietszerlegung
  - Was macht man mit den Randzellen eines Gebietes?

- Lösung des Randzellenproblems durch "Geisterzellen" (**ghost cells**)
  - jedes Gebiet hat eine zusätzliche Schicht Randzellen, wo es an ein anderes Gebiet grenzt
  - Nach jeder Iteration (red/black Sweep) werden die Geisterzellen mit dem Inhalt der Randzellen des Nachbarn **beschrieben**



- Wirkt sich die Kommunikation auf die Skalierbarkeit aus?
  - **Annahmen**: keine Überlappung zwischen Kommunikation und Rechnung, nichtblockierendes Netzwerk, 3D-Problem
- 2 Fälle:

(1) **strong scaling**: Problemgröße konstant, N steigt → Latenz *L* wird wichtig

Kommunikationszeit = $k/N^{2/3}+L$
Skalierung sättigt langsamer und auf niedrigerem Niveau!

$$S_p(N) = \frac{1}{s + \frac{1-s}{N} + kN^{-2/3} + L} \xrightarrow{N>>1} \frac{1}{s+L}$$

(2) **weak scaling**: Problemgröße steigt proportional zu N

Arbeit pro CPU bleibt konstant, ebenso die Kommunikation *k*
Performance skaliert langsamer, aber noch linear

$$P(N) = \frac{s + pN}{s + p + k} = \frac{s + (1-s)N}{1 + k}$$

CFD Applikationsperformance

SIP Solver nach Stone

- *Strongly Implicit Procedure* (SIP) nach Stone wird in Finite-Volumen Paketen oftmals zur Lösung des linearen Gleichungssystems

$$A\,x = b$$

verwendet

- Beispiele:
  - **LESOCC, FASTEST, FLOWSI** (LSTM, Erlangen)
  - **STHAMAS3D** (Kristalllabor, Erlangen)
  - **CADiP** (Theoret. Thermodynamik und Transportprozesse, Bayreuth)
  - **…**
- SIPSolver: **1) Unvollständige LU-Zerlegung von A**
  2) Serie von Vorwärts-/Rückwärts-Substitutionen und Residuenberechnungen
- Ursprüngliches Testprogramm (M. Peric): `ftp.springer.de:/pub/technik/peric`
- Optimization: **HPC-Gruppe RRZE**

Vorwärts-Substitution (naive 3D version)

Datenabhängigkeit: `(i,j,k)` ⟵ `{(i-1,j,k);(i,j-1,k);(i,j,k-1)}`

```
do k = 2 , kMax
   do j = 2 , jMax
      do i = 2 , iMax
               RES(i,j,k) ={RES(i,j,k)
$              - LB(i,j,k)*RES(i,j,k-1)
$              - LW(i,j,k)*RES(i-1,j,k)
$              - LS(i,j,k)*RES(i,j-1,k)
$                          }*LP(i,j,k)
      enddo
   enddo
enddo
```

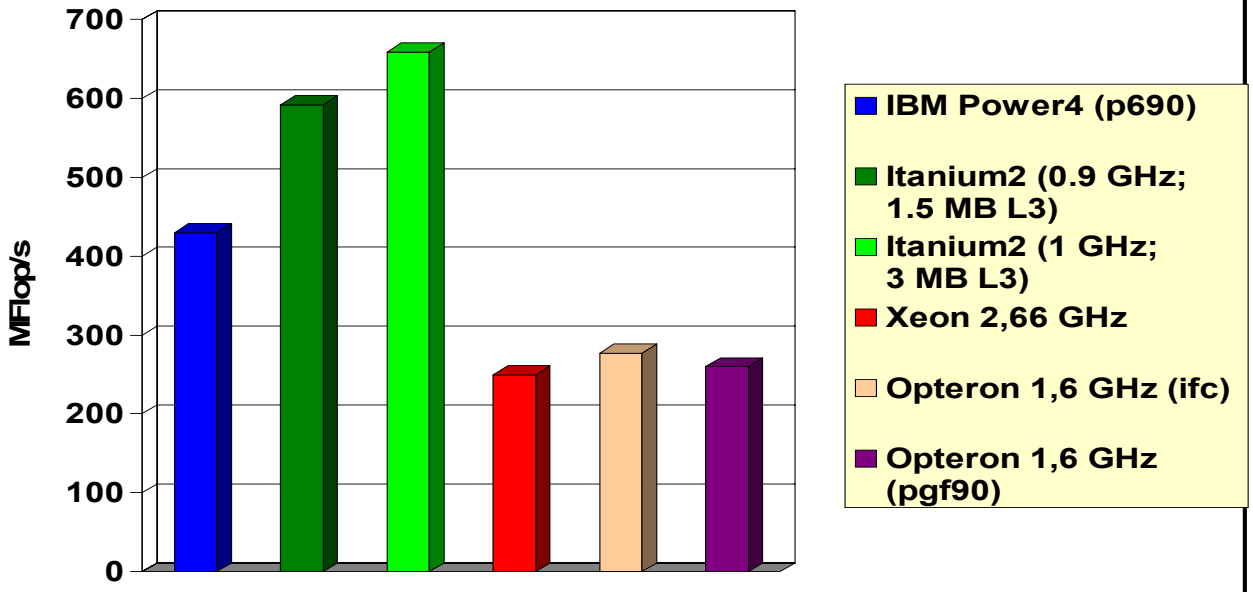
Datenabhängigkeit

- verhindert Vektorisierung oder einfache Parallelisierung
- erlaubt hohe örtliche Datenlokalität: Alle Elemente der Cachelines werden genutzt.
- erlaubt reuse von Cachelines: `RES(i,j,k),RES(i-1,j,k)`

→ Geeignete Implementierung für **Cache-basierte Prozessoren!**

Kurzlehrgang NUMET	SIPSolver: Naive 3D Implementierung: Einzelprozessorperfromance	45 / 11

- Gitter=91^3 (Problemgröße: 100 MB); 3D Version & naive Compiler-switches
- Performance: 1 Flop/Wort ➡ Obere Grenze für Itanium2: ~800 MFlop/s



Kurzlehrgang NUMET	SIPsolver: *hyperplane* Implementierung	46 / 11

Elimination der Datenabhängigkeit `(i,j,k)<-{(i-1,j,k);..}` durch Umgruppierung:

Innerste Schleife über eine *hyperplane*:

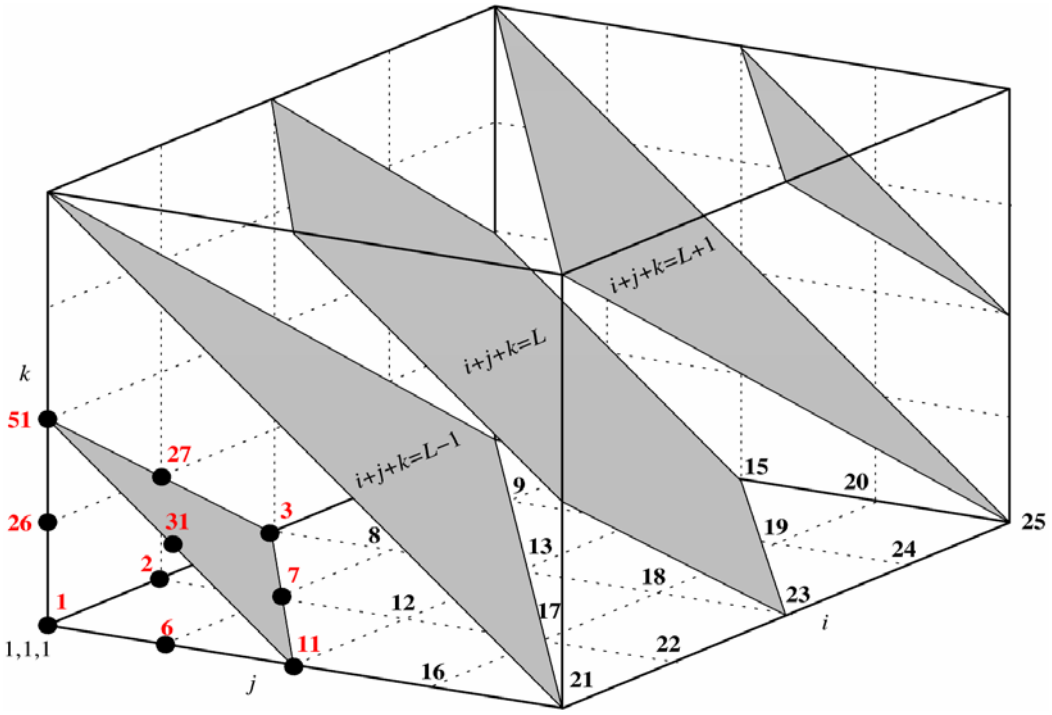
Alle Punkte **`i+j+k=l`**(mit festem **`l`**)

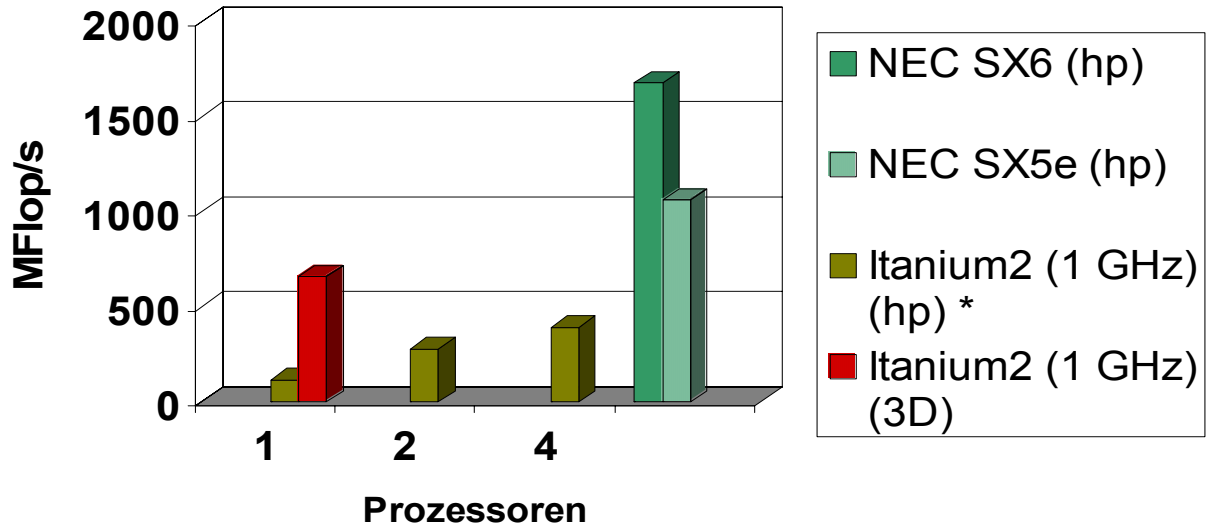
- liegen in der *hyperplane* **`l`**,
- sind voneinander unabhängig
- greifen nur auf Datenpunkte in der *hyperplane* **`l-1`** (Vorwärts-substitution) zu.

```
do l=1,hyperplanes
      n=ICL(l)
      do m=n+1,n+LM(l)
       ijk=IJKV(m)
       RES(ijk) =(RES(ijk) -
$      LB(ijk)*RES(ijk-ijMax)-
$      LW(ijk)*RES(ijk-1) -
$      LS(ijk)*RES(ijk-iMax) )
$      *LP(ijk)
      enddo
enddo
```

- Einfache Vektorisierung/ Parallelis. der inneren Schleife
- Verlust der Datenlokalität

➡ Geeignete Implementierung für **Vektorprozessoren**





- Gute Skalierung für Itanium2 SMP System, aber schlechte absolute Performance
- Gute absolute Einzelprozessorperformance für NEC SX6 Vektorprozessor

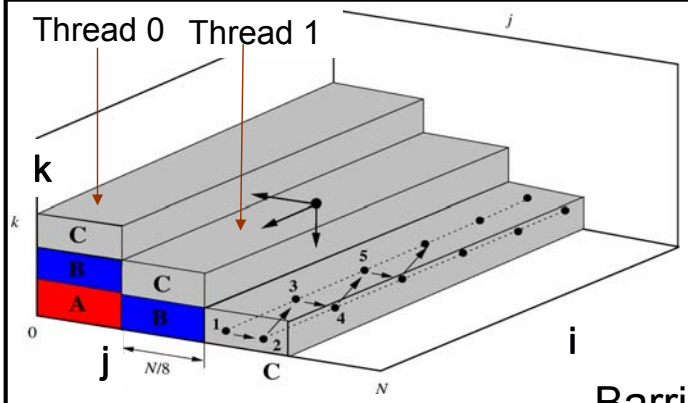
Ziel:

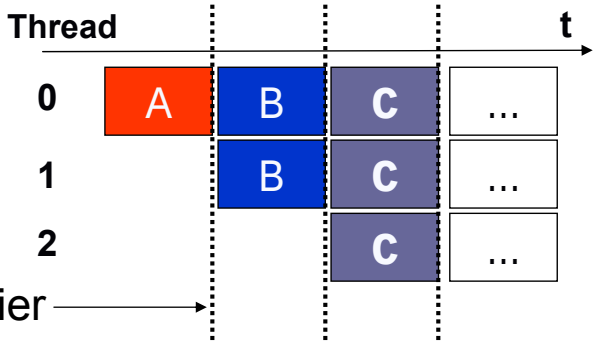
Parallelisierung (Shared-Memory) und Erhalt der Datenlokalität der 3D Implementierung

Vorgehensweise:

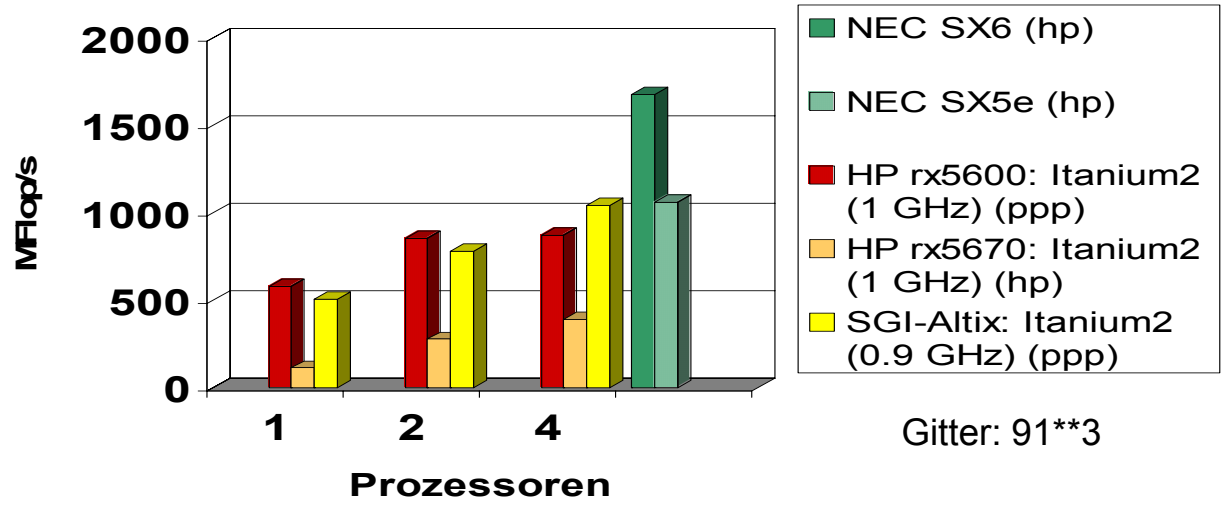
- 3D Implementierung **(i,j,k)<-{(i-1,j,k);..}**
- Parallelisierung in **j** Richtung
- Aufheben der Datenabhängigkeiten durch Synchronisieren nach jedem Schritt in **k** - Richtung

```
$omp parallel private(….)
do l =2, kMax+numThreads-2
   threadID=OMP_GET_THREAD_NUM()
   k = l – threadID
   if((k.ge.2).and.(k.le.kMaxM)) then
   do j = jS(threadID),jE(threadID)
      do i = 2 , iMax
        RES(i,j,k)={RES(i,j,k-1)-
$       LB(i,j,k)*RES(i,j,k-1)
$       …
      enddo
   enddo
   endif
$omp barrier
enddo
$omp end parallel
```

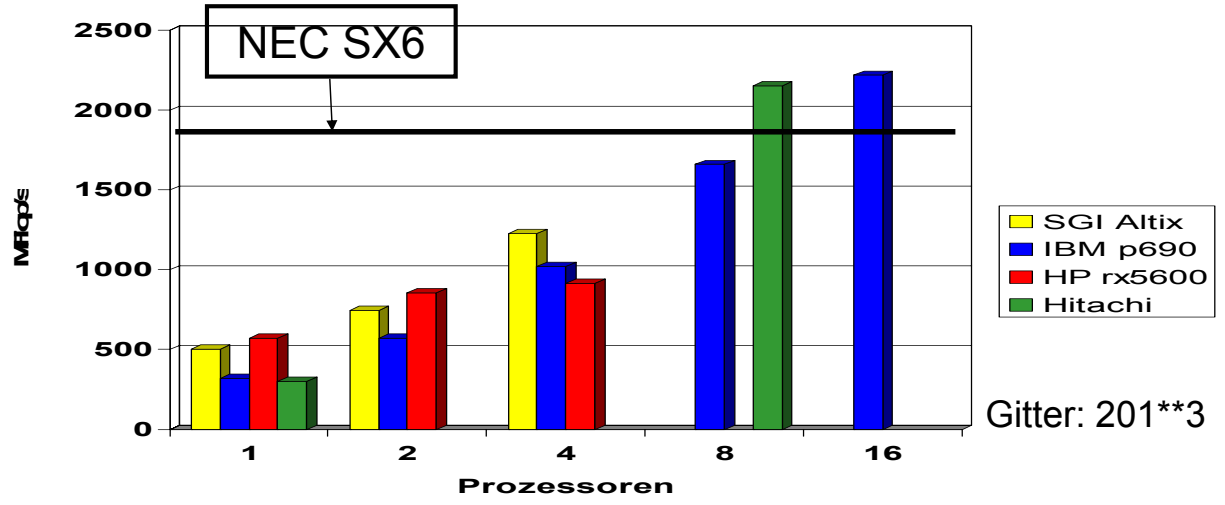


- Parallelisierung sinnvoll, falls **kMax, jMax >> #Threads**
- Automatische Parallelisierung der 3D Version durch Hitachi-Compiler!
- Beste Version für große SMP Knoten mit Cache-basierten Prozessoren



Gitter: 91**3

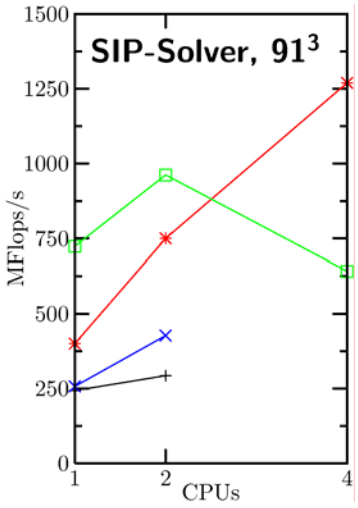
- HP rx5670: 4-fach Itanium2-System mit nur einem Speicherkanal für 4 Prozessoren (Speed-Up(4)=1,5)
- SGI Altix: 32-fach Itanium2-System mit einem Speicherkanal für **jeweils** 2 Prozessoren (Speed-Up(4)=2,0)



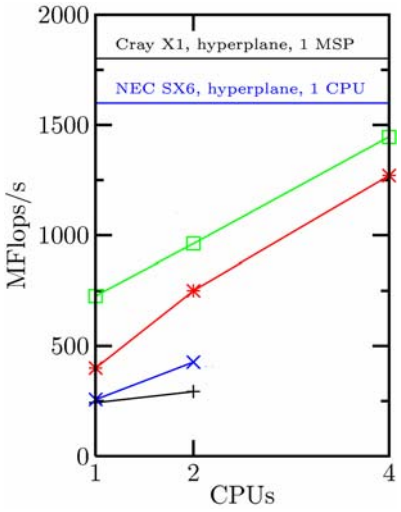
Gitter: 201**3

- IBM p690: 32-fach Power4-System mit einem Speicherkanal für **jeweils** 2 Prozessoren (Speed-Up(16)=7)
- Hitachi SR8000: 8-fach „Power3“-System mit einem Speicherkanal pro Prozessor (Speed-Up(8)=7.2)

- Performancevergleich



Falsches Placement



Korrektes Placement

Altix

- Konkrete Maßnahme: korrekte Parallelisierung der Initialisierungsschleife

```
!$omp parallel do private(i,j)
      do k=1,kMax
         do j=1,jMax
            do i=1,iMax
               T(i,j,k)=0.
            enddo
         enddo
      enddo
```

```
      do k=1,kMax
!$omp parallel do private(i)
         do j=1,jMax
            do i=1,iMax
               T(i,j,k)=0.
            enddo
         enddo
      enddo
```

Ergebnisse für CFD-Applikationen:

- Speicherbandbreite ist die entscheidende Einflussgröße
- Effiziente Nutzung schneller, Vermeidung langsamer Datenpfade ist Pflicht!
- Parallelisierung muss immer auch parallele Effizienz berücksichtigen
  - Kommunikationsaufwand beachten
  - strong/weak scaling
- Parallele Skalierbarkeit der Problemstellung ist Voraussetzung, um SMP Systeme/Cluster in Konkurrenz zu Vektorprozessoren zu setzen
- Bei ccNUMA-Systemen ist auf korrektes Placement zu achten!

Vielen Dank!

HPC-Dienste und Rechner:

- RRZE: **`http://www.rrze.uni-erlangen.de/hpc/`**
- LRZ: **`http://www.lrz-muenchen.de/services/compute/hlr/`**
- HLR Stuttgart: **`http://www.hlrs.de/`**
- NIC Jülich: **`http://www.fz-juelich.de/nic/`**

HPC allgemein:

- KONWIHR: **`http://konwihr.in.tum.de/`**
- TOP500: **`http://www.top500.org/`**
- Benchmarks: **`http://www.spec.org/`**
- LSTM: **`http://www.lstm.uni-erlangen.de/`**