Fachhochschule
Münster University of
Applied Sciences

Parallel Computing

Einsatzmöglichkeiten und Grenzen

Prof. Dr. Nikolaus Wulff

Parallel Architekturen

Flynn'sche Klassifizierung:

- SISD: s*ingle Instruction, single Data*
  - Klassisches von-Neumann sequentielles Rechenmodell.
- MISD: m*ultiple Instruction, single data*
  - Parallele Bearbeitung eines Datums, kaum realisiert außer für Hardware Beschleunigung bei Pipelineverarbeitung.
- SIMD: s*ingle instruction, multiple data*
  - Verschiedene Daten in parallel gleichartig zu verarbeiten ist in Vektor-Rechnern realisiert.
- MIMD: *multiple instruction, multiple data*
  - Universeller Parallelrechner

MIMD-Rechner

Die weitere Einteilung eines MIMD-Rechners erfolgt nach der physischen Organisation des Speichers:

- DMM: *distributed memory machines*
  - Rechnen mit physikalisch verteiltem Speicher auf unterschiedlichen Knoten => Verteiltes Rechnen
- SMM: *shared memory machines*
  - Rechnen mit physikalisch gemeinsamen Speicher in einem Multi-Prozessorsystem.
- SMP: *symmetric multiprocessors*
  - Eine Spezialform des SMM, eine kleine Anzahl von Prozessoren über eine Bus verbunden teilt sich Speicher und I/O, wie in heutigen Multi-Core CPUs realisiert.

Von Sequentiell zu Parallel

Herausforderungen an heutige Software Architekturen:

- Paralleles Rechnen mit shared memory
  - Synchronisation von Daten, Prozessen und Threads
- Verteiltes Rechnen auf verschiedenen Maschinen
  - Verteilung und Synchronisation von Daten

- Hybrid Systeme
  - Ein Cluster von parallelen arbeitenden Knoten, meist in speziellen SMP Super-Computern realisiert
- Grid- und Cloud-Computing
  - Moderne Form des Rechnen im Internet: CERN, Google...

Vorüberlegungen

Wann ist paralleles Rechnen sinnvoll?

- Wenn die Performance/Geschwindigkeit steigt.
- Wenn sich größere Probleme lösen lassen.
- Sich das Ganze finanziell lohnt!

Zwei wesentlichen Methoden dies zu erreichen sind:

- Mehrere (Multi-Core) CPUs mit shared Memory.
- Verteiltes Rechnen auf mehreren Maschinen.

Theoretische Grenzen

- Um wieviel lässt sich ein Programm durch paralleles Rechnen beschleunigen?
- Die Programmausführung zerfällt in drei Phasen

$$T_{total} = T_{septup} + T_{compute} + T_{finalize}$$

- Durch Parallelisierung wird die reine Rechenzeit proportional zur Anzahl *n* an CPUs verringert:

$$T_{total}(n) = T_{septup} + \frac{T_{compute}}{n} + T_{finalize}$$

- Es gibt offensichtlich einen seriellen Anteil σ, der sich nicht beschleunigen lässt.

Amdahl's Gesetz

Umformen der parallelen und seriellen Anteile liefert:

$$\sigma = \frac{T_{setup} + T_{finalize}}{T_{total}} \qquad T_{compute} = (1-\sigma) T_{total}$$

$$T_{total}(n) = T_{total}(1)\left[\sigma + \frac{1-\sigma}{n}\right]$$

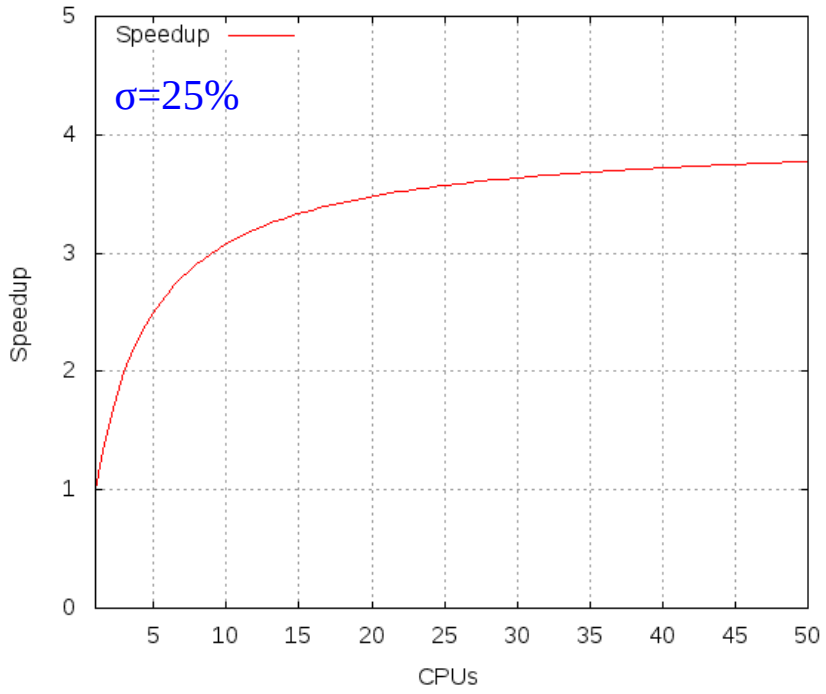
Der theoretische Geschwindigkeitszuwachs ist:

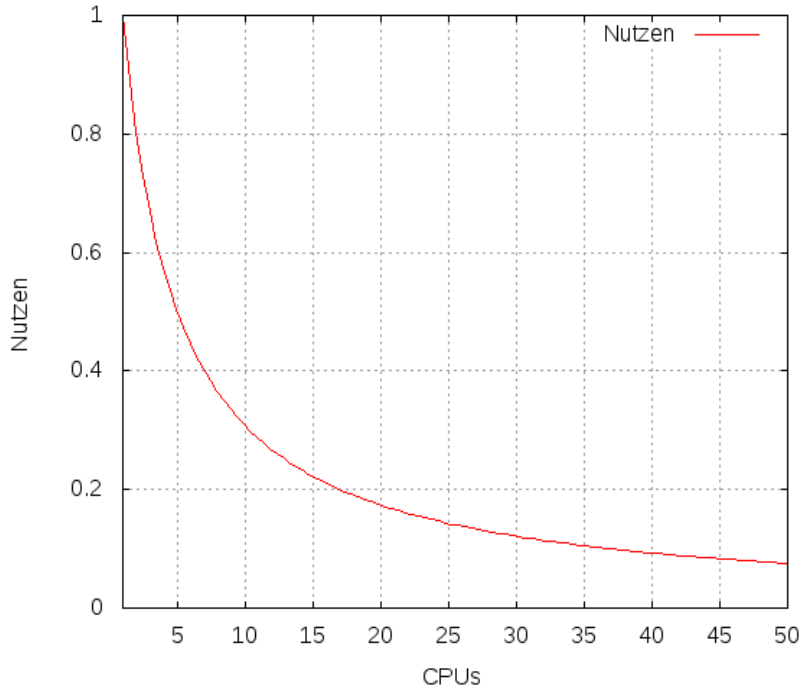
$$S(n) = \frac{T_{total}(1)}{T_{total}(n)} = \frac{1}{\sigma + \frac{1-\sigma}{n}}$$

Effizienz

- Interessant ist die Frage nach der Effizienz jeder weiteren CPU. Wie viel trägt sie zum Speedup bei?

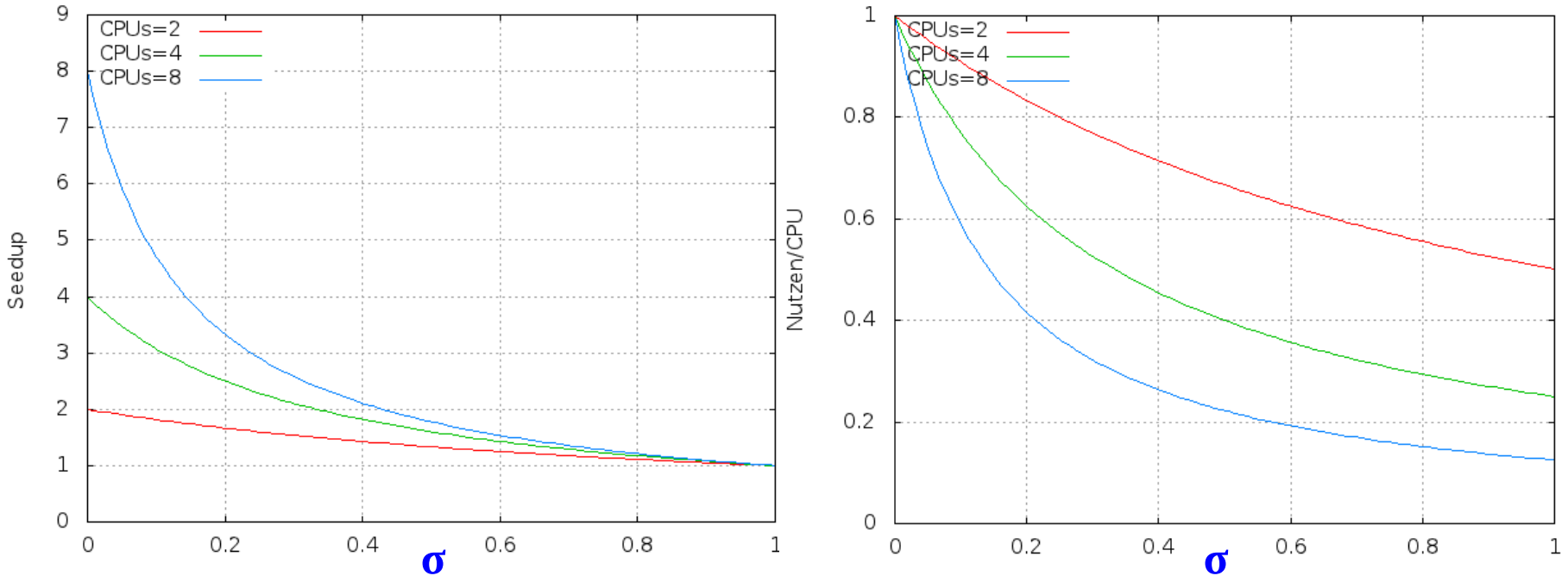
$$E(n) = \frac{S(n)}{n} = \frac{1}{\sigma n + (1-\sigma)}$$





Parallelisierung

- Anstatt $S(n)$ und $E(n)$ als Funktion der Anzahl n an CPUs zu zeichnen, wird hier der Speedup und Nutzen als Funktion $S(\sigma)$ und $E(\sigma)$ des seriellen Anteils für Duo-, Quad- und Oktacore Rechner gezeigt.



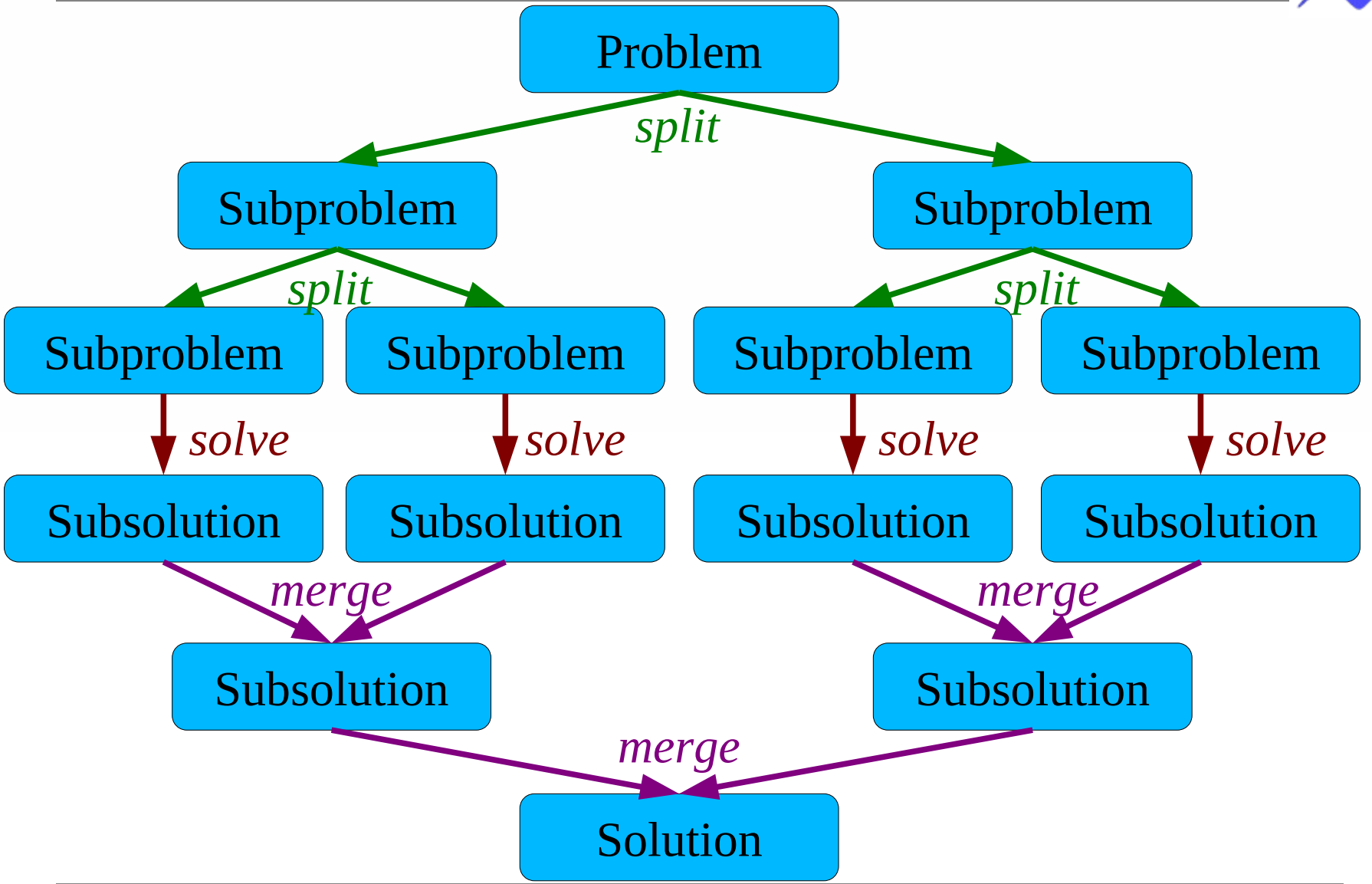
Voraussetzungen

- Das Problem muss eine Lösung durch einen parallelen Algorithmus zulassen.
- Der zu erwartende Nutzen muss den Aufwand zur Entwicklung des parallelen Algorithmus überwiegen.
  - Meistens lohnt sich dies nur bei „großen Problemen“.
- Die Hardware und die Programmiersprache müssen Parallelverarbeitung (PV) ermöglichen.
  - Alle heutigen Multicore CPUs sind für PV geeignet.
  - Sprachen wie FORTRAN, C++, C# oder Java werden für PV erweitert, bzw. entsprechende Bibliotheken ausgeliefert.
  - Funktionale Sprachen wie Scala, Haskell, Erlang, Lisp erlauben automatische Parallelisierung durch den Compiler.

Divide et Imperia

- Viele sequentielle Algorithmen sind nach der „teile und herrsche“ Strategie entwickelt.
  - Häufig sind diese rekursiv implementiert.
  - Meist wird eine effektivere, iterative Lösung durch Auflösen der Endrekursion gesucht.
- Solche Algorithmen lassen sich jedoch auch effektiv parallelisieren und sind dann für Multicore CPUs oder verteilte Anwendungen geeignet. Einige Beispiele:
  - Diskrete FastFourierTransformation (dFFT).
  - Quicksort, Mergesort etc.
  - Cholesky Zerlegung, Matrixmultiplikation, etc.

Teile und Herrsche Strategie



Paralleles Teile und Herrsche

Teile und Herrsche zerfällt in drei Bestandteile.

- Split: Aufteilung des Problems.
- Solve: Lösen der Teilprobleme.
- Merge: Zusammenführung der Teillösungen.
- Dies entspricht genau der Struktur der Eingangsüberlegung: Program={Setup, Compute, Finalize}.
- Split und Merge sind sequentielle Anteile, Solve ist parallelisierbar.

Beispiel: Komplexe Multiplikation

- Das Produkt zweier komplexer Zahlen $u, v \in \mathbb{C}$

$$z = u * v \quad \text{d.h. } z_1 + j z_2 = (u_1 + j u_2) * (v_1 + j v_2)$$

- lässt sich parallel berechnen mit

$$\Re(z) = u_1 v_1 - u_2 v_2$$
$$\Im(z) = u_1 v_2 + u_2 v_1$$

- Werden die vier Multiplikationen parallel ausgeführt, so ergibt dies einen Speedup von ~3 – 4.

Beispiel: Mergesort

- Mergesort als rekursives Sortierverfahren:

```
void sort(int[] a, int start, int end) {
      if(start < end) {
          int mid = (start+end+1)/2;
          sort(a,start,mid-1);
          sort(a,mid,end);
          merge(a,start,mid,end);
      }
  }
```

- Die beiden sequentiellen ***sort*** Aufrufe können parallelisiert werden, das finale ***merge*** erfolgt dann nach Beendigung der beiden ***sort*** Aufrufe.

Beispiel: Matrizenmultiplikation

- Parallelisierte Version des Produkts zweier Matrizen

$$C = A * B \quad \text{d.h. } c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

- Unterteilung der Matrizen in 4 kleinere Submatrizen:

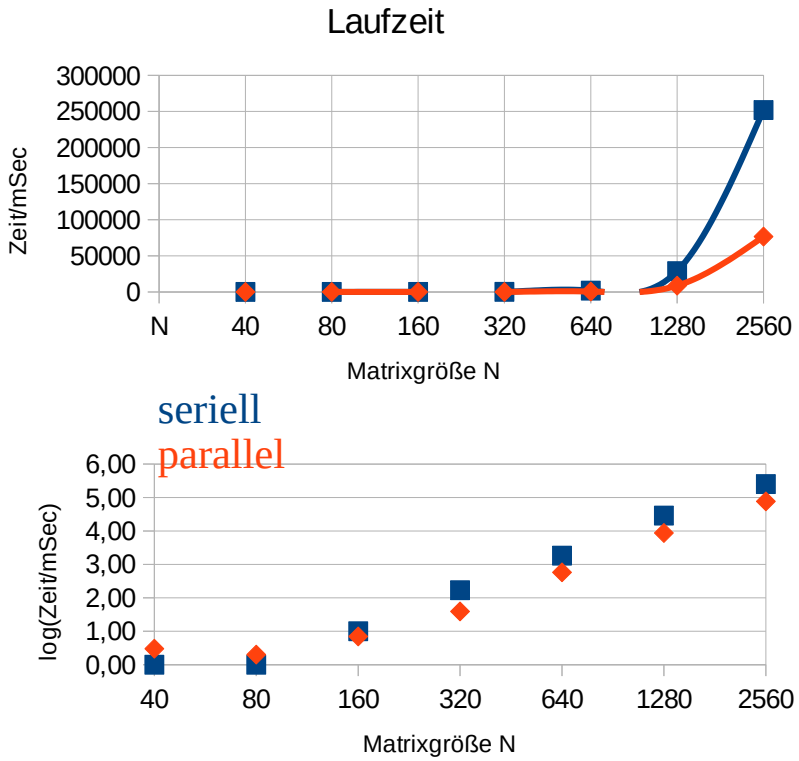
$$\left(\begin{array}{c|c} C_1 & C_2 \\ \hline C_3 & C_4 \end{array}\right) = \left(\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right) * \left(\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right)$$

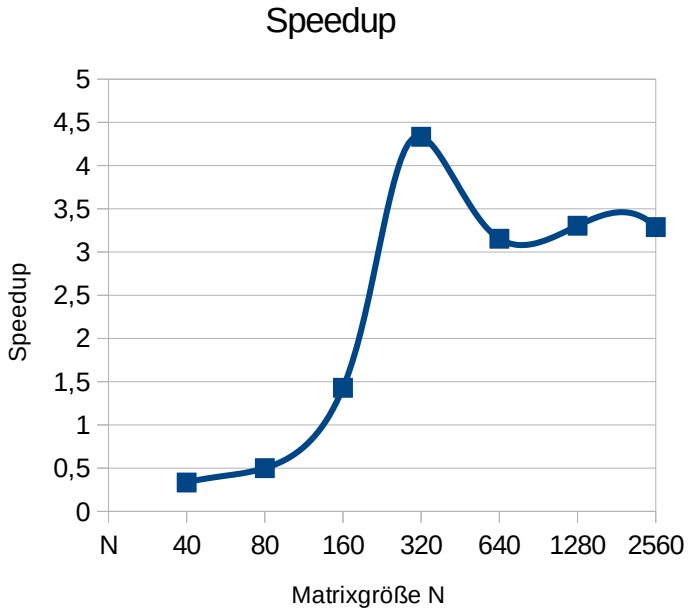
$$\begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix} = \begin{pmatrix} A_1 * B_1 + A_2 * B_3, & A_1 * B_2 + A_2 * B_4 \\ A_3 * B_1 + A_4 * B_3, & A_3 * B_2 + A_4 * B_4 \end{pmatrix}$$

- Dies sind acht unabhängige, parallelisierbare Produkte plus vier nachfolgende Additionen.

Testmessung

- Der Testlauf mit einem Quad-Core Prozessor zeigt deutlich den Geschwindigkeitszuwachs der Parallel-implementierung bei größeren Matrizen.





Zusammenfassung

- Durch Parallelisierung lassen sich Programme beschleunigen. Voraussetzung ist:
  - Die Hardware unterstützt dies.
  - Die Programmiersprache unterstützt parallele Konzepte.
  - Der Programmierer versteht parallele Konzepte :-) und
  - Es kann ein guter paralleler Algorithmus gefunden werden.
- Parallelisierung durch Verteilung auf verschiedene physikalische Knoten bietet meist nur Vorteile bei geringem Datentransfer/Synchronisierungsaufwand und dient mehr der Lastverteilung...